Model Context Protocol: An In-Depth Look at Implementation and Usage

1. Introduction to Model Context Protocol (MCP)

The Model Context Protocol (MCP) represents a significant advancement in the way AI assistants interact with the vast ecosystem of data and tools that exist beyond their training datasets [1]. Developed by Anthropic, MCP is an open standard designed to establish seamless connections between AI clients and various external systems, including content repositories, business applications, and development environments [1]. The fundamental concept behind MCP is straightforward: instead of creating bespoke integrations for each data source an AI model needs to access, a universal protocol can be employed to manage the flow of contextual information between the AI and the relevant systems [1]. This approach can be likened to a "USB-C port for AI," offering a standardized interface for connecting AI systems to a multitude of tools and data sources, thereby simplifying integration and fostering interoperability [3]. While initially conceived by Anthropic, MCP is an open-source project, inviting contributions from the wider community to enhance and expand its capabilities [7].

The adoption of MCP offers several compelling benefits over traditional API integrations [1]. Firstly, it simplifies development by providing a single, standardized protocol that AI assistants can utilize to query and retrieve data from diverse sources, reducing the need for developers to write custom code for each specific integration [1]. This unified approach also leads to more consistent security and control across different tools, replacing ad hoc API connectors with a protocol that handles authentication and standardized data formats [1]. Furthermore, MCP promotes sustainability by encouraging the creation of reusable connectors, known as "servers," which can be built once and then leveraged across multiple Large Language Models (LLMs) and AI clients, eliminating the need to rewrite the same integration repeatedly [1]. Ultimately, MCP aims to create a more uniform and standardized ecosystem for AI integration, making it easier to provide LLMs with the up-to-date, context-rich, and domain-specific information they require to generate more relevant and accurate responses [1].

The architecture of MCP comprises several key components that work together to facilitate this seamless integration [2]. **MCP Hosts** are the LLM-based applications, such as Claude Desktop or AI-powered Integrated Development Environments (IDEs), that require access to external data or tools to enhance their functionality [1]. **MCP Clients** are components integrated within these host applications that manage the dedicated, one-to-one connections with specific MCP servers [1]. These clients handle the intricacies of message exchange with the servers, routing requests and responses according to the MCP protocol. **MCP Servers** are lightweight, standalone programs or services that expose specific data or functionalities to AI applications through the MCP interface [1]. A server might provide access to a local file system, a remote API, or a database. Finally, the **Transport Layer** defines the underlying communication mechanism between these clients and servers, ensuring reliable message delivery [5].

The central advantage of MCP lies in its ability to simplify the complex task of connecting diverse data sources and tools with LLMs by offering a standardized and more manageable framework. This addresses the inherent challenge of needing unique integration solutions for every combination of LLM and data source, often referred to as the "NxM problem" [10]. By establishing a common protocol, MCP paves the way for a more interconnected and efficient AI ecosystem.

2. In-Depth Exploration of MCP Transports

The Model Context Protocol supports several transport mechanisms that facilitate communication between MCP clients and servers [5]. Two primary transport types are defined within the protocol: Standard Input/Output (stdio) and HTTP with Server-Sent Events (SSE).

Standard Input/Output (stdio) is a transport that enables communication through the standard input and output streams of a process [5]. In this model, the MCP client typically launches the MCP server as a separate subprocess [13]. The server then receives JSON-RPC messages from the client via its standard input (stdin) and sends responses back to the client through its standard output (stdout) [13]. The messages exchanged are formatted according to the JSON-RPC 2.0 specification and are delimited by newline characters, with no embedded newlines permitted within the messages [12]. For logging purposes, the server may also write UTF-8 strings to its standard error (stderr), which the client can choose to capture or ignore [13].

Implementing stdio transport in Python using the MCP SDK is relatively straightforward. A basic server can be created using the FastMCP class, which provides an interface for defining tools and resources [5]. For instance, a simple tool to add two numbers can be defined using the @mcp.tool() decorator within a FastMCP instance [5]. Similarly, resources can be exposed using the @mcp.resource() decorator [16]. When this server script is executed, it will listen for client connections via stdio. On the client side, interaction with such a server can be done programmatically using the SDK or through tools like the MCP Inspector, which can simulate a client and send requests to the server over stdio [5].

