ChatGPT

Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** is an open standard introduced by Anthropic in late 2024 for connecting AI models (like large language models) to external data sources, tools, and services. In essence, MCP defines a **universal interface** that lets AI assistants securely access files, databases, APIs, and other resources beyond their built-in knowledge. By using MCP, an AI application (such as Anthropic's Claude or OpenAI's ChatGPT) can plug into external "contexts" – think of it as a **"USB-C for AI"** that standardizes how AI systems get information and perform actions in the real world.

This comprehensive tutorial will explain the key concepts of MCP, its history and features, and provide a hands-on look at how developers can integrate it into AI applications. We'll also explore how companies like Anthropic, OpenAI, Google, and PayPal are adopting MCP, and walk through examples using popular frameworks (Claude Desktop, LangChain, LlamaIndex, etc.) with MCP servers (Filesystem, SQLite database, etc.). By the end, you should understand not just *what* MCP is, but *how* to use it in practice.

What is a "Model", "Context", and "Protocol"?

Before diving deeper, let's clarify the terms in **Model Context Protocol**:

- **Model:** In this context, "model" refers to an AI model or assistant, typically a large language model (LLM) like Claude, ChatGPT, etc. It's the AI system that will be performing tasks or answering queries.
- **Context:** "Context" means external information or capabilities that the model can utilize. This could be data (files, database entries, web content) or tools (functions like search, calculators, APIs) that **augment the model's knowledge and abilities** beyond what's in its trained parameters. For example, a calendar database provides context about your schedule, or a calculator tool provides context for arithmetic tasks.
- **Protocol:** A protocol is an agreed-upon format and set of rules for communication. MCP is the protocol that defines **how** the model (client) and the context provider (server) talk to each other – the language and message patterns they use to exchange information and requests.

So **Model Context Protocol** literally means a standardized way for an AI model to communicate with context providers. MCP formalizes how an AI assistant can *ask* for external info or perform actions, and how the external system should *respond*. Both sides follow MCP's rules, ensuring interoperability across different models and tools.

History and Transition to MCP

MCP emerged to solve a growing problem in the AI ecosystem: language models were powerful in reasoning and generation, but **isolated from external data and tools**. Before MCP, if a developer wanted a chatbot to access some data (say, a company database or an API), they often had to write custom integration code or use vendor-specific plugins. This led to a proliferation of one-off solutions – each AI system had its own way to connect to each data source, creating an N×M integration headache (N AI apps times M data sources).

Earlier Approaches: In 2023, some partial solutions appeared. OpenAI introduced *function calling* in their API, letting developers define specific functions the model could call (like "search(keyword)" or "get_weather(city)"). OpenAI also enabled **ChatGPT plug-ins**, where the model could use approved web APIs via a plugin interface. These were useful but **vendor-specific** – a ChatGPT plugin wouldn't work with other models, and each API needed its own bespoke plugin or function definition. Similarly, frameworks like LangChain and others provided libraries for tool usage, but not a cross-vendor standard.

Anthropic's Initiative: On November 25, 2024, Anthropic announced MCP as an **open-source, open standard** to generalize these ideas. They described it as a way to replace fragmented integrations with one universal protocol. MCP was explicitly inspired by the success of standard protocols in other domains – Anthropic engineers likened it to the Language Server Protocol (LSP) in software development, which standardized how code editors talk to programming language backends. In fact, MCP **reuses the message flow of LSP and uses JSON-RPC 2.0** as the underlying message format (we'll explain JSON-RPC shortly).

Transition and Adoption: MCP quickly gained traction in 2025. Major AI providers started backing this "USB-C for AI" approach. In March 2025, OpenAI announced it would adopt MCP across key products – the ChatGPT desktop app, their forthcoming Agents SDK, and more [1]. Around the same time, Microsoft demonstrated integration of MCP with its Semantic Kernel toolkit and with Azure OpenAI services. In April 2025, Demis Hassabis of Google DeepMind confirmed that Google's next-gen model **Gemini** and related infrastructure would support MCP. This means even fierce competitors recognized the benefit of a common standard. The industry buzz around MCP grew – a debate ignited on social media about whether MCP was a fleeting fad or a future standard, with prominent developers from LangChain and other frameworks weighing in. MCP projects on GitHub started trending one after another in early 2025, signaling developer excitement.

By late 2025, MCP has evolved into a **consensus standard** for tool/data integration. It's governed openly (with a public spec and community contributions), not locked to one vendor. This open approach has made it attractive even in traditionally closed ecosystems. For instance, Cloudflare supports deploying MCP servers at the edge, and numerous third-party libraries have appeared to extend MCP in various environments. In summary, MCP arose to connect AI to the world, and its rapid uptake shows it is likely here to stay as a key piece of AI infrastructure.

Key Features and Benefits of MCP

What can MCP enable? In practical terms, MCP allows AI assistants to do things that were previously difficult or impossible without custom coding. For example:

- *Personalized assistants:* An agent (LLM) could access your **Google Calendar** or **Notion** notes through MCP connectors, allowing it to answer questions about your schedule or take notes on your behalf.
- *Multimodal creative tasks:* A code-generation assistant like Claude's "Code" could connect to design tools (e.g. a Figma MCP server) to build an entire web app UI that matches a design mockup.
- *Enterprise analytics:* A chatbot in a company can hook into multiple **databases** via MCP, enabling natural language queries that join data across silos (e.g. "What were our top 5 products by revenue last quarter?") and getting live answers from those DBs.
- *Physical world actions:* AI models could even control devices – e.g. an MCP server for a 3D printer or IoT device would let the AI create a design and instruct a 3D printer to print it.

These examples highlight that MCP isn't just about reading data – it's about **taking actions** through tools, under user guidance.

Why does MCP matter? It brings several big benefits, depending on who you are:

- **For developers:** MCP dramatically **reduces development effort** and complexity. Instead of writing a custom plugin for each model or each integration, you can implement a standard MCP server once and any MCP-compatible AI app can use it. This saves time and fosters code re-use. Essentially, MCP turns the N×M problem into N+M – you build an integration once, and many models can use it.
- **For AI applications (agents):** MCP opens up a whole **ecosystem of tools and data**. An AI app immediately becomes more powerful and useful if it can tap into external knowledge sources and functionalities. Using MCP is like giving an AI agent a swiss-army knife – it can search information, call APIs, fetch documents, etc., all in a controlled, standardized way. This improves the end-user experience because the AI can do more than just chat – it can act.
- **For end-users:** Users ultimately get **more capable and personalized AI**. MCP means your AI assistant can work with *your* data (with permission) and on *your* behalf. Imagine asking an assistant, "Please analyze this spreadsheet from my finance system," and it can, because it has a secure connector to that system. Or it can book flights for you because it has a tool for that. End-users benefit from AI that isn't stuck in a silo – it's an **"action-oriented" AI** that can interface with the real world (but safely, under oversight).

In summary, MCP's key features include: - A **universal interface** (standard JSON-based messages) for reading files, querying databases, invoking API calls, etc., regardless of platform. - **Secure, two-way connections**: The protocol is designed to be secure and requires user approval for actions. Servers expose capabilities but cannot access the AI's internal data unless allowed, and the AI must get user consent to perform potentially sensitive operations. - **Context sharing across tools**: As the ecosystem matures, AI systems will maintain context as they move between different tools and data sources. For example, an AI can use one tool to retrieve a document and another tool to send an email summary of it – all in one seamless flow, preserving the conversational context. - **Open-source reference implementations**: At launch, Anthropic provided a repository of pre-built MCP servers for popular services (Google Drive, Slack, GitHub, Git, Postgres, Stripe, etc.). These serve as both ready-to-use connectors and examples for developers. The open community means many more have been contributed since (from web search tools to PDF readers – even a "Zapier" integration that exposes thousands of apps through one MCP server). - **Tool discovery & dynamic updates**: MCP allows the AI client to discover what tools/resources a server provides (we'll see how via `tools/list` calls) and even handle tools appearing or disappearing during a session. The protocol supports **notifications** so the server can inform the client of changes (e.g. "a new tool is now available"). This makes AI agents very flexible in dynamic environments.

Overall, MCP's benefit is neatly summarized by Ars Technica: it's like giving AI a universal *port* to plug into anything – and it's doing so in a **vendor-neutral, collaborative way** that brings the AI industry together on a common standard.

A Quick Introduction to Anthropic & Claude

Anthropic is the AI research company that initiated MCP. Founded in 2021 by former OpenAI executives Dario and Daniela Amodei, Anthropic focuses on creating large language models with an emphasis on safety and reliability. Their flagship product is **Claude**, an AI assistant similar to ChatGPT. Claude is known for its ability to handle very large context windows (it can take in long documents) and for features like "Constitutional AI" which aim to make its outputs safer. Anthropic has positioned Claude as

an AI that can be integrated into workflows – from chatting and coding (Claude has a "Claude Code" for programming help) to enterprise tasks.

It's important to mention Claude because Anthropic essentially built MCP to enhance Claude and similar AI systems. **Claude was among the first AI assistants to support MCP natively.** In fact, Anthropic baked MCP support into the **Claude Desktop app** (discussed below) early on, so Claude could easily connect to local or remote data through the new protocol. Anthropic open-sourced MCP not just for Claude, but to kickstart an industry-wide standard. They garnered early support from companies like Block (Square) and tools like Replit, Sourcegraph, etc., to prove MCP's utility across different domains.

To put it simply, Anthropic's vision for Claude is an assistant that **isn't confined** to what it was trained on. Claude should be able to use your tools, read your files, and act in your world – safely and with permission. MCP is the conduit to achieve that. Anthropic's leadership in proposing MCP is a strategic move to ensure their AI (Claude) can seamlessly interact with external systems, leveling the playing field with other AI providers and setting a standard others are now adopting.

Why Anthropic is Mentioned in the Context of MCP

Anthropic is mentioned so prominently with MCP because **they created it** and drove its initial development. The *Model Context Protocol* was introduced by Anthropic's team (with help from collaborators) and open-sourced in November 2024. All the initial resources – the MCP specification, reference server code, SDKs in multiple languages – were released via Anthropic's official channels (the MCP website and GitHub org).

In essence, Anthropic donated MCP as a public good to solve an industry problem. They could have kept it proprietary for Claude, but instead they chose an open model, which encouraged even competitors to join in. As a result, when you talk about MCP, Anthropic's name naturally comes up as the protocol's originator and one of its chief advocates.

Another reason Anthropic is mentioned: they immediately integrated MCP into their products. The **Claude Desktop** application gained MCP support at launch, meaning any Claude user (especially enterprise "Claude for Work" users) could start connecting Claude to internal systems through MCP servers. Anthropic also used their latest model versions (Claude 3.5, code-named "Sonnet") to help developers generate MCP server code quickly – effectively dogfooding their own tech.

Finally, Anthropic's push for MCP is part of a broader strategy of collaboration and safety. By standardizing tool use via MCP, it becomes easier to **audit and control** what tools an AI can use, which is important for safety. Anthropic often frames MCP as a way to make AI more useful *and* keep it grounded (since it can fetch real facts instead of hallucinating).

In summary, Anthropic is mentioned with MCP because they are to MCP what, say, Microsoft was to the Language Server Protocol – the initiator and early champion. But today MCP has grown beyond any single company, with OpenAI, Google, Microsoft, and many others contributing. Anthropic lit the spark, and the industry fanned the flames.

