Model Context Protocol: A Deep Dive into Server Architecture, Remote Connectivity, and Python SDK-Based Framework Development

I. Understanding the Model Context Protocol (MCP)

The Model Context Protocol (MCP) has emerged as a pivotal standard in the advancement of artificial intelligence, particularly in how Large Language Models (LLMs) interact with the broader digital world. This section lays the foundational knowledge of MCP, establishing its importance and core mechanics. It explains *why* MCP exists and *what* it fundamentally does, setting the stage for more complex topics such as remote server creation, client connectivity over the web, and the development of an API-agnostic framework.

A. Core Concepts: What is MCP, its Purpose, and Key Benefits

The Model Context Protocol (MCP) is an open standard, prominently developed and promoted by Anthropic.[1] Its primary function is to standardize how applications provide contextual information to LLMs and, crucially, how LLMs can discover and interact with external tools, data sources, and services.[3] Anthropic compellingly analogizes MCP to a "USB-C port for AI applications" [2], emphasizing its role in simplifying and universalizing connections between AI models and the external environment. This standardization is not merely a matter of convenience for developers; it forms a critical underpinning for the creation of a scalable and robust agentic AI ecosystem. Agentic AI systems derive substantial utility from the capacity of LLMs to engage with a diverse and expanding array of external tools and data repositories. In the absence of a common protocol, each pairing of an LLM with a tool would demand a bespoke integration. This would lead to an unmanageable surge in complexity, often described as the N x M integration problem, where N models must connect to M tools.[6] Such inherent complexity would inevitably act as a brake on innovation, decelerate development timelines, and promote the formation of isolated integration silos. MCP, by providing a universal "language," seeks to significantly reduce this friction. This allows developers to adhere to a "build once and integrate everywhere" paradigm [8], thereby fostering a more dynamic and interconnected environment in which novel tools and AI agents can be more readily conceptualized, developed, and adopted by the wider community.

The **purpose of MCP** is to empower LLMs to dynamically connect with a broad spectrum of external systems. These systems can encompass local data sources, such as files and databases, as well as remote services and APIs, including platforms

like Salesforce, Box, Asana, and numerous others.[1] MCP enables LLMs not only to retrieve data from these systems but also to write data and execute actions.[1]

The adoption of MCP offers several **key benefits**:

- **Simplified Integration & Reduced Development Overhead**: MCP establishes a clearer and more straightforward pathway for LLM providers and Software-as-a-Service (SaaS) applications to integrate. This significantly diminishes the necessity for custom "glue code" for each new tool or data source an LLM needs to access.[1]
- **Standardization & Interoperability**: It provides a common "language" and interaction pattern, enabling AI agents to consistently discover, inspect, and invoke tools.[4] This fosters interoperability, potentially allowing developers to switch between different LLM providers or AI frameworks without the substantial effort of re-implementing all tool integrations, thereby mitigating vendor lock-in.[10]
- **Enhanced LLM Capabilities & Autonomous Agents**: By furnishing structured access to external tools and real-time data, MCP empowers LLMs to execute complex, multi-step workflows autonomously. This transforms LLMs from being merely isolated "brains" into versatile "doers" capable of accomplishing more sophisticated and contextually aware tasks.[1]
- **Improved Security and Governance**: MCP aims to provide standardized governance mechanisms concerning how contextual information is stored, shared, and updated across disparate environments.[1] When architected correctly, it can facilitate the secure containment of sensitive data within an organization's own infrastructure, even while permitting interaction with AI models.[10]

The "USB-C for AI" analogy [2] extends beyond a simple metaphor for convenience. It signifies an ambition for MCP to evolve into a ubiquitous and fundamental layer within the AI technology stack, much like TCP/IP underpins network communication [7] or USB standardizes peripheral connectivity in the hardware domain. Historically, standards such as USB and TCP/IP achieved widespread adoption because they successfully abstracted away underlying complexities and guaranteed interoperability. This, in turn, catalyzed the growth of vast ecosystems of compatible devices and applications. MCP aspires to fulfill a similar role for AI tool integration. Should MCP attain this level of adoption, the implications would be profound: it could stimulate the emergence of new markets for MCP-compliant tools, specialized MCP server hosting and management solutions, security products specifically designed for MCP environments, and potentially entirely new categories of AI agent platforms constructed with MCP as a core architectural assumption. Consequently, a deep understanding and proficiency in MCP would transition into a critical skill set for

developers and architects operating within the AI landscape.