The stdio transport offers several advantages [5]. It is particularly well-suited for local integrations where the client and server run on the same machine, such as command-line tools or desktop applications needing to access local resources [5]. The communication is generally efficient due to the minimal overhead of inter-process communication compared to network-based transports [11]. Furthermore, it simplifies process management as the client typically manages the lifecycle of the server subprocess [11]. For local tools, this method also avoids the complexities and potential latencies associated with network communication [5].

However, stdio transport also has certain limitations [8]. Its primary drawback is that it is designed for local communication and is not ideal for scenarios where the MCP server needs to run remotely from the client [5]. In sandboxed desktop applications, executing external binaries as subprocesses might be restricted, posing a challenge for stdio-based servers [19]. Additionally, using stdio for remote development over SSH can sometimes lead to issues [20]. The lack of a standardized authentication mechanism in the base protocol can also be a concern, especially in environments where security is paramount [21]. Configuration errors can also present hurdles, as the local nature of the server might limit data accessibility in certain setups [8]. Finally, the design of stdio might not be optimal for serverless deployments that often involve short-lived processes [18].

HTTP with Server-Sent Events (SSE) provides a transport mechanism designed for remote communication between MCP clients and servers [5]. In this approach, the MCP server operates as an independent process capable of handling connections from multiple clients [13]. The server must expose two HTTP endpoints: an SSE endpoint for transmitting server-to-client messages and a standard HTTP POST endpoint for receiving client-to-server messages [12]. When a client initiates a connection to the server's SSE endpoint, the server typically sends an initial event containing a unique URI that the client should use for sending subsequent messages back to the server [13]. All client-originated messages are then sent as HTTP POST requests to this URI [12]. Conversely, the server sends messages to the client as SSE message events, with the content of these messages encoded in JSON format within the event data [12].

Implementing SSE transport in Python often involves using a web framework like Flask or FastAPI to create the necessary HTTP endpoints. While the MCP Python SDK might offer utilities for simplifying SSE integration, the fundamental principle involves setting up a route that handles incoming GET requests for the SSE stream and another route for POST requests. For the SSE endpoint, the server needs to maintain a persistent connection with the client and send events as they occur. On the client side, libraries that support SSE, combined with standard HTTP libraries like requests for sending POST requests, can be used to interact with the MCP server [23].

The advantages of using SSE transport include its suitability for remote communication, allowing MCP servers to run independently of the client's local environment [5]. It inherently supports server-to-client streaming, which can be beneficial for applications that need to receive continuous updates from the server [5]. SSE can also be advantageous in certain network environments where long-lived HTTP connections are more readily permitted than other protocols [13]. For scenarios involving simple, unidirectional updates from the server to the client, SSE provides a relatively straightforward solution [13].

Despite its benefits, SSE transport also has several limitations [14]. The primary limitation is that it is fundamentally a unidirectional communication channel (server to client); client-to-server communication requires separate HTTP POST requests, which can introduce additional overhead [12]. SSE is also limited to transmitting text-based data formats, primarily JSON, which might not be optimal for applications needing to handle binary data [24]. Maintaining persistent connections for each client can lead to scalability challenges on the server side, especially with a large number of concurrent users [24]. Older browsers, such as Internet Explorer, may not natively support SSE, requiring fallback mechanisms [24]. Handling reconnections after a dropped connection can be complex and might result in duplicate messages if not managed carefully [24]. Furthermore, support for SSE in some development and testing tools might be limited [24]. Browsers also impose limits on the number of concurrent open connections, which can be a concern when using SSE, particularly over HTTP/1.1 [25]. There is also a noted lack of native support for authorization headers in the SSE specification [25]. Finally, the long-lived nature of SSE connections might not be ideal for serverless environments with short-lived request lifecycles [18].