How MCP is Trending on GitHub ☆

The response from the developer community to MCP has been extremely enthusiastic. On GitHub, MCP-related projects have skyrocketed in popularity: - The official **MCP reference servers repository** (which

contains implementations for various tools) gained tens of thousands of stars within the first year of release. By late 2025 it has on the order of **70k+ stars** – a strong signal of interest (for context, that's as many stars as some of the most popular machine learning libraries!). This makes it one of the top trending AI repos of the year. - Many community-built MCP connectors and tools have appeared, often trending on GitHub weekly. Developers have created "awesome MCP" lists to catalog the growing ecosystem. There are MCP servers for everything from fetching stock prices, to controlling Minecraft, to hooking into Spotify – demonstrating both the versatility of the protocol and the creativity of the community. - In early 2025, as MCP buzz picked up, there was a notable debate on social media that ended up *boosting* its visibility. Key figures from frameworks like LangChain engaged in discussions about MCP's merits, and shortly after, **MCP projects were regularly in GitHub's trending charts**. One article noted that MCP's rapid uptake by OpenAI and Google had essentially settled the debate in favor of it being a "fixture" rather than a fad. - Besides the core MCP repos, projects enabling MCP in different contexts have lots of traction. For example, *LangChain's MCP adapter* and *LlamaIndex's MCP integration* (which we'll discuss) are being widely adopted by developers building AI agents, each with their own GitHub presence.

In short, MCP went from unknown to **mainstream open-source project** in a matter of months. It's now common to see devs on Reddit or Discord recommending "just use an MCP server for that" when someone asks how to give an AI access to a new data source. That mindshare is reflected on GitHub through thousands of stars and forks. The excitement is comparable to the early days of APIs or browser plugins – except here it's about plugging tools into AI. MCP's trending status on GitHub underscores that developers see real value in a standard that makes AI **more extensible and practical**.

(Fun fact: as of mid-2025, some MCP server packages even outpaced many ChatGPT plugins in usage, since they can be used across multiple AI platforms, not just one product.)

Core Architecture of MCP

MCP follows a **client-server architecture** with clearly defined roles and layers. Let's break down how it's structured:

- **MCP Server:** This is a program that provides access to some external data or functionality via MCP. For example, a "Filesystem Server" might expose tools to read/write files, or a "Calendar Server" might expose a user's calendar events. The server runs as a separate process (which could be on the user's machine or a remote cloud) and listens for MCP requests.
- **MCP Client:** This is a component (usually a library) that runs within the AI application (host) and communicates with an MCP server. The client opens a connection, sends requests (like "list available tools" or "call this tool with these args"), and receives responses. Each MCP server that the AI connects to has its own dedicated client instance [2].
- **MCP Host:** The **host** is essentially the AI application or agent environment that is using the MCP client(s). It "hosts" the AI model and orchestrates its interactions. For example, **Claude Desktop** is an MCP host – it can launch clients to connect to various servers. Visual Studio Code with an AI plugin could act as a host, or a LangChain agent script can be a host. The host coordinates between the AI model and one or more MCP clients (and through them, servers) [3].

Relationships: An MCP host can connect to multiple servers at once. It will create one client per server (one-to-one mapping) [2]. For instance, imagine VS Code (host) connects to a Sentry error-logging MCP server *and* a local filesystem MCP server. VS Code spawns two MCP clients, each maintaining a connection to one server [4]. The AI model in the host (say a coding assistant) can then utilize both servers' tools. From the server perspective, it doesn't matter whether the client is Claude Desktop or VS Code or anything else – it just speaks MCP.

Local vs Remote Servers: MCP doesn't care where the server runs; it defines the communication. A server might run locally (on the same machine as the host) or remotely (on some URL). The **transport mechanism** differs (we'll detail STDIO vs HTTP/SSE next), but the logical protocol is the same. For example, Claude Desktop can **launch a local server** process for, say, filesystem access using a direct STDIO pipe. In contrast, connecting to an online service (like a Sentry cloud MCP server) would use a network connection (HTTP + streaming).

Layered Architecture: MCP's design is layered into two primary layers – the **Data layer** and the **Transport layer**:

- **Data Layer:** This is the core of MCP – it defines the **JSON-RPC 2.0 based message protocol** for all MCP interactions. It includes:
- *Lifecycle management:* how client and server establish a connection, negotiate capabilities, and terminate gracefully.
- *Primitives (Server-offered features):* "**Tools**" (actions the server can perform at the AI's request), "**Resources**" (data sources the server can provide, like documents or database records), and "**Prompts**" (pre-defined prompt templates or instructions the server offers). We'll dive into these primitives soon.
- *Client-initiated actions:* The AI (client) can request to call a tool, read a resource, etc.
- *Server-initiated actions:* MCP interestingly allows reverse requests – servers can ask the client/host to do things like "please get user input" or "have the model generate text given this prompt" or log something. This is a nod to interactive workflows (imagine a tool that occasionally needs the user to confirm something – the server can trigger a user prompt via the client).
- *Utility messages:* such as **notifications** (server sends an event to client without expecting a response) for things like "a tool list changed" or "progress update 50% done".

Essentially, the data layer is the *what* – what messages and commands are exchanged.

- **Transport Layer:** This defines *how* those messages get delivered between client and server. MCP deliberately separates transport so that it can work in different environments. The standard defines two transports:
- **STDIO Transport:** Uses standard input/output streams for communication. This is typically used for local servers. For example, if the host spawns the server as a subprocess, they can just read/write JSON messages via pipes. STDIO transport has virtually no latency or network overhead (it's just inter-process comm on the same machine) and is very fast. It's analogous to how LSP works between an editor and a language server.
- **Streamable HTTP Transport:** Uses HTTP for requests, combined with **Server-Sent Events (SSE)** for real-time streaming from server to client. In practice, this means the client will POST requests to the server's HTTP endpoint, and the server will stream responses or events back over an SSE channel. This transport is ideal for remote servers (host and server aren't on the same machine) and aligns with web standards. It also allows using standard web auth mechanisms (bearer tokens, API keys, etc.) for securing the connection. For instance, an MCP server running on a cloud service might require an API key – the client can include that in an HTTP header.

Both transports achieve the same outcome: they ferry JSON-RPC messages back and forth. STDIO is simpler but limited to local use; HTTP+SSE is more complex but enables networked scenarios and streaming.

Putting it Together: When an MCP client connects to a server, the sequence is typically: 1. **Initialization handshake:** They exchange info about themselves and what features they support. For example, the server advertises which primitives it offers ("I have tools and resources, and I send

notifications for tool changes"), and the client does similarly ("I as a client support receiving logs or taking user input if you ask"). They also exchange IDs or versions (clientInfo, serverInfo) – akin to a "hello" with capabilities negotiation. 2. **Discovery phase:** The client will query what tools/resources the server has. For example, it sends `tools/list` and gets back a list of available tools plus metadata about each. This is crucial so the AI knows what actions it *can* do. The response might include human-readable descriptions, input parameter types, etc., for each tool. 3. **Normal operation:** Now the AI (via client) can call tools (`tools/call` requests) to perform actions [5]. It can request resources (`resources/read`, etc.) to get data. The server executes those and returns results. Everything is exchanged as JSON objects following JSON-RPC (with method names like "tools/call", parameters, and result fields). 4. **Notifications/events:** If something changes on the server side or a long process is ongoing, the server can send notifications. For example, if a new tool becomes available mid-session, the server could send a `notifications/tools/list_changed` event to the client. The client (host app) upon receiving that might refresh its tool registry and inform the AI that it has an updated set of abilities. This event-driven aspect means the AI can adapt in real-time (no need for constant polling). 5. **Termination:** Either side can close the connection politely (there are messages for shutting down) or if the host app closes, it ends. Multiple sessions can be managed independently if multiple servers are connected.

The architecture emphasizes **modularity**: any MCP-compliant client can talk to any MCP-compliant server. This decoupling is powerful. For instance, you could replace an old "filesystem server" with an improved one, and your AI app wouldn't need code changes – it would just discover slightly different tools via MCP. Or an AI app might use the same `@modelcontextprotocol/server-filesystem` package whether it's Claude Desktop or another host – consistency for both devs and users.

To visualize: *the MCP client-server pair is like a translator sitting between the AI model and an external resource.* The model says (via the client) "I need data X" in MCP language; the server translator knows how to get X from the resource, then hands it back in MCP language. Neither the model nor the resource needs to know the other's details – they meet at MCP.

Key Components of MCP – Deep Dive into Primitives

Now let's take a closer look at the **key components (primitives)** that MCP servers provide, and how they work:

- **Tools:** In MCP, a **tool** is an executable function or action that the AI can invoke via the server. Tools are analogous to "skills" or API endpoints. Examples of tools: `search(query)` on a web search server, `send_email(to, subject, body)` on an email server, `query_db(sql)` on a database server, or `get_weather(city)` on a weather server. Each tool has a name and typically a description, plus a defined input schema (parameters) and output schema. Tools are used for **AI actions** – when the model decides it needs to do something to fulfill the user's request, it will call a tool via MCP.
- *Discovery:* The client uses `tools/list` to get all available tools from a server. The server responds with a JSON array of tool objects, each including fields like `name`, `description`, `parameters` (expected input format), and possibly `output_format`. This lets the AI reason about what tools are at its disposal. For instance, the server might list a tool named `"add_numbers"` with description `"Adds two numbers"`, and parameters `{a: number, b: number}` – so the AI knows it can call `"add_numbers"` if needed for arithmetic.
- *Invocation:* To use a tool, the client sends a `tools/call` request with the tool's name and arguments [5]. The server executes the corresponding function and returns the result or an

error. The AI model (especially those with function calling capabilities like GPT-4, Claude, etc.) often will decide to call a tool when its internal reasoning says "I need this external info". The MCP client/host can either feed the tool result back into the model's context, or (in an autonomous agent scenario) the model's planner will continue the conversation after getting the tool output.

- *Example:* Suppose a user asks, "What's the population of the city where the current temperature is below 5°C?" The AI might break this down and use a weather tool then a wiki tool. With MCP, it could have a `weather.get_current_temperature(city)` tool and a `wiki.search(query)` tool from two servers. It calls `weather.get_current_temperature` for many cities to find ones <5°C, then calls `wiki.search` on those city names to find their populations. Each call is an MCP `tools/call` under the hood.

- *Dynamic tools:* Tools can appear or disappear if the server's capabilities change. MCP supports sending a `tools/list_changed` notification from server to client to signal that the set of available tools was updated. The AI host would then refresh by calling `tools/list` again. This is useful in long-running sessions where available actions might evolve (or based on user permissions toggling something). The client doesn't have to poll; it reacts to notifications.

- **Resources:** A **resource** in MCP represents data that the server can provide access to, typically in a read-only fashion. You can think of resources as *files, documents, or other chunks of content*. For example, a filesystem server might expose a directory or file as a resource, a database server might expose a particular table or query result as a resource, or a Google Drive server might expose a document. Resources are meant to provide **context data** to the AI.

- Servers usually implement methods like `resources/list` (to enumerate available resources or their identifiers) and `resources/read` (to retrieve the content of a resource, possibly in chunks or with filters). For instance, the AI could do `resources.list(directory="/Users/Alice/Documents")` to get file names, then `resources.read(file_id=123)` to get the content of a specific file.
- In practice, the host UI often helps the user select resources to feed into the conversation. E.g., Claude Desktop after connecting a filesystem server lets the user *attach* a file from their computer into the chat – behind the scenes, that triggers an MCP resource read and provides the file content to Claude as context.
- Resources often come with **metadata** (like name, size, type) and may require pagination for large content. But from the AI's perspective, a resource is something it can ask to read or list.

- *Example:* An enterprise chatbot connected to a Confluence MCP server might let the user ask, "Summarize the document *TeamPlan2025*." The AI, via MCP, finds the resource (document) by name and reads its content, then generates a summary. Without MCP, you'd need a custom integration or to paste the doc text manually; with MCP, it's a standard operation.