B. MCP Architecture: Hosts, Clients, and Servers Explained

The Model Context Protocol architecture is characterized by three primary roles, a structure consistently detailed across multiple authoritative sources.[7] These roles are the MCP Host, MCP Client, and MCP Server.

1. **MCP Host**: This entity represents the primary AI application that orchestrates the utilization of AI capabilities. Illustrative examples include Anthropic's Claude Desktop [6], Integrated Development Environments (IDEs) like VS Code [8] or Cursor [6], and various chatbots or custom agent frameworks.[11] The Host functions as a "container" or coordinator for one or more MCP Client instances. It assumes responsibility for managing the lifecycle of these clients and for enforcing overarching security policies, permissions, and user consent mechanisms.[7]
2. **MCP Client**: This component typically resides and operates within the MCP Host.[6] Its principal function is to manage communication with a *specific* MCP Server.[5] This encompasses handling the initial capability negotiation phase, orchestrating the flow of messages (both requests and responses) between itself and the designated server, and upholding security boundaries.[7] A defining characteristic of the MCP Client is that each instance establishes a one-to-one, stateful session with an MCP server.[7]
3. **MCP Server**: This is generally a lightweight program, process, or service meticulously designed to expose specific external capabilities—such as tools, resources, or prompts—derived from various underlying systems. These systems can be local in nature (e.g., file systems, local databases) or remote (e.g., third-party APIs, cloud-based services).[4] The MCP Server acts as the crucial intermediary or bridge, translating between the standardized MCP interface and the particular implementation details of the functionality it encapsulates.[11]

This architectural arrangement effectively decouples the core AI application, embodied by the Host, from the concrete implementations of the diverse tools and data sources with which it needs to interact. This decoupling is a key factor in promoting modularity within MCP-based systems.[6]

The delineation of distinct Host, Client, and Server roles within the MCP architecture is fundamental to achieving modularity, a clear separation of concerns, and enhanced security within what can be a complex agentic AI ecosystem. The Host is entrusted with managing the overall user experience and enforcing high-level policies, such as obtaining user consent for actions. The Client component effectively isolates the communication specifics for a single MCP Server, thereby ensuring that interactions

with one tool do not directly impinge upon or interfere with interactions involving another. The Server, in turn, encapsulates a particular external tool or data source, abstracting its unique API or access methodology behind the standardized MCP interface. This pronounced separation of responsibilities allows each component to be developed, deployed, updated, and secured with a degree of independence. For instance, a security vulnerability identified within one MCP Server is less likely to compromise the entire Host application or other connected Servers, particularly if principles of runtime isolation, as emphasized by Microsoft in the context of Windows MCP [8], are effectively implemented. This inherent modularity is also pivotal for scalability, as it allows for new tools (manifested as new Servers) to be introduced into the system, or existing ones to be replaced, without necessitating a comprehensive overhaul of the Host application.