Beyond the standard stdio and SSE transports, the Model Context Protocol is designed to be extensible, allowing for the implementation of **custom transport mechanisms** to suit specific needs [13]. The protocol itself is transport-agnostic, meaning it can be implemented over virtually any communication channel that supports bidirectional message exchange [13]. Use cases for creating custom transports might include the need for integration with custom network protocols, specialized communication channels, existing legacy systems, or for performance optimization in particular scenarios [14]. To implement a custom transport, developers need to ensure that their implementation conforms to the Transport interface defined by the MCP specification and that it correctly handles the serialization and deserialization of JSON-RPC messages [14]. For interoperability, it is crucial that any custom transport implementation is well-documented, outlining its specific connection establishment and message exchange patterns [13].

In terms of **performance**, the choice of transport can have implications [1]. Generally, for local communication, stdio tends to be faster due to its minimal overhead [5]. SSE, while suitable for remote scenarios, introduces network latency, but its server-push capability can be efficient for real-time updates [12]. However, the actual performance in any given application will depend on a variety of factors, including network conditions, server load, and the size and frequency of messages being exchanged.

To summarize the key differences between the two primary transport types, the following table provides a comparison:

Feature	**stdio**	**SSE**
Use Cases	Local integrations, command-line tools, simple process communication	Remote communication, server-to-client streaming, simple updates
Communication Direction	Bidirectional (via stdin/stdout)	Primarily unidirectional (server to client), client to server via HTTP POST
Data Format	JSON-RPC (text-based)	JSON (text-based)
Local/Remote	Primarily local	Primarily remote
Performance (General)	Faster for local communication	Introduces network latency, efficient for server-push
Advantages	Simple for local setup, efficient for same-machine communication, avoids network overhead for local tools	Suitable for remote servers, enables server-to-client streaming, works with restricted networks

Disadvantages	Not well-suited for remote servers, potential sandbox restrictions, issues with remote development, lacks standardized authentication	Primarily unidirectional, limited to text data, potential scalability issues, browser compatibility concerns, reconnection complexity

The selection of the appropriate transport mechanism is a critical decision in the design and implementation of MCP-based applications. It hinges on the specific requirements of the use case, particularly the proximity of the client and server and the desired communication pattern. While stdio offers simplicity and speed for local interactions, SSE provides a standardized approach for remote communication with server-initiated data flow. Custom transports offer the greatest flexibility but come with the added responsibility of implementation and documentation.

3. Implementing Communication with MCP

Establishing communication between an MCP client and server involves a well-defined lifecycle, starting with connection establishment and protocol negotiation [11]. This process ensures that both the client and server can understand and interact with each other effectively.

The **initialization phase** is crucial for setting up the communication session. When an MCP client starts a connection with a server, it first sends an initialize request. This request typically includes information about the client's supported protocol version and its capabilities, such as the types of messages it can handle and the features it supports [11]. The server then responds to this request, indicating its own supported protocol version and capabilities, which might include the tools, resources, and prompts it offers [11]. Once the client receives the server's response, it sends an initialized notification back to the server as an acknowledgment that the initial handshake is complete [11]. After this initial negotiation, the connection is considered established, and the normal message exchange can begin [11]. During this phase, clients can send requests to the server to invoke tools or access resources, and the server will respond with the results or errors. Both parties can also send notifications for one-way communication that does not require a response [11]. Finally, the communication session can be terminated by either the client or the server, depending on the application's needs [11].

All messages exchanged between MCP clients and servers are formatted according to the **JSON-RPC 2.0 specification** [5]. This standard defines the structure of different types of messages used in the protocol [11]. A **Request** message contains a method, which is a string identifying the procedure to be called, and an optional params field, which can be either an object or an array containing the parameters for the method [11]. Request messages expect a corresponding **Result** message in response, which contains the data returned by the called method [11]. If an error occurs during the processing of a request, the server sends back an **Error** message. This message includes a code (a number indicating the type of error), a message (a short description of the error), and an optional data field that may contain additional information about the error [11]. Lastly, a **Notification** message is used for one-way communication from either the client or the server that does not require a response. It has a similar structure to a Request, with a method and optional params, but no Result is expected in return [11].