- **Prompts:** A **prompt** is a less obvious but useful primitive. Prompts in MCP are pre-defined pieces of text or instructions that the server can supply to the client or vice versa. Essentially, a prompt is like a template for the AI. For example, a server might have a prompt that provides a few-shot example or guidance for using its tools effectively. A "Database" server might have a prompt with system instructions like "When querying the database, always format your query as SQL and double-check column names." The AI can incorporate these to improve its performance with that server.

- Prompts can be thought of as **stored context or canned instructions** on the server side. The client might fetch them via something like `prompts/list` or a server might send a prompt as part of initialization.
- This mechanism allows tool providers to supply the AI with some "know-how" on using the tool, without hardcoding it into the AI. It's an advanced feature and not used in every server, but it's part of MCP's design for completeness.

- *Example:* An MCP server for a complex API might include a prompt that shows an example conversation of how to format requests to that API. When the client connects, it could retrieve that prompt and insert it into the AI's context, so the AI is primed to use the API correctly.

- **Notifications:** As touched on earlier, MCP supports server-initiated notifications (which are JSON-RPC notifications – a message with a method but no response expected). Notifications allow the server to **push updates** to the client in real time. Key uses:

- **Tool/Resource list changes:** as described, inform the client that it should refresh available items.
- **Progress updates:** if a tool action is long-running (e.g., a "download_file" tool might take 10 seconds), the server could send periodic notifications like `{"method": "notifications/progress", "params": {"tool": "download_file", "percent": 50}}`. The client/host might show a progress bar or just hold the thought until completion.
- **New data events:** imagine a server monitoring a real-time data feed. It could send notifications of new entries (like a new email arrived in your inbox) to the AI. The AI could potentially decide to surface that to the user or take some action. (Of course, such behavior needs careful design to not overwhelm or distract, but MCP provides the capability.)
- The client does not answer notifications; it just processes them. The MCP spec defines certain standard notification types (like `tools/list_changed`, `resources/updated`, etc.). Notifications make the integration **event-driven and reactive** rather than purely request-response, which is powerful for maintaining up-to-date context.

To summarize these components: **Tools, Resources, Prompts, Notifications** are the core abstractions an MCP server can offer. Not every server has all of them – many servers have only tools, or only resources, depending on purpose. For instance, a **Calculator MCP server** might have just one tool (`calculate(expression)`) and no resources or prompts. A **Wiki document MCP server** might primarily offer resources (articles) and maybe a search tool. But the client-side logic is uniform: discover what's available, then use it.

One more component worth noting is **permissions and approval**. MCP itself is an open pipe, so it relies on hosts to enforce user permissions. Many hosts (like Claude Desktop) implement a rule that **any action that changes user data or is sensitive requires user approval**. For example, the Filesystem server will not actually delete a file or write a file unless the user clicks "Allow" in the UI when Claude tries to invoke such a tool. This ensures that even though the AI *could* execute powerful tools, it cannot do so autonomously without oversight, preserving user control. The servers themselves also often implement internal safety, like the SQLite server runs by default in read-only mode to prevent unintended writes [6].

In conclusion, MCP's key components create a rich protocol that goes beyond simple question-answer. They allow for a **workflow**: an AI can list its abilities, reason about them, use them to fetch data or effect changes, get new info in real-time, and do so in a controlled, auditable way. For developers, learning these primitives is the key to building or using MCP integrations effectively.

What is JSON-RPC 2.0 (and why MCP uses it)?

JSON-RPC 2.0 is the communication format at the heart of MCP's data layer. JSON-RPC is a lightweight Remote Procedure Call protocol that uses JSON for encoding messages. In plainer terms, it's a way to send a request as a JSON object like `{"method": "...", "params": { ... }, "id": 123}` and to get a JSON response `{"result": ... , "id": 123}`. It's called "RPC" because it's like calling a function (method) on the remote system, with parameters and a return value, but over a data connection.

Some key points about JSON-RPC 2.0: - It is **stateless and transport-agnostic**. The protocol itself doesn't assume HTTP or any specific transport – which made it a great fit for MCP, since MCP runs over STDIO or SSE/HTTP depending on scenario. JSON-RPC just says: you send JSON messages and get JSON responses, and if you send a "notification" (a message without an id), no response is expected. - It's **simple**: JSON-RPC 2.0 has only a few elements – `method` (a string naming the action), `params` (arguments data), `id` (to match responses to requests), and `error` (if something goes wrong). There's no elaborate schema, making it easy to implement in many languages. The simplicity is by design: *"It is designed to be simple!"* as the spec states. - It supports **batching** (you can send multiple requests in one go if needed) and **out-of-order responses** (responses can come back in any order, they're matched by id). - Why not REST or gRPC or GraphQL? MCP designers chose JSON-RPC likely due to its success in the Language Server Protocol and similar use cases. JSON-RPC is lighter weight than full HTTP REST calls for each action (fewer bytes overhead and no need to define lots of endpoints – methods are just names). It's also easier to stream with (especially using SSE, where the server can push partial results or multiple results in sequence for one request). - In practice, JSON-RPC 2.0 in MCP might look like: - Client to server: `{"jsonrpc": "2.0", "id": 42, "method": "tools/list", "params": {}}` - Server to client response: `{"jsonrpc": "2.0", "id": 42, "result": { "tools": [ ... ] } }` - Server to client notification (no id): `{"jsonrpc": "2.0", "method": "notifications/tools/list_changed", "params": { ... }}`

Each message is a standalone JSON object. By using JSON, it's human-readable (for debugging) and easily consumable by any programming language (virtually every language has a JSON parser). The "2.0" version of JSON-RPC is the modern one that MCP uses (there was an earlier 1.0 but it lacked some features like differentiation of notifications).

To summarize, JSON-RPC 2.0 is like the **grammar** of the MCP conversation. It's a well-proven, minimalist protocol for RPC that fits the needs of communicating between AI clients and servers. Its benefits of being stateless and lightweight make it easier to implement secure and fast interactions. If you've worked with JSON-based APIs or function-calling in OpenAI, JSON-RPC will feel natural – it's essentially packaging up a function call in JSON format.

MCP Transport Layer: STDIO vs HTTP(SSE)

As mentioned in the architecture overview, MCP supports two transport mechanisms, **STDIO** and **Streamable HTTP (with SSE)**, each suited to different scenarios [7]. Let's compare and understand them:

1. STDIO Transport (Standard I/O): This uses the standard input/output streams of processes for communication. In practice, the MCP client will spawn or attach to the MCP server process and then read from its stdout and write to its stdin. The JSON-RPC messages flow through this pipe.

- **Use case:** Primarily for *local MCP servers*. When the server runs on the same machine as the host app, STDIO is ideal. For example, Claude Desktop launching a local Filesystem server uses STDIO by default. The Desktop app starts the server process (perhaps a Node.js script or a Python program) and then exchanges messages via pipes. No network ports, no HTTP – just an internal connection.
- **Advantages:** Speed and simplicity. STDIO has **minimal overhead** – it's just local IPC. There's no HTTP parsing beyond the JSON itself. Latency is extremely low, which is good if the AI is making a lot of rapid tool calls. It's also inherently secure in that no external entity can intercept, since it's not listening on a network socket.
- **Considerations:** STDIO is not suitable if the server is on a different machine. It also requires the host to manage the process lifecycle. Additionally, STDIO doesn't naturally support multiple simultaneous requests (though an MCP client can still send multiple JSON-RPC messages, they'd just be processed sequentially by the single server process).
- **Example:** If you build a custom MCP server called "MyTool" in Python, you can distribute it as a script. A user's AI app could run `mytool_server.py` and talk via STDIO. If you open the process's stdout in a terminal, you'd literally see JSON flying by when the AI uses it.

2. HTTP + SSE Transport: This is essentially a **web API** approach. The server runs an HTTP server. The client establishes a long-lived connection for receiving events (SSE – Server-Sent Events) and uses HTTP requests to send commands.

- **Use case:** For *remote or distributed MCP servers*. If you want to host an MCP server on a cloud service or allow multiple clients to connect, HTTP is the way. For example, Anthropic provides connectors to third-party SaaS (like Sentry, GitHub) which run as cloud services – these use HTTP+SSE so any client can reach them over the internet. Also, if you build a personal MCP server and want to share it with colleagues, running it as a web service with HTTP+SSE allows that.
- **Server-Sent Events (SSE):** SSE is a mechanism over HTTP where the server can push a stream of messages to the client after the client opens a special endpoint. It's simpler than full WebSockets and fits well for one-way real-time data (server -> client). MCP uses SSE to stream outputs (e.g., token-by-token generation if a tool involves LLM, or intermediate results) and to send notifications/events. The client will typically do a GET request to something like `http://server-url/sse` and keep that connection open to listen for events.
- **HTTP requests:** The client likely calls a POST endpoint for requests. For example, to send a `tools/call`, it might POST the JSON to `http://server-url/jsonrpc`. The server processes and then sends the response back *via the SSE channel* (or possibly the HTTP response if it's quick, but often SSE is used especially if streaming). This pattern is sometimes called **long-polling or streaming responses**.
- **Authentication:** HTTP transport means you can layer on standard web auth. MCP servers often require an API key or token in a header (e.g., an `Authorization: Bearer <token>` header). For instance, a remote Google Drive MCP server might need an OAuth token to actually fetch your Google Drive data – the client would include that token in requests. The MCP spec doesn't dictate auth but encourages using existing HTTP auth schemes.
- **Advantages:** It allows *centralized* or *cloud* MCP servers. One server can serve many clients (each client would typically get its own SSE stream, possibly multiplexed by client ID). It also crosses platform boundaries easily – a JavaScript web app or a Python script can both hit the same HTTP MCP service.

- **Considerations:** There is overhead of HTTP (headers, etc.) and complexity of maintaining a streaming connection. Network latency also comes into play – a remote call might be, say, 100 ms latency instead of 1 ms locally. For many use cases (like database queries or web API calls) that latency is negligible compared to the action itself. But for very rapid-fire interactions, STDIO is faster. Additionally, if hosting an MCP server online, you must handle concurrency, security, uptime – essentially treat it like a mini web service.
- **Example:** The **Claude Custom Connector** interface allows users to connect Claude to a remote MCP server by providing its URL. Under the hood, Claude (the client) then initiates an SSE connection to that URL and begins sending JSON-RPC requests via HTTP. If the user's internet drops, that SSE will break – so remote connectors depend on network stability, whereas local ones do not.

Choosing STDIO vs HTTP: Many MCP servers are actually packaged to support both. For instance, the official reference implementations often allow running with `--server_type=stdio` or `--server_type=sse` (SSE meaning HTTP/SSE mode). As a developer, you choose based on the deployment: - If you are building a *local extension* (e.g., a tool meant to run on a user's PC with their app), STDIO is simplest. The user just installs your server and their MCP-compatible app can launch it. - If you are offering a *cloud service* (e.g., "MCP Weather API" that anyone can connect to via the internet), you'll run it in SSE mode behind some URL and perhaps provide a key for access. - Some scenarios might even use both: start developing locally with STDIO (easier to debug), later deploy remotely.

Interoperability: A great thing is that an AI client like Claude can handle both at once – you could have one connector using STDIO (local files) and another using HTTP (a cloud knowledge base) simultaneously. The user experience is unified in the host app.

In summary, STDIO vs HTTP(SSE) is about *where* the server lives and how you connect: - *STDIO = local, single-user, fast lane. - HTTP+SSE = networked, multi-user (potentially), standard web tech.*

MCP deliberately accommodates both, making it flexible. As a developer, you don't have to drastically change your server's logic – the SDKs usually abstract much of the difference. For example, the Python SDK's `FastMCP` can run in either mode with a flag; the main difference is just how requests come in and responses go out (pipe vs HTTP endpoint).