Furthermore, the "one-to-one session" characteristic [7], which defines the relationship between an MCP Client and an MCP Server, carries significant implications for multi-tool AI agents. For an AI agent to concurrently utilize multiple tools, the MCP Host must be capable of managing multiple Client instances, each of which maintains its own distinct and stateful session with its corresponding Server. Consider a sophisticated AI agent that needs to interact simultaneously with several different tools—for example, accessing a calendar API, querying a customer relationship management (CRM) database, and utilizing a file system tool. If each client-server link constitutes an independent session, the Host application (be it Claude Desktop or a custom-built agent framework) bears the responsibility of managing these multiple Client instances. This management scope includes overseeing their individual lifecycles, handling their separate security contexts (which might involve, for instance, distinct OAuth tokens for different remote services), and potentially aggregating or orchestrating the information and actions flowing through these diverse channels. While this undoubtedly adds a layer of complexity to the development of advanced Host applications, it is an essential architectural prerequisite for enabling powerful, versatile, and multi-tool AI agents.

C. Fundamental Primitives: Tools, Resources, and Prompts

MCP Servers expose their functionalities through three fundamental building blocks, or primitives, which are consistently described across various technical documents.[5] These primitives are Tools, Resources, and Prompts.

1. **Tools**: These represent executable functions or actions that the AI model (LLM) can autonomously decide to invoke. Tools are employed to perform operations, trigger external workflows, or interact with external systems. Examples include making API calls, writing to files, or executing database queries.[5] The decision to

utilize a specific tool is typically model-controlled, meaning the LLM itself determines when and how to use an available tool based on the context of the interaction.[7] The definition of a tool is critical for its usability by an LLM; it must include a clear name, a human-readable description (to enable the LLM to understand its purpose and applicability), and a precise schema that defines its input parameters and the expected format of its output.[3]

2. **Resources**: These primitives represent data endpoints that furnish external information or context to the LLM. Resources are typically designed to be read-only or to have minimal side-effects. Examples include fetching the content of a file, retrieving records from a database, or obtaining the current state from an API.[5] The provision or selection of resources is generally application-controlled. This means that the Host application often plays a direct role in determining which resources are made available to the LLM at any given time.[7]
3. **Prompts**: These are predefined conversational templates or reusable sets of instructions. Prompts are designed to guide or structure interactions for specialized tasks and are often invoked directly by the user, for instance, through slash commands in a chat interface or by clicking a button in a GUI.[5] The initiation of a prompt-based interaction is typically user-controlled.[7]

These three primitives collectively enable LLMs to discover what capabilities are available from an MCP server, understand how to use those capabilities effectively, choose the most appropriate one for a given task or query, invoke it with the necessary parameters, and receive structured responses that can be further processed or presented to the user.[3]

The differentiation between Tools, Resources, and Prompts within MCP reflects distinct control flows and interaction patterns inherent in advanced agentic systems. This nuanced approach allows for a more sophisticated and powerful means for LLMs to engage with external capabilities than would be possible through simple, undifferentiated API calls. **Resources**, being application-controlled, permit the host application to proactively furnish context to the LLM (e.g., "Here is the content of the document you are tasked with summarizing"). **Prompts**, which are user-controlled, empower users to directly trigger specific, predefined interactions with the server's capabilities (e.g., a user selecting a "Summarize Open Document" option that, under the hood, invokes an MCP prompt). **Tools**, on the other hand, are model-controlled, granting the LLM itself the autonomy to decide to utilize a capability based on the ongoing conversation and its own reasoning processes (e.g., the LLM independently determining that it needs to call a search_web tool to answer a user's query). This

multi-faceted interaction model facilitates more sophisticated agent behaviors where context can be proactively supplied, user actions can be directly mapped to server functionalities, and the LLM can exhibit initiative in problem-solving.

The efficacy of "dynamic tool discovery" [6] and the quality of an LLM's reasoning about when and how to use these primitives are profoundly dependent on the quality of the metadata associated with them—specifically, their descriptions and schemas. LLMs rely heavily on these provided descriptions and parameter schemas to comprehend what a tool, resource, or prompt is designed to do, and the correct way to utilize it.[3] Primitives that are poorly described or have inaccurately defined schemas will inevitably lead to the LLM making suboptimal choices, failing to leverage available capabilities when appropriate, or misusing them in ways that produce errors or undesirable outcomes. Consequently, the task of crafting clear, concise, and accurate metadata is of coequal importance to the implementation of the underlying functionality itself. This observation has significant implications for the development of best practices and specialized tooling aimed at assisting developers in the creation of high-quality, effective MCP servers.