The Python SDK provides abstractions that simplify the implementation of this communication flow for both stdio and SSE transports [16]. On the **server-side** when using stdio, a server instance can be created with FastMCP. Request handlers for specific tools and resources can be defined using decorators like @mcp.tool() and @mcp.resource() [16]. The FastMCP server automatically handles the underlying protocol details, listening for incoming JSON-RPC requests over the standard input stream and sending responses back via the standard output stream. For example, a function decorated with @mcp.tool() will be registered as an available tool, and when the server receives a tools/call request with the name of this tool, the decorated function will be executed, and its return value will be sent back as a JSON-RPC result.

On the **client-side** with stdio, a client can be implemented to connect to the server. The client would typically send JSON-RPC requests to the server's standard input and read the responses from its standard output. While the user's query did not specifically ask for client-side implementation details, tools like the MCP Inspector can be used to interact with an MCP server over stdio, sending requests and viewing the responses, which helps in understanding the communication flow [5].

For **SSE transport** on the server-side in Python, one might use a web framework like Flask along with the MCP SDK (if it provides specific SSE support) or standard Python libraries. The server would need to set up an HTTP route that serves as the SSE endpoint. When a client connects to this endpoint, the server would initiate a persistent connection and send SSE events containing JSON-formatted MCP messages. Another HTTP route would be needed to handle POST requests from the client, which would be used to send commands or data to the server [14].

On the **client-side** with SSE, a client would first establish a connection to the server's SSE endpoint, typically by sending an HTTP GET request. Once the connection is established, the client would listen for incoming SSE events from the server. To send messages to the server, the client would make HTTP POST requests to the specific URI provided by the server during the initial connection phase. Libraries like the requests library in Python can be used for making these POST requests [23].

The reliance on the widely adopted JSON-RPC protocol for MCP communication ensures a high degree of interoperability and makes it easier for developers familiar with web service architectures to understand and implement MCP clients and servers. The initial handshake involving the exchange of protocol versions and capabilities is a critical step that allows clients and servers to determine their compatibility and the features they both support. The Python SDK abstracts many of the low-level details of the MCP protocol, simplifying the process of building both clients and servers and allowing developers to focus on the specific functionalities they want to implement.

4. Leveraging MCP Tools

MCP tools are a fundamental concept within the protocol, representing executable functionalities that MCP servers expose to clients, allowing LLMs to interact with the external world [1]. Unlike resources, which provide static data, tools enable dynamic operations, allowing LLMs to perform actions, computations, and interact with external systems [31]. It is important to distinguish tools from resources and prompts; while resources offer context and prompts define interaction patterns, tools provide the means for LLMs to take action [16]. MCP tools are designed to be model-controlled, meaning that the intention is for AI models to be able to automatically discover and invoke these tools (often with a human in the loop for approval) to fulfill user requests [31].

MCP clients can discover the tools offered by a server by using a specific protocol endpoint, typically tools/list [31]. The response to this request provides a list of available tools, with each tool definition including a name, a human-readable description of its functionality, and a JSON schema that specifies the expected input parameters [31]. This schema is crucial as it guides the LLM in understanding how to correctly use the tool. Once a client (or an LLM integrated with a client) determines that a particular tool is relevant to the user's request, it can invoke the tool using another protocol endpoint, such as tools/call [31]. This invocation includes the name of the tool to be called and the parameters, which must conform to the JSON schema defined in the tool's description. The server then executes the requested operation and returns the result to the client in a structured format.