Claude Desktop – What Is It?

Claude Desktop is Anthropic's desktop application that allows users to interact with the Claude AI assistant on their own computer (available for Windows and Mac). It provides a ChatGPT-like chat interface, but with some extra powers tailored for integration and enterprise use. Specifically, Claude Desktop was built with **MCP connectivity** in mind from the ground up.

In simpler terms, Claude Desktop is like having a dedicated chat app for Claude AI, instead of using a web browser. This has a few advantages: - It can run in the background, keep chat history locally (for enterprise security), and offer offline-friendly features. - It allows **Desktop Extensions** – which are essentially one-click installations of MCP servers that run locally and connect to Claude. This means you can extend Claude's abilities on your desktop by enabling extensions for things like filesystem access, local databases, or other tools. It's analogous to how the ChatGPT app supports plugins, but here it uses MCP under the hood as the interface.

Claude Desktop's interface is similar to other chat UIs: a conversation pane where you chat with Claude, and settings for configuration. The unique part appears when you use MCP: - In the UI, connected MCP

servers show up often with a little “ ” or similar icon (indicating tools). For instance, after enabling the Filesystem extension, you might see an icon for file tools. - You can usually attach resources from MCP servers via an “attachments” or paperclip menu. For example, selecting files from your computer to share with Claude uses the Filesystem MCP server in the back end. - Claude’s responses will sometimes mention it’s using a tool, or the UI might show a step-by-step (some hosts show the reasoning, though I’m not sure if Claude Desktop does in the UI). - There is a **Connectors settings** screen where you can add remote connectors by URL or manage local ones [8]. This is where you configure MCP servers.

In effect, Claude Desktop serves as a **reference MCP host implementation for end-users**. It hides the technical details and provides a polished experience: - It can **automatically launch** local MCP server processes (via config or the extension directory) when it starts, so users don’t have to run command-line tools manually. - It handles **auth flows** for remote connectors. When you add a connector URL, if that server requires OAuth or an API key, Claude Desktop will prompt you through it (as described, it might pop up an authentication dialog or ask for an API key). - It offers a **debugging console/log** for MCP, helpful for developers. Claude Desktop writes logs for MCP connections (so you can see if a server connected or if any error occurred). It even has a feature called **MCP Inspector** for testing servers locally in a UI [9] – kind of like a Postman for MCP. - Being a native app, Claude Desktop can also integrate with system features. For example, it could open files, or integrate with other desktop apps through MCP.

To illustrate how Claude Desktop sits in the architecture, consider this diagram (from a scientific tools integration perspective):

Claude Desktop (the host) provides the user interface and reasoning (Claude AI model), and connects via the MCP protocol to local or remote MCP Servers that expose tools/data. In this example, Claude Desktop is connected to a “ToolUniverse” MCP server which provides over 600 scientific tools. This allows the user to interact with those tools through natural conversation, with Claude handling reasoning and the MCP server handling execution.

As shown above, Claude Desktop is essentially the **central hub** where the AI lives, and MCP servers are modular add-ons plugging into it.

Summary of Claude Desktop: It’s your desktop AI assistant app for Claude, enhanced by the ability to plug in connectors via MCP. If you have Claude on your machine, you can give it “superpowers” by enabling connectors – reading files, browsing your local code, querying databases – and all in a standardized, secure way. Anthropic provides it so that enterprise users (who may not want cloud-only usage) and power users can get more out of Claude. It’s comparable to OpenAI’s ChatGPT Plus Plugins + Desktop app combined, but built around the open MCP ecosystem.

Demo: Using Claude Desktop with an MCP Connector

Let’s walk through a hypothetical demo scenario to show **how Claude Desktop uses MCP in practice**. We’ll use a simple example: connecting Claude to a local filesystem and asking it to do something with a file.

Setup: Suppose you have Claude Desktop installed and running. You want Claude to be able to read and organize files on your computer via conversation. Anthropic provides a **Filesystem MCP server** for this purpose. In Claude Desktop, you’d do the following: 1. Go to **Settings > Extensions (Connectors)** within Claude Desktop. (In some versions, you might click your profile > Settings > Connectors.) There you’ll see available connectors. 2. Enable or install the **Filesystem** connector. Claude Desktop might list

“Filesystem (local)” as an option. Enabling it essentially configures Claude Desktop to start the Filesystem MCP server. 3. After enabling, Claude Desktop likely prompts you to allow certain directories. For example, you might edit the config JSON to specify which folders Claude can access (for safety). The configuration could look like this:

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "C:\\Users\\YourName\\Documents",
        "C:\\Users\\YourName\\Pictures"
      ]
    }
  }
}
```

This tells Claude Desktop to, on startup, run `npx -y @modelcontextprotocol/server-filesystem <Dir1> <Dir2>` (which launches the filesystem server with access to those two directories). You could also include environment variables or other options if needed.

1. Restart Claude Desktop (or maybe it auto-reloads config). Now, it will launch the Filesystem MCP server in the background (using STDIO transport because it’s local). If all goes well, you’ll see in Claude’s UI some indication that a new tool is connected – possibly an icon or just knowing that Claude can use file tools.

Using it: Now let’s **ask Claude to do something** that requires file access. For example:

User types: *“Claude, can you find all text files in my Documents folder that mention ‘project X’ and summarize them?”*

Claude (the AI model) receives this query. Under the hood, here’s what happens: - Claude thinks: It needs to list files in Documents and search their contents. Thanks to the Filesystem MCP integration, Claude is aware of tools like `filesystem.search` or has the ability to iterate files. - Claude formulates a plan: maybe first use a tool to search for files containing “project X”. The Filesystem MCP server indeed provides tools such as reading file contents or searching text. For instance, let’s assume it has: - `list_directory(path)` – lists files in a directory. - `read_file(path)` – reads file content. - (If it has a search built-in, maybe `find_in_files(query, path)` – some might.) - The Claude MCP client sends a request to the Filesystem server: `{"id":1,"method":"tools/call","params":{"name":"find_in_files","arguments":{"query":"project X","path":"C:\\Users\\YourName\\Documents"}}}` (This exact JSON is abstract, but conceptually.) - The Filesystem MCP server executes that: it scans files under Documents for the query. Suppose it finds 3 text files with matches. - It returns a result (perhaps a list of file paths or names with matches, maybe excerpts). - Claude receives that result via the MCP client. Now Claude knows which files are relevant. - Claude might then call another tool to get more detail, e.g., `read_file(filePath)` for each of those files, to gather content. - Each time, **Claude Desktop will prompt you for approval** because reading file content is sensitive. The UI likely pops up, “Claude wants to read file `ProjectX_Plan.txt`. Allow?”.

You click Allow (since you trust it for this task). - Once Claude has the content (maybe not the full, but it could if needed given Claude's large context), Claude generates summaries of each and then a combined summary.

Finally, Claude replies in the chat:

"I found three files that mention 'project X'. They are `ProjectX_Plan.txt`, `ProjectX_Design.txt`, *and* `MeetingNotes_ProjectX.txt`. *Here's a brief summary of each: - ProjectX_Plan.txt: (summary...) - ProjectX_Design.txt: (summary...) - MeetingNotes_ProjectX.txt: (summary...) In summary, Project X is described as ..."*

During this, the **Claude Desktop UI** might show an indicator when tools are being used. Some UIs show a spinner or text like "(Using Filesystem Connector...)". In the background logs you'd see JSON messages to `tools/list` (at startup) and `tools/call` for each action.

Additionally, if you click on the "attachments" icon in Claude Desktop's message box, you would see an option to attach a file from the connected filesystem (since the Filesystem server exposes your directories as resources). You could manually attach files to the chat – which effectively calls `resources/read` on them and dumps their content (or a reference) into Claude's context.

Here's another mini-demo scenario: - You ask: "Please write a poem about sunrise and save it to my desktop." - Claude crafts a poem (pure generation). - Claude then invokes the Filesystem tool to create a new file on Desktop and write the poem into it. - Before writing, Claude Desktop asks "Allow Claude to create file 'SunrisePoem.txt' on your Desktop?" You click Yes. - File is written via the MCP call. Claude responds, "I've saved a poem to your Desktop called *SunrisePoem.txt*." And indeed the file appears.

Screenshot: Adding a remote MCP connector in Claude Desktop. Here, the user is entering the URL of a remote MCP server in Claude's "Add custom connector" dialog. Once added and authenticated, Claude can use the tools and resources provided by that server, which will then appear in the attachment menu or be accessible to the AI.

The above image (imagine it shows a prompt for entering an MCP server URL) exemplifies how easy it is for a user to extend Claude via MCP – just plug in a URL. Compare this to previous systems where each integration might require installing a plugin or writing custom code.

In sum, using Claude Desktop with MCP connectors involves: - Enabling/installing the connector (local or remote). - Possibly configuring specifics (like which directories or API keys). - Engaging in conversation where Claude seamlessly uses the connector's capabilities to help fulfill your requests. - You as the user occasionally approving actions (ensuring safety).

From a user perspective, it feels like Claude "just knows how to do more things". Under the hood, it's the MCP mechanism doing the heavy lifting in a standardized way.

Enabling the Filesystem MCP Server in Claude Desktop

One of the first things many users try is giving their AI assistant access to local files. Let's detail how to enable and use the **Filesystem MCP Server** with Claude Desktop (some of which we touched on in the demo, but here more step-by-step):

Purpose of Filesystem MCP Server: It lets Claude list directories, read file contents, create or modify files, etc., on *your* machine (but only in areas you allow). Essentially, it's like giving Claude a very controlled file explorer and editor. This is hugely useful for tasks like code assistance (reading your code files to answer questions or make edits), document summary, search, and more.

Installation Requirements: - Claude Desktop installed (with latest version supporting MCP). - Node.js installed on your system, because the official Filesystem server is implemented in Node (and distributed via npm). You can verify Node by running `node --version` in a terminal. - (Optionally, the `uv` package manager that Anthropic suggests for running MCP servers, but not strictly necessary if using npx directly.)

Steps to Enable:

1. **Open Claude Desktop settings:** Click on the Claude icon in menu bar -> Settings..., then go to the **Developer** or **Connectors** section. There you should see an **"Edit Config"** button or a way to edit the configuration JSON for MCP servers.

2. **Edit** `claude_desktop_config.json`**:** Claude Desktop uses a JSON config file (path differs by OS: on Mac `~/Library/Application Support/Claude/claude_desktop_config.json`, on Windows `%APPDATA%\Claude\claude_desktop_config.json`). When you click Edit Config, it opens this file. If it's empty or not existing, it will start as an empty JSON object.

3. **Add Filesystem server config:** You need to insert a snippet under `"mcpServers"`. According to Anthropic's docs, the snippet looks like:

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/Users/yourname/Desktop",
        "/Users/yourname/Documents"
      ]
    }
  }
}
```

Replace `/Users/yourname/Desktop` etc. with the paths you want to allow access to. You can list multiple directories. In Windows format, that might be `C:\\Users\\YourName\\Desktop` (note JSON escaping). The `-y` auto-confirms the npm package install (so it will download `@modelcontextprotocol/server-filesystem` the first time). This config basically says: on startup, run `npx @modelcontextprotocol/server-filesystem <dir1> <dir2>...`.

(If you had Node installed via nvm or not globally, ensure `npx` is available in PATH. The troubleshooting notes say that if `npx` isn't global, one might need to install npm globally or specify an env path [10].)