D. Communication Layer: JSON-RPC 2.0 over HTTP

The communication backbone of the Model Context Protocol is fundamentally JSON-RPC 2.0, typically transported over HTTP for remote interactions.[4] JSON-RPC 2.0 is a lightweight, stateless, and transport-agnostic remote procedure call (RPC) protocol.[4] Its design allows for method invocation across various communication channels, including standard input/output (stdio) for local inter-process communication, HTTP, and WebSockets.[4] However, for the crucial use case of remote MCP servers, HTTP-based transports (specifically HTTP with Server-Sent Events or the more recent Streamable HTTP) are the predominant standards.

The choice of JSON-RPC 2.0 provides MCP with a standardized format for requests and responses [5], a defined structure for error reporting, and a clear mechanism for tool invocation.[3] This reliance on a well-established protocol like JSON-RPC 2.0 accelerates the adoption of MCP. It allows the ecosystem to leverage existing developer knowledge and a wide array of available libraries, rather than necessitating the invention and popularization of an entirely new RPC mechanism. Many developers are already familiar with JSON-RPC, and mature implementations exist in numerous programming languages.[12] Building upon an existing standard effectively lowers the learning curve and reduces the implementation burden for both server and client developers. This reduced barrier to entry is a critical factor in fostering the growth and vibrancy of the MCP ecosystem.

While JSON-RPC itself is transport-agnostic, MCP's practical application for remote access scenarios heavily standardizes on HTTP-based transports, namely HTTP+SSE and Streamable HTTP. This strategic choice brings with it the full suite of web-standard considerations that are pertinent to any HTTP-based service. For instance, security for remote MCP communication inherently relies on HTTPS, necessitating the use of TLS/SSL encryption for data in transit, as highlighted in security best practices.[13] Furthermore, concepts central to web applications, such as HTTP-level session management and robust authentication mechanisms (like OAuth 2.1, discussed later), become integral to the design and deployment of secure remote MCP servers. This also implies that existing HTTP infrastructure components, including load balancers, firewalls, and API gateways, can potentially be leveraged to manage and secure MCP traffic. However, these components must also be configured with a nuanced understanding of MCP's specific communication patterns to ensure correct operation and security.

II. Building Robust Remote MCP Servers

Transitioning from the theoretical underpinnings of MCP, this section delves into the practical and technical aspects of constructing MCP servers that are accessible over the web. The focus here is on the architectural choices, implementation strategies, and critical considerations involved in developing reliable, secure, and scalable remote MCP servers, particularly addressing how clients can connect to such servers using a URL over the web, rather than solely through local stdio/stdout mechanisms.

A. Prerequisites and Setup for Server Development

Developing and deploying a production-grade MCP server, especially one intended for remote access and potentially hosting complex tools or interacting with sensitive data, is a significant engineering endeavor that extends beyond simple scripting.

Knowledge Prerequisites:
A solid foundation in several technical domains is essential. This includes advanced Python programming skills, a conceptual understanding of machine learning (particularly if the tools involve ML models), principles of distributed systems architecture, fundamentals of network programming, and at least basic knowledge of DevOps practices and cloud infrastructure management.13
Hardware and Software Requirements:
While specific needs vary with the complexity and load of the MCP server, general recommendations for production environments often include multi-core 64-bit processors (e.g., Intel Xeon or AMD EPYC series), substantial RAM (32GB minimum, 64GB+ recommended, with ECC memory for critical deployments), and fast SSD/NVMe storage.13 Linux distributions (like Ubuntu 20.04 LTS+ or CentOS/RHEL 8+) are the preferred operating systems, with