The MCP ecosystem boasts a growing number of pre-built servers that offer a wide range of tools for various real-world applications [2]. For instance, the **GitHub Server** provides tools for interacting with GitHub repositories, enabling functionalities like repository management, file operations, and integration with the GitHub API [2]. To configure this server, one might use a command-line tool like npx or uvx and provide necessary credentials such as a GitHub personal access token [32]. Once configured, an LLM connected via an MCP client could use tools from this server to perform actions like creating pull requests or searching for specific code within a repository [20]. Similarly, the **Slack Server** offers tools for managing Slack channels and sending messages [2]. Its configuration would typically involve providing a Slack bot token, allowing an AI to interact with Slack workspaces by reading and writing messages [10]. The **Filesystem Server** provides tools for secure file operations, allowing an LLM to read, write, create, list, delete, move, and search files and directories on the local file system, with configurable access controls for security [32]. Running this server might involve using npx and specifying the directories that the server is allowed to access [35]. Another useful example is the **Brave Search Server**, which provides tools for performing web and local searches using Brave's Search API [12]. Configuring this server usually requires obtaining a Brave API key and installing the server, often using a package manager like npm [12]. Once set up, AI clients like Claude Desktop can utilize this tool to perform web searches based on user prompts [34].

MCP servers can be configured in several ways [20]. One common method is by directly editing configuration files, such as the claude_desktop_config.json file used by Claude Desktop to register and configure MCP servers [20]. Configuration details might include the command to run the server, any command-line arguments it requires, and environment variables, such as API keys [20]. Some servers might also accept configuration directly through command-line arguments when they are launched [33]. While some AI clients might offer a graphical user interface for adding MCP servers, manual configuration via files is often necessary for more advanced settings or when environment variables are required [20].

The existence of a growing ecosystem of readily available MCP servers is a significant advantage for developers looking to integrate AI with external systems. These pre-built connectors for popular services and data sources enable rapid development and deployment of context-aware AI applications without the need to build every integration from the ground up. Understanding how to configure and utilize these existing tools is a crucial first step in leveraging the power of the Model Context Protocol.

5. Creating and Integrating Custom MCP Tools

Designing custom MCP tools allows developers to extend the capabilities of LLMs to address specific needs and integrate with unique systems [16]. The process begins with clearly defining the intended functionality of the tool, the necessary inputs it will require, and the structure of its output [31]. It is also important to consider potential error scenarios and how the tool should report them back to the client [31]. Security should be a paramount concern, especially when the tool interacts with external systems or handles sensitive data; careful thought should be given to authentication, authorization, and input validation [31].

The Python SDK provides a straightforward way to implement custom tools on the server-side [16]. By using the @mcp.tool() decorator to annotate a Python function, developers can register that function as an MCP tool [16]. The function's parameters, defined with type hints, are automatically used by the SDK to generate the tool's input schema, which is then advertised to MCP clients [16]. For example, a custom tool to calculate the Body Mass Index (BMI) could be implemented as a Python function that takes weight_kg and height_m as input parameters with float type hints and returns the calculated BMI as a float [16]. The docstring provided for the function serves as the tool's description, which is also made available to clients [16]. More complex tools that need to interact with internal systems or external APIs can also be implemented using this approach. The SDK allows for handling context and dependencies within tool implementations, enabling access to resources that might be initialized when the server starts [16]. Clear and concise docstrings are essential for ensuring that LLMs can understand the purpose and usage of the custom tool [16].

Once a custom tool has been implemented on an MCP server, any compatible MCP client can discover and utilize it [16]. The client first retrieves the list of available tools from the server, which will include the newly created tool along with its name, description, and input schema. When the client needs to use this tool, it will send a tools/call request to the server, specifying the name of the tool and providing the required parameters in a format that matches the tool's schema. For instance, using the BMI calculation tool example, a client would send a request with the tool's name and the weight and height values as parameters. The server would then execute the calculate_bmi function and return the result (the BMI value) to the client. For initial testing and understanding, a simple MCP client or the MCP Inspector tool can be used to manually call the custom tool and inspect the response. In more advanced scenarios, AI clients like Claude Desktop or Cursor, when prompted by a user, can automatically discover the available custom tools and potentially invoke them based on their understanding of the user's intent and the tool's description and schema [20].

The ability to easily create custom tools using the MCP SDKs, particularly the Python SDK with its intuitive decorator syntax, is a powerful feature of the protocol. It allows developers to tailor the capabilities of LLMs to their specific needs by integrating with internal systems, proprietary data sources, or custom functionalities. By carefully designing the tool's interface and implementation, developers can seamlessly extend the reach and utility of AI applications within their unique contexts.