1. **Save and restart Claude Desktop:** After editing, save the JSON. Then close and reopen Claude Desktop. Upon launching, it should read that config and attempt to start the filesystem server. You might see a quick notification or icon. If it fails, check the **MCP logs**:
2. On Mac: `~/Library/Logs/Claude/mcp.log` for general MCP connection info, and `mcp-server-filesystem.log` for server output.
3. On Windows: `%APPDATA%\Claude\logs\mcp.log` etc. If it started successfully, Claude Desktop's interface might show a little **hammer** 🛠 icon (just as a hint that developer tools are connected). Also, if you click the attachment (paperclip) icon in the chat input, you should now see your allowed directories listed, enabling you to pick files from them to share with Claude.

4. **Using the Filesystem server:** Now try a simple command to test:

5. Ask: "What files are in my Desktop folder?" Claude will use the `list_directory` tool of the filesystem server. Immediately, Claude Desktop will **request approval** for listing that directory (if it's implemented to require it). Once you allow, Claude will output the list of files.
6. Next, try: "Open the file `notes.txt` on my Desktop." Claude will call `read_file` on that file. Claude Desktop will pop up *"Claude wants to read notes.txt (size 2 KB). Allow?"* You allow it. Claude then provides the content or a summary, depending on what you asked.
7. Try creation: "Create a file `todo.txt` in my Documents and add the line 'MCP is awesome' to it." Claude will call a tool (maybe `write_file(path, content)` or a combination of create and write). You'll get an approval prompt: *"Allow Claude to create file todo.txt?"*. After allowing, it executes and Claude says "File created."
8. See the effect: that file actually appears in your Documents.

Claude Desktop with the Filesystem extension enabled. This screenshot shows Claude interacting with the local file system: listing files and requesting permission for a file operation (note the prompt asking the user to approve the action). With the filesystem MCP server connected, Claude can manage files on the user's behalf, under explicit user approval.

1. **Security & Approval:** It's worth emphasizing the security model: **Every potentially risky action requires your OK**. Reading a file, writing, deleting, moving – each time, Claude Desktop will ask you. The Anthropic docs note that you maintain full control: no file operations happen without your explicit permission. This is critical, as it prevents any misstep by the AI from causing havoc on your files. If you say "no" to a request, Claude will receive a rejection and typically apologize or say it cannot proceed with that action.

2. **Troubleshooting:** If the filesystem server isn't working:

3. Make sure Node is installed and accessible. Try running the same npx command manually in a terminal to see if it launches.
4. Check that your JSON syntax is correct (a missing comma could break the config).
5. Ensure the paths you provided exist and you have permission to access them. Relative paths won't work – use absolute paths.
6. Look at the logs for errors (e.g., missing package, path issues).
7. If on Windows and you see weird `${APPDATA}` in errors, a workaround is to provide an `env` entry in config as per docs (this is a quirk with some Node setups on Windows) [11] [12].

Once configured properly, the Filesystem MCP server can significantly enhance Claude's usefulness for tasks involving local data. It's like giving Claude read/write access to parts of your brain (your digital memory on disk) – under a leash.

What the Filesystem server provides: According to documentation, it has tools for: - Listing directory contents - Reading file content - Writing/creating files - Moving/renaming files - Searching for files by name or content (perhaps a grep-like tool) - Maybe deleting files (though maybe disabled or requires strong confirmation)

All these actions will result in user confirmation prompts in Claude Desktop, which is great. You wouldn't want an AI that accidentally deletes a file without asking! The design is such that **you are always the ultimate gatekeeper**.

After enabling the filesystem extension, many users report a "wow" moment: you can ask Claude things like "Search all PDFs in my research folder for mentions of quantum tunneling" and it can actually do it, delivering answers that combine its language ability with direct access to your documents. That's the power MCP unleashes.

Enabling the SQLite Database MCP Server in Claude Desktop

Another powerful integration is giving Claude access to query a database. **SQLite** is a lightweight, file-based database – many developers or applications use SQLite for local data storage. With the **SQLite MCP server**, you can allow Claude to run SQL queries on specified database files, enabling natural language Q&A or analysis on your data.

Enabling the SQLite MCP server in Claude Desktop follows a pattern similar to the filesystem:

Prerequisites: - Node.js installed (the SQLite server is also distributed via npm as `@modelcontextprotocol/server-sqlite`). - A SQLite `.db` file that you want Claude to access. For example, say you have `mydata.sqlite` containing some tables. - (Claude Desktop and Node as before.)

Steps to set up:

1. **Install the SQLite server package (optional):** You can either run it via npx each time, or install globally. For one-off usage, we'll use npx. If you want to ensure it's downloaded, you could run `npm install -g @modelcontextprotocol/server-sqlite` which installs it globally [13]. But npx with `-y` will auto-install anyway.

2. **Edit Claude Desktop Config:** Open the same `claude_desktop_config.json`. Under `mcpServers`, add an entry for SQLite, for example [14]:

```
"sqlite": {
  "command": "npx",
  "args": [
    "-y",
    "@modelcontextprotocol/server-sqlite",
    "C:\\path\\to\\your\\database.sqlite"
```

```
  ]
}
```

Replace the path with the full path to your `.sqlite` file. You can name the server key `"sqlite"` or something descriptive like `"chinook_db"` depending on use (if you had multiple). If you have multiple databases, you can add multiple entries or in some implementations you can list multiple files for one server (but typically one server = one database unless designed otherwise; however, the medium piece suggests multiple DBs can be configured by multiple server entries [15]).

Save the config. Now Claude Desktop will know to launch an MCP server for SQLite on startup.

1. **Restart Claude Desktop:** It should spawn the SQLite server via npx. The server will open the database file in read-only mode by default (according to its design) [6]. Check `mcp-server-sqlite.log` in the logs if any issues.

2. **Using it:** Suppose your SQLite database has tables like `Customers` and `Orders`. Now you can ask Claude questions about the data:

3. "How many customers are in the database?" – Claude will use the SQLite server's `query` tool to run a SQL query `SELECT COUNT(*) FROM Customers;`. The server executes and returns the result (say 123). Claude then replies: "There are 123 customers in the database."
4. "Show the names of the first 5 customers and their total order count." – Claude might compose a more complex SQL with a JOIN. This depends on Claude's capability to translate natural language to SQL, which it is generally good at if it knows the schema. The **SQLite MCP server provides schema information as resources** too [16] – meaning Claude can retrieve the list of tables and columns. Likely the SQLite server has a resource or a tool like `schema` that returns the schema. Claude might first get the schema (or it could be configured to have been provided on connection as a resource).
5. When Claude tries a query, the **SQLite server sees it's read-only** (by design it won't do modifications unless explicitly allowed). This ensures safety – the AI won't accidentally `DROP TABLE` or insert nonsense, at least by default [6].
6. If the query result is large, the server might cap it or paginate. The AI might then ask follow-ups or request only certain info.
7. As a user, you might see Claude step through reasoning (if you have chain-of-thought visible, but if not, it's behind the scenes). But results will appear as answers.

8. You can also explicitly instruct: "Use the database to answer: what's the total revenue?" and Claude will formulate an SQL.

9. **Security and Permissions:** Claude Desktop might also require confirmation when the AI performs a query. If the server or host classifies queries as actions needing approval, you would get a prompt like *"Allow Claude to execute a query on database.sqlite: SELECT ...?"*. This might depend on settings; querying data might be allowed without prompt since it's read-only, but anything that returns potentially sensitive info could arguably prompt. (The design could go either way; the user can always restrict by not connecting a DB they don't want accessed.) Also, because the DB is local, the data never leaves your machine – the server runs locally and returns results to Claude internally, maintaining privacy as long as your conversation remains local.

10. **Example Q&A:** Let's say your database is the famous `Chinook` sample (music store):

11. You ask: “Which artist has the most albums in the database?”
12. Claude (via MCP SQLite): it could do `SELECT Artist.Name, COUNT(*) as AlbumCount FROM Artist JOIN Album ON Artist.ArtistId = Album.ArtistId GROUP BY Artist.Name ORDER BY AlbumCount DESC LIMIT 1;`
13. SQLite server returns (“Iron Maiden”, 21 albums) for example.
14. Claude: “The artist with the most albums is Iron Maiden, with 21 albums.”

15. This is a natural language answer derived from a live database query – something incredibly useful for analytics folks or anyone with data.

16. **Multi-database:** If you set up multiple DBs (like `customer_db` and `product_db` as in the config example [15]), Claude could in theory use both, but it would treat them as separate tool sets. It might prefix tools by server name or have separate connections. Likely simpler is one at a time unless there’s a need to combine.

In essence, enabling a SQLite MCP server turns Claude into a conversational SQL analyst. Instead of writing SQL yourself, you ask questions, and it figures out the queries. This is similar to some products out there (e.g., ChatGPT Code Interpreter or other NL2SQL tools), but here it’s your own database and using an open standard interface.

Again, watch the logs if something is off. Common issues might be: - The path to DB is wrong (server might log “file not found”). - Node not found (same as before, ensure `npx` works). - If DB is locked by another process (SQLite is single-writer, but read-only should be fine). - If your DB requires write and you need that, you’d have to configure the server to allow write (maybe an option, but consider risk).

The SQLite server highlights the benefit of MCP: **complex capabilities (SQL querying)** are made available to AI in a standardized way. The AI didn’t need built-in SQL parser (though it helps that LLMs can do SQL); the server could even do some query planning. But the standard parts are listing tables (resources) and executing queries (tools). From the host perspective, whether it’s SQLite, Postgres, or MySQL via MCP – it’s similar. In fact, there are MCP servers for Postgres and others too, all working on the same client principle.

With both Filesystem and SQLite servers configured, Claude on your desktop becomes quite the personal assistant: it can fetch your files and cross-reference them with your database, all in one conversation, under your supervision.

Preparing for Future Lab Exercises (Prerequisites)

If you’re following along in a hands-on manner, or perhaps attending a workshop/lab on MCP, it’s important to get your environment ready. Here are some **prerequisites and setup tips** to ensure you can experiment with MCP integrations smoothly:

1. **Install Claude Desktop:** If you plan to use Anthropic’s Claude for local labs, download the Claude Desktop app from Anthropic’s site. It’s available for Windows and Mac. Install it and sign in (you’ll need a Claude account/API access, especially if using Claude for Work or an API key if that’s supported for Desktop). Verify it’s updated to the latest version (there’s a menu option “Check for Updates…”).

2. **Basic Developer Tools:** Many MCP servers (especially the reference ones) are written in Node.js or Python. So you should have:

3. **Node.js (and npm/npx)** installed (prefer LTS version).
4. **Python 3.10+** installed if you plan to build or run Python-based servers. Also pip.
5. Optionally, **Git** if you plan to clone repos, and possibly **Visual Studio Code** or another editor if you want to inspect or modify server code.

6. If on Windows, you might need Build Tools for Node or Python if compiling something, but most MCP packages are pre-built.

7. **The `uv` tool (optional):** Anthropic often uses a tool called `uv` (by Astral) for environment and package management in their examples. It's not mandatory, but can simplify running MCP servers in isolated environments. `uv` is like a wrapper around virtualenvs and package installation. You can install it via the one-liner in their docs (curl script). If that's too much, you can manage with npm and pip manually. The labs may instruct using uv to ensure consistent environments.

8. **API Keys and Accounts:** Depending on which integrations you'll explore:

9. **OpenAI API key:** If you'll use OpenAI's models (like GPT-4 or GPT-3.5 via LangChain or directly), get your API key ready.
10. **Azure OpenAI access:** If testing Azure's route, have an Azure account with OpenAI service deployed (and the endpoint & key).
11. **Anthropic API key:** If using Claude via API (though with Desktop, your login covers it if you have a subscription).
12. **Google Cloud account:** If you plan to use Google's Vertex AI (for Gemini model when available) or the Google ADK, you might need project credentials or API keys. For ADK specifically, there might be a separate setup (some of Google's agent stuff might require enabling services in Cloud console).
13. **PayPal developer sandbox:** For the PayPal MCP agent demo, having a PayPal developer account with sandbox credentials (Client ID, Secret) is needed. If a lab covers this, they'll likely provide instructions to get those from developer.paypal.com (which gives you a sandbox account and API keys).