Python 3.8 or newer being the primary development language for the modelcontextprotocol/python-sdk.13
Essential libraries frequently cited in the context of MCP server development (though some may be for the tools themselves rather than the MCP framework) include asyncio for asynchronous operations, pydantic for data validation and settings management, and web server components like fastapi and uvicorn.13 The cornerstone for Python-based MCP development is, of course, the official modelcontextprotocol/python-sdk (often referred to as mcp or found at github.com/modelcontextprotocol/python-sdk).14
Development Environment Setup:
Standard Python development practices apply, starting with the creation of a virtual environment (e.g., using python3 -m venv mcp_environment) to isolate project dependencies. This is followed by upgrading core packaging tools (pip install --upgrade pip setuptools wheel) and then installing the necessary libraries, including the MCP SDK (pip install "mcp[cli]").13 Recent documentation and tutorials increasingly highlight the use of uv as a modern, faster alternative to pip and venv for Python package and environment management.14 This adoption of uv within the MCP development ecosystem signals a trend towards more performant and streamlined development workflows. Developers accustomed to older tooling may find it beneficial to familiarize themselves with uv, as future tooling and examples might increasingly assume or recommend its use.
Compliance and Security Policies:
From the outset, development must adhere to relevant data protection regulations (such as GDPR or CCPA, depending on jurisdiction and data handled), internal organizational security policies, and established ethical AI deployment guidelines.13
The comprehensive nature of these prerequisites underscores that while tools like FastMCP (a component of the Python SDK) can simplify the creation of basic MCP servers [14], deploying robust, secure, and scalable MCP servers requires diligent engineering, careful architectural planning, and adherence to operational best practices.

B. Transport Protocols for Remote Communication

For an MCP server to be accessible remotely by clients across different machines or networks, it must utilize a web-standard communication protocol. MCP primarily specifies two HTTP-based transport mechanisms for this purpose.[3] The choice of transport protocol has implications for server implementation, client compatibility, and network infrastructure configuration.

1. **Legacy HTTP with Server-Sent Events (HTTP+SSE)**:
    - This transport, associated with the 2024-11-05 version of the MCP specification, relies on two distinct HTTP endpoints for a single logical session.[3]
    - A GET /mcp endpoint (or a similar path like /sse as seen in many examples [9]) is

used to establish a Server-Sent Events (SSE) stream. SSE is a technology that allows a server to push real-time, unidirectional updates to a connected client over a persistent HTTP connection.[20] This stream is primarily used for server-to-client communication, such as delivering tool responses or server-initiated messages.
    - A separate POST /messages endpoint is used for client-to-server communication, where the client sends its JSON-RPC requests (e.g., tool invocations).[3]
    - Session identification in this model often involves passing a session ID as a query parameter in the URL for the /messages endpoint (e.g., /messages?sessionId=xxx).[3]
    - Many existing remote MCP servers, including those listed by Anthropic from partners like Asana and Intercom, utilize /sse endpoints, indicative of this transport.[9]
2. **Modern Streamable HTTP**:
    - This newer transport, specified in the 2025-03-26 version of MCP, aims to simplify remote communication by using a single POST /mcp endpoint for all bidirectional operations.[3]
    - Both client-to-server requests and server-to-client responses (including streamed data) are handled over this single, persistent POST connection.[3]
    - Session identification is typically managed via an HTTP header, such as Mcp-Session-Id, exchanged between the client and server.[3]
    - Streamable HTTP is generally considered to have lower implementation complexity and more straightforward connection management compared to the dual-endpoint HTTP+SSE approach.[3] It can support responses in either plain JSON or an SSE-formatted stream over the POST response body.[14]

The evolution from the dual-endpoint HTTP+SSE model to the single-endpoint Streamable HTTP reflects a broader trend in web protocols towards simplification and efficiency. Managing two separate connections for what is logically a single session, as required by the older HTTP+SSE mechanism, inherently adds complexity to both client and server implementations and can be less efficient in terms of resource utilization and network overhead. Streamable HTTP's single-endpoint architecture aligns more closely with typical HTTP request-response patterns, even when dealing with streaming data, making it easier for developers to reason about, implement, and debug. This simplification has the potential to lower the barrier to entry for new MCP server developers and contribute to more robust and performant implementations. Furthermore, this protocol difference also influences how network infrastructure components like load balancers, firewalls, and API gateways need to be configured to

correctly handle MCP traffic.