6. Real-World Implementation and Use Cases

The Model Context Protocol is finding increasing application in a variety of real-world scenarios, demonstrating its versatility and potential to transform how AI interacts with external systems [2]. In the realm of **enhanced code editors and IDEs**, MCP enables seamless integration with version control systems like Git, GitHub, and GitLab. This allows AI assistants embedded within these tools to perform tasks such as code analysis, suggesting improvements, creating pull requests, and tracking project issues, all through standardized MCP interfaces [2]. For **productivity and collaboration tools**, MCP facilitates connections to email platforms like Gmail, messaging apps such as Slack, and project management systems like Linear, Jira, and Todoist. This integration allows for automation of tasks, retrieval of information from these platforms, and more intelligent workflows driven by AI [2]. In the domain of **data analysis and business intelligence**, MCP provides a standardized way for AI to interact with various databases, including PostgreSQL, SQLite, BigQuery, and Snowflake. This enables users to leverage natural language queries to analyze data, generate reports, and gain insights directly through their AI assistants [10]. For tasks involving **web automation and information retrieval**, MCP can be used to connect to browser automation tools like Puppeteer and search engines like Brave Search, allowing AI to scrape websites, gather real-time information, and perform other web-based tasks more effectively [12]. Finally, in **knowledge management systems**, MCP can facilitate integration with knowledge bases and document repositories such as Google Drive and Obsidian, enabling AI models to retrieve context-aware information to enhance their responses and provide more relevant assistance [1].

The adoption of MCP is growing across a diverse range of AI clients and development platforms, highlighting its platform-agnostic nature and the increasing recognition of its value [1]. Several prominent AI clients already support MCP, including Claude Desktop, Cursor, Zed, Replit, and Sourcegraph Cody, among others like Continue and Cline [1]. Furthermore, MCP is being integrated into AI development frameworks such as LangChain4j and PydanticAI, making it easier for developers to build MCP-enabled applications [15]. This widespread support underscores the fundamental principle of MCP: to provide a common language that allows any compliant client to communicate with any compliant server, regardless of the underlying AI model or the specific tool or data source [4]. This interoperability is key to fostering a rich and diverse ecosystem of AI applications and tools.

The expanding range of practical applications and the increasing support from various AI clients and platforms strongly suggest that MCP is poised to become a cornerstone standard for integrating LLMs with the external world. This growing ecosystem encourages developers to create more MCP servers, further enriching the available tools and data sources and driving even wider adoption of the protocol.

7. Further Learning and Resources

For those interested in delving deeper into the Model Context Protocol, several valuable resources are available. The **official MCP documentation** provides a comprehensive overview of the protocol, its architecture, and its specifications. This can be found at the official website: modelcontextprotocol.io and the specification website: spec.modelcontextprotocol.io [3].

The **official MCP GitHub repositories** are also excellent resources for developers. The main repository, github.com/modelcontextprotocol [7], hosts the SDKs for various programming languages (including Python, TypeScript, Java, and Kotlin), the protocol specification, and the core documentation. The repository dedicated to example servers, github.com/modelcontextprotocol/servers [7], contains a wealth of reference implementations and community-contributed servers that demonstrate how to connect to different data sources and tools.

Beyond the official resources, the **MCP community** is a valuable source of information and support. The Awesome MCP Servers list [32] and the GitHub Discussions forum [32] are great places to discover community-driven projects and engage with other developers. Community channels on platforms like Discord [3] also provide opportunities to ask questions, share experiences, and stay updated on the latest developments.

The **MCP Inspector** is an essential tool for developers building and testing MCP servers [5]. This visual testing tool allows developers to interact with their servers, send requests, and inspect the responses, greatly simplifying the debugging process.

Experimenting with the **example servers and clients** provided in the official and community repositories is highly recommended for gaining practical experience with MCP [12]. Running these examples and modifying them to suit specific use cases is an effective way to deepen understanding of the protocol and its capabilities.