14. **Other service keys:** e.g., if connecting to a Google Calendar MCP server, you'd need Google API credentials; if a Notion MCP server, a Notion integration token; etc. It depends on lab scenarios, but be prepared to obtain and supply API credentials for external services if needed.

15. **LangChain and LlamaIndex setup:** If labs involve these:

16. Install **LangChain** (pip install `langchain` or follow their docs).
17. Install **LlamaIndex** (a.k.a GPT Index, pip install `llama-index`).
18. For LangChain MCP adapter, you might need to install an extra like `langchain-mcp-tools` or similar. Check LangChain's documentation on MCP integration.
19. For LlamaIndex, ensure you have `llama-index-tools-mcp` or the relevant package. The LlamaHub or LlamaIndex docs have a section on MCP.

20. Ensure these can access the models you intend (OpenAI or others). That may require setting environment variables like `OPENAI_API_KEY` or Azure info, etc.

21. **MCP SDKs (if building custom):** If one of the exercises is to *build your own MCP server*, you should install the SDK in your chosen language:

22. Python: `pip install mcp` (or the specific `mcp-server` and `mcp-client` packages).
23. Node: the packages usually are `@modelcontextprotocol/server` etc., but often using npx is enough for reference servers.

24. Other languages (Java, C#, Go, etc.) have SDKs too – install via Maven/NuGet/etc. if needed. However, workshops might focus on Python or Node for simplicity.

25. **Test a basic connection:** A quick pre-lab sanity test could be using the MCP Inspector or a simple server:

26. Run `npx -y @modelcontextprotocol/server-echo` (if one exists, just guessing) or an example server they provide, and use the MCP Inspector tool (Anthropic has one in the Developer Tools on the MCP site) to see if you can connect and send a message. The MCP Inspector is essentially a UI to simulate a client [17].

27. Or use the `mcp-cli` if installed (`uv run mcp dev server.py` as in one example) to ensure messages flow.

28. **Understanding consent flows:** For remote connectors in Claude Desktop, be ready to handle OAuth flows. For example, connecting a Notion MCP server might pop open a browser for you to authorize access. This is normal. The lab might include those steps. Just don't panic when Claude Desktop opens a web login – it's getting a token to store and use with the connector.

29. **Update everything:** Make sure your Claude Desktop app is up to date, Node is recent, pip packages are updated to latest MCP versions, etc. The MCP spec and SDK have versioning but aim for the latest minor release to have all features.

By preparing these prerequisites, you'll avoid spending workshop time on environment setup issues and can dive straight into experimenting with MCP. Essentially: - **Have the tooling** (Claude Desktop, programming runtimes). - **Have access** (keys/accounts for relevant APIs). - **Have knowledge** of where config files and logs reside, so you can tweak and debug quickly.

Organizers often provide a list like the above before a hands-on MCP session, because a lot of moving pieces can be involved (LLM access, local installations, etc.). The good news is once it's set up, using MCP-enabled systems becomes very fluid and you can do amazing integrations in minutes.

Lab Exercise Idea 1: LangChain + Gemini + SQLite via MCP

Now we'll look at some scenarios combining everything we've learned. The first is using **LangChain** (a popular framework for building AI agent pipelines) with Google's **Gemini** model, and integrating a **SQLite** database via MCP. This shows how MCP enables *even different AI platforms (Gemini) to use the same tools*.

Context: Google's **Gemini** is an advanced LLM from Google DeepMind, slated to be integrated in Google Cloud's Vertex AI. Let's assume we have access to Gemini through Vertex AI or the Google ADK (Agent Development Kit). Meanwhile, we have a local (or cloud) SQLite database with data we want our agent to query.

Goal: Build an agent that uses the Gemini model to answer questions that require data from the SQLite database, using MCP to interface with the DB.

Steps:

1. **Set up the SQLite MCP server:** (We did this above for Claude Desktop, but here we might run it standalone since LangChain can connect independently.)
2. Suppose the SQLite DB is `chinook.db` (music store data). We launch the server manually: `npx -y @modelcontextprotocol/server-sqlite /path/to/chinook.db`.
3. This by default will start an HTTP + SSE server (since it's not Claude Desktop launching with stdio, many servers default to SSE mode if run standalone). It might print something like "Server running at http://localhost:4000" or similar in console. Now we have an MCP server listening.

4. Alternatively, LangChain's MCP tools adapter might allow starting it internally, but let's assume it expects a running endpoint.

5. **LangChain MCP integration:** LangChain recently added support for MCP. Specifically, there's a `langchain-mcp-adapters` library that makes MCP servers accessible as LangChain tools. We install that (`pip install langchain-mcp-tools` or as guided).

6. **Initialize the LangChain agent with Gemini:**

7. We authenticate to Vertex AI (maybe via Google Cloud SDK or set env vars).
8. In LangChain, we instantiate an LLM for Gemini, e.g.:

```
from langchain.llms import VertexAI
llm = VertexAI(model_name="gemini-benchmark", temperature=0)  #
hypothetical usage
```

9. Or if using ADK, that might involve ADK's own agent runtime. But assume we can use Gemini through a normal LLM interface in LangChain.

10. **Connect MCP as a tool:**

11. Using the MCP adapter, we create a tool that represents all the SQLite server's capabilities. For instance:

```
from langchain_mcp import MCPTool
mcp_tool = MCPTool(server_url="http://localhost:4000")  # URL where the
SQLite MCP server is running
tools = mcp_tool.get_tools()  # this might fetch the list of actual
tools (like "query", "schema")
```

    Under the hood, this tool likely:
    - Connects via SSE to `localhost:4000/sse`,
    - Calls `tools/list` to retrieve all endpoints (maybe it finds a `query` tool and some resource for schema).
    - Wraps each as a LangChain Tool object. We might end up with a list like `[Tool(name="query", func=...), Tool(name="list_tables", func=...)]`.

12. We include these in the agent's tool list. If LangChain's agent is one of the standard ones (like an OpenAI Functions agent or a ReAct agent), it can now call these tools.

13. **Configure the agent:**

14. We choose an agent type that can use tools. For instance:

```
from langchain.agents import initialize_agent, AgentType
agent = initialize_agent(tools, llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
```

    If the MCP adapter provided tool descriptions, the agent's prompt will contain those, so Gemini knows how to invoke them.

15. **Ask questions through the agent:**

16. Now we use the agent to answer questions that require the DB. E.g.:

```
response = agent.run("Which customer spent the most on music
purchases?")
print(response)
```

17. What happens: The agent prompt includes something like:

```
Tools:
query: Executes an SQL query on the connected SQLite database. Input: a
valid SQL SELECT query.
list_tables: Lists tables in the database.
...
```

    The question is asked. The Gemini LLM (with ReAct style) might think:
    - "I need to find the customer who spent the most. I likely need to sum invoices by customer."
    - It might first call `list_tables` to see table names (if not already known).
    - The adapter sends that MCP request, gets table names like Customers, Invoices, InvoiceLines, etc.
    - Gemini sees the result.
    - Then it forms a plan: likely calls `query` with `SELECT CustomerId, SUM(Total) as TotalSpent FROM Invoices GROUP BY CustomerId ORDER BY TotalSpent DESC LIMIT 1;`.
    - The MCP server executes the SQL, returns e.g. CustomerId 45, TotalSpent $100.
    - The adapter returns that to the agent.
    - Agent (Gemini) now has the raw result. It might cross-reference CustomerId 45 to name (maybe another query or maybe it knows to join with Customers table).
    - Possibly it does another `query`: `SELECT FirstName, LastName FROM Customers WHERE CustomerId=45;`.
    - Gets "John Doe".

    - Finally, it formulates an answer: *"The customer who spent the most is John Doe, who spent $100 in total."*

18. LangChain's agent mechanism will output that answer. We print it.

19. **Observe**: We should see the intermediate steps if verbose: tool calls and their outputs, which is great for learning. This shows that *Gemini (the LLM) was orchestrated via LangChain to use an MCP-provided tool* (SQLite query).

This lab demonstrates a few things: - **Interoperability:** We used a Google model (Gemini) with an Anthropic-born protocol (MCP) to access data. Open standards allow this mixing. - **LangChain synergy:** Instead of LangChain's built-in SQLDatabase chain (which exists), we tried using MCP. Why? Perhaps our scenario is that the database isn't directly accessible (maybe it's remote and LangChain can't connect, but an MCP server could be running closer to it or with caching). Or maybe to standardize across different DB types with one approach. - **Ease of adding tools:** With the MCPTool adapter, adding potentially *hundreds of tools* is easy. If the server had multiple tools (imagine connecting to a Zapier MCP with 50 actions), the adapter could ingest all as LangChain tools in one go. This is much faster than manually writing each tool wrapper. - **Future of Agents:** This hints that future agent frameworks might not hardcode tool logic but rather pull from MCP registries. In fact, LangChain's LangSmith now allows each agent to expose an MCP endpoint, meaning you could chain agents by MCP or expose your agent to others.

If we specifically mention **Gemini**: According to TechCrunch, Google plans MCP support with Gemini. This likely means that Google's **Agent** offering or ADK will natively speak MCP for tools. So an alternative approach could be using Google's ADK (Agent Development Kit) to build this agent: - ADK provides an agent orchestration (like LangChain, but Google-flavored). - ADK can integrate external tools via protocols like A2A and MCP. - We could register the SQLite MCP server as a tool in ADK, and the agent (using Gemini model) would similarily do NL→SQL via that.

The LangChain route is more vendor-neutral, while ADK would be Google's official path. Both aim to utilize MCP to avoid reinventing integration logic.

Conclusion of this exercise: We successfully combined a cutting-edge model with a live database using MCP, showcasing how questions in natural language can trigger structured queries behind the scenes. The user (or developer) didn't have to write any SQL or custom integration code for SQLite – the MCP server and standard tools did all the heavy lifting, with the LLM bridging the gap from NL to formal query. This pattern can be repeated for other combinations (e.g., a HuggingFace open model using MCP, etc.).

Lab Exercise Idea 2: LlamaIndex + Azure OpenAI + Filesystem MCP

Next, let's consider using **LlamaIndex** (another framework for connecting LLMs with external data, especially via indices) with **Azure OpenAI** (Microsoft's hosted version of OpenAI models like GPT-4), and integrate a **Filesystem** MCP server for document retrieval.

Scenario: Suppose you have a large set of text files (or PDFs converted to text) on your filesystem containing knowledge – perhaps company reports or academic papers. You want to build a Q&A system where the AI (GPT-4 via Azure) can answer questions using those files as references. You could load all text into memory or a vector index, but maybe you want to demonstrate on-demand retrieval with MCP.

Plan: Use LlamaIndex (which is good at creating query engines) and instead of its usual "download all data" approach, let it fetch data via an MCP Filesystem tool when needed. Azure OpenAI's GPT-4 will be the brain doing the heavy reasoning.

Steps:

1. **Set up Filesystem MCP Server:** As before, ensure the Filesystem server is running and has access to the directories with your documents. For a lab, maybe we restrict to a specific folder with sample docs. We can run:

   ```
   npx -y @modelcontextprotocol/server-filesystem "/path/to/docs"
   ```

   This likely starts on a default port (maybe 4001 or 7000). Let's say it's on `http://localhost:7000` for example. Alternatively, if local, one could integrate via STDIO. But since LlamaIndex likely expects a URL, we do HTTP.