For optimal interoperability, especially during the transition period as clients adopt the newer Streamable HTTP, production-ready MCP servers should ideally be capable of supporting both transport protocols.[3] For example, the Cloudflare Agents SDK is designed to allow a single server instance to simultaneously handle both Streamable HTTP and legacy HTTP+SSE requests.[23] The modelcontextprotocol/python-sdk also provides support for these standard transports [14], and includes examples for servers using stdio, SSE, and Streamable HTTP.[14] However, supporting both transports concurrently does introduce additional overhead in terms of server development, testing, and maintenance. While many existing clients and prominent services were initially built using the HTTP+SSE model [9], the advantages of Streamable HTTP suggest it will likely become the preferred standard over time. For now, dual support remains a pragmatic consideration for achieving the broadest client compatibility. This also implies that client SDKs and tools, such as Anthropic's MCP Connector [2], need to be able to either auto-detect the server's supported transport or be explicitly configured for the correct one.

The following table provides a comparative analysis of these two HTTP transport protocols:

Table II.B.1: Comparative Analysis of MCP HTTP Transport Protocols

Feature	Streamable HTTP (Modern)	HTTP+SSE (Legacy)
Protocol Version	2025-03-26 [3]	2024-11-05 [3]
SDK Transport Class	StreamableHTTPServerTransport (conceptual) [3]	SSEServerTransport (conceptual) [3]
Endpoints	Single POST /mcp [3]	Dual: GET /mcp (or /sse) + POST /messages [3]
Client-to-Server Comm	POST /mcp [3]	POST /messages?sessionId=xxx [3]
Server-to-Client Comm	Same POST response (streamed) [3]	GET /mcp (SSE stream) [3]

Session Identification	Mcp-Session-Id HTTP header [3]	Query parameter in URL (e.g., sessionId) [3]
Session Termination	DELETE /mcp or connection close [3]	Connection close [3]
Implementation Complexity	Lower [3]	Higher [3]
Connection Management	Simpler, connection terminates naturally [3]	Requires explicit management of two connections [3]
Primary Benefit	Simpler, single connection, more efficient [3]	Compatible with older/existing clients [3]
Key Considerations	Better for serverless; supports JSON or SSE response formats.[14]	Persistent SSE connection may impact serverless scale-to-zero.[3]

This table is valuable as it directly addresses a core technical decision point in MCP server development. It provides a clear, concise comparison of the two main remote transport options, enabling developers to make informed choices based on their specific needs for client compatibility, implementation effort, and desired architectural properties.

C. Implementing Session Management for Stateful Interactions

Session management is a cornerstone of MCP, essential for maintaining conversational context and state across multiple requests between an LLM application (via an MCP Client) and an MCP Server.[3] Without effective session management, complex interactions requiring memory of previous exchanges or ongoing state (such as managing a multi-step task or a shopping cart) would be impractical.

Upon an initial connection from a client, the MCP server is responsible for creating a unique session identifier (session ID). This ID must then be communicated back to the client. For Streamable HTTP, this is typically done via the Mcp-Session-Id HTTP header in the server's response to the initial POST /mcp request.[3] The client is then expected to include this session ID in all subsequent requests pertaining to that session, allowing the server to retrieve and utilize the correct session context. If a session expires or becomes invalid, the server typically responds with an error (e.g.,

HTTP 404 Not Found), signaling the client to reinitialize the session if necessary.[3]

For **simple implementations**, particularly in single-instance servers or during development, session data (which might include the transport instance itself or associated state) can be stored in memory. For example, a Python dictionary or a JavaScript Map object can be used to map session IDs to their corresponding session data or transport objects.[3] High-level frameworks like FastMCP within the modelcontextprotocol/python-sdk likely handle much of this in-memory session management automatically for basic use cases, abstracting these details from the developer.[19]

However, **complex scenarios** involving scalability, fault tolerance, or persistence necessitate more robust session management strategies.[3] These scenarios include:

- Running multiple server instances behind a load balancer.
- Ensuring session state survives server restarts or deployments.
- Handling very high traffic volumes with thousands of concurrent sessions.