For those who are interested in contributing to the growth of the MCP ecosystem, the project is open-source, and contributions are welcome [2]. Building new MCP servers or enhancing existing ones helps to expand the range of tools and integrations available to the community.

Finally, staying informed about the latest news and developments in the MCP ecosystem is important. Following the Anthropic blog and keeping an eye on announcements from the MCP community will help users stay up-to-date with new features, updates, and best practices [1].

The Model Context Protocol ecosystem is still in its early stages but is rapidly evolving. By actively engaging with the official documentation, exploring the open-source repositories, participating in the community, and experimenting with existing examples, users can gain a thorough understanding of MCP and its potential to revolutionize the way AI applications interact with the world around them.

Works cited

1. What is the Model Context Protocol (MCP)? - WorkOS, accessed on March 16, 2025, https://workos.com/blog/model-context-protocol
2. Introducing the Model Context Protocol - Anthropic, accessed on March 16, 2025, https://www.anthropic.com/news/model-context-protocol
3. Model Context Protocol (MCP) - Anthropic, accessed on March 16, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/mcp
4. What is Model Context Protocol (MCP)? How it simplifies AI integrations compared to APIs, accessed on March 16, 2025, https://norahsakal.com/blog/mcp-vs-api-model-context-protocol-explained/
5. Model Context Protocol (MCP) in AI | by BavalpreetSinghh | Mar, 2025 - Medium, accessed

on March 16, 2025,
https://medium.com/@bavalpreetsinghh/model-context-protocol-mcp-in-ai-9858b5ecd9ce
6. Everything You Need to Know About the Model Context Protocol (MCP) from Anthropic, accessed on March 16, 2025,
https://medium.com/@glennlenormand/everything-you-need-to-know-about-the-model-context-protocol-mcp-from-anthropic-84acdb3c1a2f
7. Model Context Protocol - GitHub, accessed on March 16, 2025,
https://github.com/modelcontextprotocol
8. What is MCP (Model Context Protocol)? - Daily.dev, accessed on March 16, 2025,
https://daily.dev/blog/what-is-mcp-model-context-protocol
9. What is Model Context Protocol? (MCP) Architecture Overview | by Tahir | Mar, 2025 | Medium, accessed on March 16, 2025,
https://medium.com/@tahirbalarabe2/what-is-model-context-protocol-mcp-architecture-overview-c75f20ba4498
10. What is the Model Context Protocol (MCP) and How It Works - Security Boulevard, accessed on March 16, 2025,
https://securityboulevard.com/2025/03/what-is-the-model-context-protocol-mcp-and-how-it-works/?utm_source=rss&utm_medium=rss&utm_campaign=what-is-the-model-context-protocol-mcp-and-how-it-works
11. Core architecture - Model Context Protocol, accessed on March 16, 2025,
https://modelcontextprotocol.io/docs/concepts/architecture
12. Model Context Protocol (MCP) Quickstart - Glama, accessed on March 16, 2025,
https://glama.ai/blog/2024-11-25-model-context-protocol-quickstart
13. Transports - Specification (Latest) - Model Context Protocol, accessed on March 16, 2025,
https://spec.modelcontextprotocol.io/specification/draft/basic/transports/
14. Transports - Model Context Protocol, accessed on March 16, 2025,
https://modelcontextprotocol.io/docs/concepts/transports
15. Model Context Protocol (MCP) - LangChain4j, accessed on March 16, 2025,
https://docs.langchain4j.dev/tutorials/mcp/
16. The official Python SDK for Model Context Protocol servers and clients - GitHub, accessed on March 16, 2025, https://github.com/modelcontextprotocol/python-sdk
17. MCP - Model Context Protocol - SDK - Python - YouTube, accessed on March 16, 2025,
https://www.youtube.com/watch?v=oq3dkNm51qc
18. State, and long-lived vs. short-lived connections · modelcontextprotocol specification · Discussion #102 - GitHub, accessed on March 16, 2025,
https://github.com/modelcontextprotocol/specification/discussions/102
19. stdio vs SSE transport question · modelcontextprotocol specification · Discussion #63 - GitHub, accessed on March 16, 2025,
https://github.com/modelcontextprotocol/specification/discussions/63
20. Model Context Protocol - Cursor, accessed on March 16, 2025,
https://docs.cursor.com/context/model-context-protocol
21. What is Model Context Protocol (MCP): Explained - Composio, accessed on March 16, 2025, https://composio.dev/blog/what-is-model-context-protocol-mcp-explained/
22. Reflections on building with Model Context Protocol (MCP) - Outlore, accessed on March 16, 2025, https://outlore.dev/blog/model-context-protocol/
23. Simple example client demonstrating how to connect to MCP servers over HTTP (SSE), accessed on March 16, 2025, https://github.com/slavashvets/mcp-http-clilent-example
24. The Hidden Risks of SSE（Server-Sent Events): What Developers Often Overlook, accessed