2. **LlamaIndex MCP integration:** LlamaIndex has support for MCP tools via a `ToolSpec` class for MCP. We'll use that.

3. Install llama-index's MCP plugin (`pip install llama-index-tools-mcp` if needed).

4. Use their `BasicMCPClient` to connect:

   ```
   from llama_index.tools import MCPToolSpec, BasicMCPClient
   client = BasicMCPClient("http://127.0.0.1:7000/sse")  # connecting to
   filesystem server SSE
   mcp_tool_spec = MCPToolSpec(client=client)
   ```

   This will connect to the Filesystem server. Under the hood, it does similar to LangChain: `tools/list` to get available actions. The filesystem server might offer:

   - `list_dir`
   - `read_file`
   - `search_text` (if implemented)
   - etc. `MCPToolSpec` likely encapsulates all those. We might filter to only expose certain ones (some examples show you can pass an `allowed_tools` list). Let's say we allow `read_file` and maybe `search` if available.

5. Now, LlamaIndex can use this `mcp_tool_spec` in its toolkit. LlamaIndex's typical use-case is building an index, but it also supports a **Tool-augmented query engine**. Possibly we create a `QueryEngine` that knows how to use tools when needed. For example, LlamaIndex has a `FunctionAgent` or similar that can call tools (the medium example [28] shows constructing a `FunctionAgent` with the tools from `MCPToolSpec`).

   In our scenario, we might not even need a vector index at first. We could rely purely on search. But to illustrate hybrid usage: - We could combine a vector index for quick lookup of which files might be relevant, then use MCP to actually read those files fully. However, to keep it

straightforward, let's assume we'll use an MCP `search` tool to find relevant files, and then `read_file` to get content, similar to how an agent would do it step by step.

6. **Configure Azure OpenAI GPT-4:** We set up LlamaIndex to use Azure OpenAI:

```
from llama_index import ServiceContext, LLMPredictor
import os
os.environ["OPENAI_API_TYPE"] = "azure"
os.environ["OPENAI_API_BASE"] = "<your Azure endpoint>"
os.environ["OPENAI_API_KEY"] = "<your key>"
os.environ["OPENAI_API_VERSION"] = "2023-05-15"  # or relevant
llm_predictor = LLMPredictor(llm=OpenAI(model="gpt-4",
deployment_name="GPT4DeployName", ...)
service_context =
ServiceContext.from_defaults(llm_predictor=llm_predictor)
```

   Now LlamaIndex can use GPT-4 via Azure to generate.

7. **Create an agent or query engine with tools:** Using the MCPToolSpec from earlier, we can do:

```
tools = mcp_tool_spec.to_tool_list()  # maybe get as actual callable
tools
# Suppose tools now include methods: read_file and list_dir etc.
from llama_index.agent import OpenAIAgent  # hypothetical agent
agent = OpenAIAgent.from_tools(tools, llm_predictor=llm_predictor)
```

   If LlamaIndex doesn't have a single-call Agent, we might need to manually do the reasoning: But they showed an example where they define `handle_user_message` that streams through agent events. In a lab, a simpler approach might be to just demonstrate tool usage.

Actually, LlamaIndex often constructs a **Graph** of tools or **Router**. There might be a way to wrap the MCP server as a "Node" in LlamaIndex such that when a question is asked, it routes to a tool usage.

Another approach: LlamaIndex could simply call the MCP as needed inside a query function: - For example, override the `QueryEngine.query` method to first call the search tool of MCP to get relevant file paths, then call read on those files, then feed into GPT for answer. But that would be doing manually what an agent would automate.

Since this is a lab exercise, an **interactive demonstration** might be done: - Prompt GPT-4 something like: "You have access to a filesystem tool. Use it to answer questions based on documents in the directory." And have the conversation with the model with you manually approving tool usage. But to keep it within code, better to automate.

1. **Ask a question using the agent:** e.g.:

```
response = agent.query("Summarize the key findings from all reports
about Project X in the docs folder.")
```

The agent (GPT-4 behind the scenes) sees the tools. It might:

2. Use a search tool: like `search("Project X")` on the folder.
    - The filesystem MCP server returns a list of files and maybe snippets where "Project X" appears.
3. GPT-4 decides which files look relevant from their names or snippet.
4. For each relevant file (maybe it picks top 2-3), it calls `read_file(file_path)`.
5. Each time, we (or our code) will see the requests. If not auto-approved, we might need to configure the server to auto-approve or simulate it in code. Since we don't have the Claude Desktop UI here, and our code is directly hitting the server, by default it might not have an approval step (the server might be configured in open mode because it's you running it; the approval UI is a function of Claude Desktop host, not the server itself – the server trusts the client in this case).
6. The content comes back. GPT-4 reads the content (which LlamaIndex passes into the prompt). If the content is large, LlamaIndex might chunk it or GPT-4 may summarize chunk by chunk.
7. Finally, GPT-4 produces a consolidated summary of "Project X" from those reports.

The agent returns that as `response`.

1. **Output & verification:** We examine the response, maybe it correctly summarizes. We could print intermediate steps if we logged them.

This exercise highlights: - Using **Azure's AI with MCP**: We didn't need OpenAI's plugin system or anything proprietary; even running through Azure, we can use MCP connectors. - **LlamaIndex's flexibility**: It allowed pulling in an MCP tool to get data on the fly, rather than requiring a pre-ingestion of all text. In real scenarios, LlamaIndex often builds an index, but MCP could allow a hybrid where if something isn't indexed, it can still retrieve it. Or one could imagine a "FilesystemQueryEngine" that uses GPT to decide to open files – basically what we did. - The benefit of **structured tool use** with a strong model: GPT-4 is very capable of deciding how to use tools if given proper instructions. It likely does this well, making it a powerful document assistant.

For completeness, note that in a retrieval context one might worry about hallucination or accuracy. If we wanted to ensure quotes from documents, we might retrieve relevant parts and feed them into GPT as context. MCP can serve as a way to fetch those relevant parts (like a poor-man's RAG system). LlamaIndex could also have built an index of vector embeddings of file contents to first identify relevant docs by semantic similarity, then use MCP to pull the actual text of those docs. That would combine best of both worlds: efficient search by embedding, and precise retrieval via MCP. That could be an advanced lab exercise.

Lab Exercise Idea 3: OpenAI SDK & Azure Services via MCP Server

For another angle, consider using OpenAI's own developer tools (which now support MCP after March 2025 [1]) to integrate with an Azure service. Perhaps an example: **Azure Cognitive Search MCP server** or an **Azure Blob Storage MCP server**.

This hypothetical exercise could be: - Use OpenAI's Python SDK (or the new Agents SDK if released) to connect to an Azure-provided MCP server that gives access to some Azure resource. - For instance, Microsoft might have an "Azure MCP Server" that allows natural language queries to Azure Resource Graph or controlling Azure services (the search results [43] show something like that: *Azure MCP Server enables AI agents to interact with Azure resources through NL*). - Or consider a simpler one: A **CData Connect** MCP server that interfaces with an Azure SQL database or Power BI (CData has something like that which was referenced).

However, to make it concrete, we saw: **Azure MCP Server** docs hint that Microsoft created an MCP interface for Azure. Possibly you can ask “Create a virtual machine” and it goes through MCP to Azure CLI under the hood, etc.

A hands-on lab might: 1. Deploy or enable the Azure MCP Server in your Azure environment (there might be a way to spin it up via Azure CLI or a container). 2. Use OpenAI’s Agent to connect to it. Since OpenAI’s ChatGPT (desktop or web) now supports custom MCP connectors, one could even use ChatGPT to connect to Azure’s MCP if allowed. But focusing on code: the OpenAI SDK might have something like:

```
import openai
conn = openai.MCPConnection(server_url="https://azuremcp.mycloud.com/...",
auth_token="...")
# (This is speculative API)
```

Or the OpenAI Agents SDK might allow registering a tool via an MCP descriptor.

If none of that is public, we can simulate by using OpenAI function-calling to call an MCP client code.

1. Example query: “List all storage accounts in my Azure subscription.” The AI model (OpenAI GPT-4 via Azure perhaps, or just OpenAI GPT-4 if doing directly) would have a function (tool) corresponding to an MCP method for listing storage accounts. The MCP server on Azure side would be connected to Azure Resource Manager and can execute that query, returning data. The AI returns a user-friendly listing.

Given not enough concrete public info on Azure’s MCP server and OpenAI’s integration in code, the lab might pivot to using an **Azure data source** through MCP: - e.g., hooking up an **Azure Cosmos DB** via an MCP server (CData perhaps offers connectors, or Azure may have one). - Or using Azure OpenAI’s **Plugins** support indirectly. But they asked specifically *OpenAI SDK & Azure MCP Server*, which suggests using OpenAI’s dev toolkit to hit an Azure MCP.

Perhaps a clearer example: **Use OpenAI’s API with an MCP server running on Azure to fetch data.** - For instance, run a *Knowledge Base MCP server* on Azure (like something that indexes a company’s SharePoint or so). - Then from a Python script, use OpenAI’s API with function calling: define a function that when called will call the MCP server (the Python code acts as glue). - If the question requires data, the model triggers the function, which our code implements by calling the MCP server’s HTTP endpoint, gets result, returns to model. - The model then responds with the answer.

This basically is implementing what OpenAI might natively do eventually, but as a developer you can already do this.

So in code:

```
import openai, requests, json

# Suppose we define a fake function for the AI:
functions = [
  {
    "name": "azure_search",
```

```
    "description": "Search Azure for resources by name",
    "parameters": {"type": "object", "properties": {"query": {"type":
"string"}}}
  }
]
# Chat with function support
resp = openai.ChatCompletion.create(
  model="gpt-4-0613",
  messages=[ {"role": "user", "content": "How many VMs are running in region
West Europe?"} ],
  functions=functions
)
message = resp['choices'][0]['message']
if message.get("function_call"):
    func_name = message["function_call"]["name"]
    args = json.loads(message["function_call"]["arguments"])
    if func_name == "azure_search":
        # Call the Azure MCP server with args['query']
        result = requests.post(AZURE_MCP_URL, json={"jsonrpc":"2.0","id":
1,"method":"tools/call","params":{...}}).json()
        # Format result...
        answer_data = ...
        # Return function result to model
        second_resp = openai.ChatCompletion.create(
           model="gpt-4-0613",
           messages=[
             {"role":"user", "content": user_query},
             {"role":"assistant", "content": None, "function_call":
message["function_call"]},
             {"role":"function", "name": func_name, "content":
json.dumps(answer_data)}
           ]
        )
        print(second_resp['choices'][0]['message']['content'])
```

This sketch shows using OpenAI's function calling to integrate an MCP call within an OpenAI chat. Essentially, we manually bridged the gap. But since the question suggests by now OpenAI's SDK might natively support MCP (maybe not yet, but let's assume maybe via a simpler interface), the goal is demonstrating *OpenAI model + MCP for Azure*.

Bottom line for lab: The participant sees that even if an AI model doesn't directly have data about something (like Azure resources), through MCP it can get live data and then answer, with minimal fuss.

Given space, perhaps we won't elaborate too much on this one. It's more showing how open protocols allow synergy between OpenAI and Microsoft ecosystems.

Lab Exercise Idea 4: Google ADK & PayPal MCP Server

This is an exciting demonstration of multi-agent collaboration using open protocols (MCP and also A2A). It combines **Google's ADK (Agent Development Kit)** with PayPal's **Agent Toolkit MCP server** to create

an “agentic commerce” scenario. Essentially, an AI sales assistant (built with ADK and perhaps powered by Gemini) can interact with PayPal’s agent (which handles payments) to complete a transaction, all via protocol messages.

Background: As mentioned, PayPal and Google Cloud showcased an “agentic commerce” demo in 2025. The PayPal agent exposes payment operations (like creating orders, checking out, handling subscriptions) through a combination of A2A and MCP (AP2 is a layer on top for trust with blockchain, but in practical tool terms, PayPal provided an MCP server with commerce APIs).

Lab Setup: 1. **PayPal MCP server / Agent Toolkit:** PayPal’s Agent Toolkit likely includes an MCP server. Possibly available via their developer platform. We might simulate with sandbox. That GitHub `pp_mcp_adk` by gkcng was exactly such a demo codebase. It listed PayPal API capabilities and used ADK.

For a lab, assume we have the PayPal MCP server running, configured with sandbox credentials (Client ID/Secret from PayPal dev). It might run at a URL or locally (maybe one launches it similarly: `npx @paypal/mcp-server` if available).

According to the README, it used certain APIs: orders.create, orders.capture, etc.. So the tools available likely are `orders_create`, `orders_capture`, `subscriptions_create`, etc., each corresponding to a PayPal API call.

1. **Google ADK environment:** ADK is a framework (perhaps Python library `google-adk`). The demo repo indicates they used `pip install google-adk==0.3.0` etc.. So we set that up in a venv.

ADK would allow defining agents and connecting them over A2A protocol. But here, perhaps we create a single agent that has access to the PayPal MCP tools. Alternatively, the full experience is two agents: - Merchant’s Shopping Agent (with product catalog, user conversation) - PayPal Payment Agent (handles payment, likely a separate service)

The blog said: Merchant agent talks to PayPal agent via A2A, and AP2 ensures secure transaction etc..

But in an MVP lab, one could just integrate the PayPal MCP directly into a single agent to show it performing a purchase.

1. **Agent conversation simulation:**
2. The user says to the merchant agent: “I’d like to buy a size M blue T-shirt.”
3. Merchant agent (with product DB perhaps or knowledge) finds the item and says: “Great, that’s $25. Shall I proceed to checkout with PayPal?”
4. User: “Yes.”
5. Now the agent needs to create an order, have the user pay.
6. It uses the PayPal MCP server: calls `orders.create` tool (which through sandbox API returns an approval URL or immediate approval if using test account).
7. Possibly it outputs a link for user to simulate payment (in a real scenario, the user would login to PayPal at that URL – in sandbox maybe we skip or assume auto-approval).
8. Then agent calls `orders.capture` to finalize (in sandbox, capturing a test order).
9. The PayPal MCP server returns success of payment.
10. The agent then confirms to user: “Payment successful! Your order is confirmed. Here’s your receipt…”

In a lab, since we can't actually integrate a user clicking a link easily, one might mock the payment step or auto-approve via sandbox (some sandbox flows can auto-approve if you use special testing parameters).

The focus is to illustrate the agent calling multiple MCP tools (create order, capture order, etc.) in response to conversation.

1. **Security aspects (AP2):** While a lab might not implement cryptographic credentials, it could be mentioned: AP2 mandates a "Cart Mandate" digitally signed by user etc., which PayPal agent and merchant agent exchange. But that's beyond coding scope, more concept.

So, practically, the lab code would: - Use ADK to create an agent that can call functions (tools). Possibly ADK has a similar structure to LangChain's where you register MCP tools. - Provide the conversation logic: maybe some if/then to simulate user going off to pay or not required if auto-captured.

Given complexity, perhaps the lab is more demonstrative, running a pre-built flow (like running that gkcng demo which may have interactive prompts).

Nonetheless, the outcome shows: - AI agents can perform real transactions via MCP-mediated tool use. - The user sees the agent handle the entire process in natural language, from selection to payment, by coordinating with PayPal's APIs through MCP. - It underscores the power of combining **Agent (LLM)** with **structured finance APIs** in a safe, standardized way. (One can imagine future e-commerce where you chat with a bot to buy things, which under the hood uses exactly these moves, with all the trust and verifications baked in via protocols like A2A/AP2).

For time, we can summarize as above, citing the relevant parts: The PayPal agent toolkit MCP included numerous commerce API endpoints, which the agent can call to complete tasks like create/capture order. Google ADK facilitates building the logic to use those in a conversation flow.

Building Your Own MCP Server – A Quick Guide

Finally, a lab or tutorial would often culminate in **building a custom MCP server** to reinforce understanding. Let's outline how to create a simple MCP server from scratch (e.g., in Python) and test it with a client.

Use case: Suppose we want an MCP server that provides a simple **Calculator** and a **Joke generator** (just as an arbitrary example). This server will have two tools: - `calculate(expression: str)` – evaluates a math expression and returns the result. - `tell_joke()` – returns a random joke from a small list.

And maybe a resource: - `pi` – a resource that always gives value of π (just for fun).

Steps to build (Python SDK example):

1. **Install MCP SDK for Python:** e.g. `pip install mcp`.

2. **Define the server with FastMCP:** Using the docs guidance:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("MyToolsServer")
```

This creates an MCP server instance with a given name.

3. **Define Tools:** The SDK likely allows a decorator or method to add tools. For instance:

```
@mcp.tool()
def calculate(expression: str) -> float:
    """Evaluates a mathematical expression and returns the result."""
    try:
        result = eval(expression, {"__builtins__":None}, {})
    except Exception as e:
        raise Exception(f"Invalid expression: {e}")
    return result

@mcp.tool()
def tell_joke() -> str:
    """Tells a random joke."""
    import random
    jokes = [
        "Why did the math book look sad? Because it had too many
problems.",
        "I told my AI it was average, it responded: 'That's mean.'"
    ]
    return random.choice(jokes)
```

The decorator `@mcp.tool()` likely registers these functions as MCP tools, inferring their name and docstring as description (the SDK often uses function name and type hints to auto-generate the JSON schema).

4. **Define a Resource (optional):** Perhaps adding a constant resource. If the SDK supports it, maybe something like:

```
mcp.add_resource("pi", "3.1415926535")
```

Or we could make a dynamic resource: Actually, the concept of resource in code might be handled by special methods like `@mcp.resource()` decorator. But for simplicity, skip or just return via a tool.

5. **Run the server:** At the end:

```
if __name__ == "__main__":
    mcp.run("stdio")  # or "sse"
```

For local testing, we use stdio.

6. **Test it with MCP Inspector or a client:** Launch this script: `python mytools_server.py`. It will print something like "Server running (stdio)".

7. If using **MCP Inspector** (a GUI tool Anthropic provides [17]), open it, it can connect to a running stdio server by launching it itself or attach.

8. Alternatively, write a small client snippet or use the Python SDK's client:

```
from mcp import Client
client = Client("stdio:./mytools_server.py")
# hypothetical connect via stdio by spawning
client.connect()
tools = client.list_tools()
print("Available tools:", [t.name for t in tools])
result = client.call_tool("calculate", {"expression": "2+2*2"})
print("Calculation result:", result)
```

   The client would show the list includes "calculate" and "tell_joke". The call to calculate should return 6. Similarly, `client.call_tool("tell_joke", {})` should return one of the jokes.

9. **Integrate with an AI model:** As a final step, we can show our new server working with an AI:

10. If Claude Desktop is installed, add it to config just like earlier sections, then ask Claude to use it. For example, add:

```
"mcpServers": {
    "mytools": {
        "type": "stdio",
        "command": "python",
        "args": ["path/to/mytools_server.py"]
    }
}
```

    Restart Claude Desktop. Claude now has "mytools" with a calculate and tell_joke tool. Ask: "What's 2+2*2?" – Claude should decide to use calculate and respond "6". Ask: "Tell me a math joke" – it may call tell_joke and reply with the joke string.
11. Or use OpenAI function calling like earlier to integrate it similarly.

This demonstrates how relatively easy it is to write an MCP server: a few lines to define functions and you get a standardized API for AI to use. You didn't have to worry about how the AI will parse input or format output – the MCP SDK will handle the JSON wrapping, type enforcement (via type hints), etc. It's a huge win for developer productivity.

Best practices noted: - Logging: We avoid `print()` in server code for STDIO mode because printing would conflict with JSON output. Instead we'd use logging to stderr if needed (the doc warned about that). - Tool naming conventions: e.g., recommended not to use underscores etc. (Perhaps `tools` are namespaced or require snake_case). The doc suggests following a format like kebab-case or something.

- Security: using `eval` unsafely is not recommended; we sanitized `eval` by no builtins. In real servers, be mindful of code execution, file access, etc., and validate inputs since the AI might send unexpected stuff.

Conclusion: Building your own MCP server is quite straightforward with the help of SDKs. You focus on the actual function logic – making sure it's something the AI might need – and the framework handles exposing it over JSON-RPC, advertising it to clients, etc. This modular approach means anyone can extend the AI ecosystem by writing a small server that connects to *anything* – hardware, web API, legacy system – and instantly any MCP-aware AI can use that new capability.

By completing such a lab, developers gain the skills to incorporate their custom tools and data into AI workflows in a robust and standardized way, rather than resorting to ad-hoc prompt engineering or custom integration for each use case. **MCP essentially allows you to *plugin-ize* anything you can code.**

Sources:

– Official MCP introduction (modelcontextprotocol.io) explaining MCP as a USB-C for AI, connecting to files, databases, tools.

– Wikipedia on MCP, describing it as open standard for AI-tools integration, noting reuse of LSP ideas and JSON-RPC 2.0 transport.

– Anthropic announcement of MCP (Nov 2024) highlighting isolation of models and need for universal protocol, and introduction of Claude Desktop support.

– Wikipedia details confirming JSON-RPC 2.0 usage and STDIO/HTTP+SSE as transports.

[1] – Wikipedia on adoption: OpenAI adopted MCP (Mar 2025) across ChatGPT app and Agents SDK; Google DeepMind confirming Gemini support (Apr 2025).

– Medium (Roberto Infante) describing MCP as USB-C for AI that avoids custom glue code, and listing early adopters like Block, Replit, etc.

– Medium explanation of STDIO vs HTTP+SSE for MCP, from a code integration perspective.

– ToolUniverse doc illustrating Claude Desktop connecting to an MCP server (with a diagram showing Claude Desktop and a server with tools, e.g., 600+ scientific tools).

– Claude support doc listing capabilities of the Filesystem server (file operations like read, create, move, search) and stressing user approval for each action.

– Claude support doc showing example user requests ("write a poem…" and "organize images…") and explaining that Claude will request user approval for file operations.

– MCP architecture doc explaining what tools and resources are, with examples (tools for actions, resources for data like file contents, etc.).

– MCP architecture explaining tool update notifications.

– Medium NocoBase article quoting debate on MCP and that it’s solving last-mile integration, implying consensus forming around it.

– PayPal ADK demo README listing which PayPal APIs (tools) were used in the demo (orders.create, capture, subscriptions).

---

1 7 Model Context Protocol - Wikipedia
https://en.wikipedia.org/wiki/Model_Context_Protocol

2 3 4 5 Architecture overview - Model Context Protocol
https://modelcontextprotocol.io/docs/learn/architecture

6 13 14 15 16 How to Step and Use SQLite MCP Server | by Erik Milošević | Towards AGI | Medium
https://medium.com/towards-agi/how-to-step-and-use-sqlite-mcp-server-c87ac20f913e

8 Connect to remote MCP Servers - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/connect-remote-servers

9 Accelerating LLM-Powered Apps with MCP and A2A Protocols | by Roberto Infante | Medium
https://medium.com/@roberto.g.infante/accelerating-llm-powered-apps-with-mcp-and-a2a-protocols-73d388fb4338

10 11 12 Connect to local MCP servers - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/connect-local-servers

17 Build an MCP server - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/build-server