on March 16, 2025,
https://dev.to/patrick_61cbc6392b72286f6/the-hidden-risks-of-sseserver-sent-events-what-developers-often-overlook-1b34
25. Server-Sent Events (SSE) Are Underrated - Hacker News, accessed on March 16, 2025, https://news.ycombinator.com/item?id=42511318
26. WebSockets vs Server-Sent Events: Explore the Key differences - Apidog, accessed on March 16, 2025, https://apidog.com/blog/websockets-vs-server-sent-events/
27. Downside of using Server-Sent events for bidirectional client-server communication (instead of WebSockets) - Stack Overflow, accessed on March 16, 2025, https://stackoverflow.com/questions/13278365/downside-of-using-server-sent-events-for-bidirectional-client-server-communicati
28. Use any community MCP server remotely over SSE : r/ClaudeAI - Reddit, accessed on March 16, 2025, https://www.reddit.com/r/ClaudeAI/comments/1hs6vg1/use_any_community_mcp_server_remotely_over_sse/
29. Model Context Protocol (MCP) :: Spring AI Reference, accessed on March 16, 2025, https://docs.spring.io/spring-ai/reference/api/mcp/mcp-overview.html
30. What is Model Context Protocol (MCP): Explained in detail - DEV Community, accessed on March 16, 2025, https://dev.to/composiodev/what-is-model-context-protocol-mcp-explained-in-detail-5f53
31. Tools - Model Context Protocol, accessed on March 16, 2025, https://modelcontextprotocol.io/docs/concepts/tools
32. Example Servers - Model Context Protocol, accessed on March 16, 2025, https://modelcontextprotocol.io/examples
33. modelcontextprotocol/servers: Model Context Protocol ... - GitHub, accessed on March 16, 2025, https://github.com/modelcontextprotocol/servers
34. How to Use MCP Tools on Claude Desktop App and Automate Your Daily Tasks - Medium, accessed on March 16, 2025, https://medium.com/@pedro.aquino.se/how-to-use-mcp-tools-on-claude-desktop-app-and-automate-your-daily-tasks-1c38e22bc4b0
35. How to use MCP tools with a PydanticAI Agent | by Finn Andersen | Mar, 2025 | Medium, accessed on March 16, 2025, https://medium.com/@finndersen/how-to-use-mcp-tools-with-a-pydanticai-agent-0d3a09c93a51
36. MCP Documentation - Quick Start, accessed on March 16, 2025, https://www.claudemcp.com/docs/quickstart
37. MCP Server Development Protocol - Cline Documentation, accessed on March 16, 2025, https://docs.cline.bot/mcp-servers/mcp-server-from-scratch
38. The Developer's Guide to MCP: From Basics to Advanced Workflows - Cline Blog, accessed on March 16, 2025, https://cline.bot/blog/the-developers-guide-to-mcp-from-basics-to-advanced-workflows
39. Example Clients - Model Context Protocol, accessed on March 16, 2025, https://modelcontextprotocol.io/clients
40. cli-mcp-server - Glama, accessed on March 16, 2025, https://glama.ai/mcp/servers/q89277vzl1