In such cases, relying solely on in-memory session storage is inadequate because session data is not shared across different server processes or instances. If a load balancer directs subsequent requests from the same client session to a different server instance that does not have the session data in its memory, the session will be effectively lost. This is a known issue with the stateful nature of the modelcontextprotocol/python-sdk when deployed in multi-worker environments (e.g., using gunicorn with more than one worker, or in a Kubernetes cluster) without an external shared session store. A workaround is to configure the server to use only a single worker, but this severely limits scalability.[28]

To address these challenges, **persistent and distributed session stores** are recommended, such as Redis or a traditional database.[3] These external stores allow all server instances to access and update shared session data, enabling true horizontal scalability and resilience. The mcp-CodeAnalysis server, for instance, provides a practical example of using Redis for session storage in production environments, while gracefully falling back to an in-memory store for development purposes. This server features a modular session store architecture, which is a commendable pattern.[29]

There is a fundamental tension between MCP's original design, which leans towards rich, stateful interactions requiring persistent connections and server-side session memory, and the architectural patterns favored by modern, highly scalable, and often stateless web services (particularly serverless architectures). MCP's core protocol often involves stateful sessions to manage ongoing context, subscriptions to resource

changes, and even server-initiated messages.[7] In contrast, contemporary scalable architectures frequently advocate for stateless application tiers, offloading any necessary state to external, specialized services like Redis or databases. This approach allows application instances to be easily created, destroyed, and scaled horizontally without loss of session continuity. The default in-memory session management behavior implied by issues like [28] for the Python SDK can create significant challenges in these distributed environments, as session data is not inherently shared. This necessitates either architectural workarounds (such as configuring single-worker processes or using "sticky sessions" at the load balancer level, which can have their own drawbacks) or, more ideally, explicit integration by the developer with distributed session stores. Such integration, however, adds to the implementation complexity for those aiming for high scalability and resilience.

The evolution of the MCP specification and its associated SDKs appears to be actively addressing this dichotomy. Features such as resumability in the Streamable HTTP transport [23], and ongoing discussions within the MCP community regarding better support for stateless server profiles and the use of session tokens that can encapsulate or identify externally stored state [30], are indicative of efforts to make MCP more amenable to stateless and serverless deployment models. These developments suggest that future iterations of the protocol and SDKs will likely offer more robust built-in support or clearer, standardized patterns for managing state in distributed environments. This could potentially reduce the current burden on individual developers to engineer custom solutions for session persistence and sharing. For the present, however, developers utilizing the Python SDK to build scalable remote MCP servers must proactively consider and implement strategies for external session state management if their tools or interactions require state to be maintained across multiple requests in a distributed deployment. The modular session store approach seen in the mcp-CodeAnalysis example [29]—using Redis for production and an in-memory store for development—offers a valuable architectural pattern that could be adopted when building the API-agnostic framework. Such a design allows the same core server codebase to adapt to different operational environments through configuration, enhancing both flexibility and deployability.

D. Essential Security for MCP Servers

Security is paramount for MCP servers, especially those accessible remotely, as they can expose powerful tools and potentially sensitive data. A multi-layered security approach is necessary, encompassing authentication, authorization, protection against common vulnerabilities, and adherence to general web security best

practices.

Authentication and Authorization:

- **OAuth 2.1** is the designated standard for AI clients (or MCP Host applications on their behalf) to request access to protected resources on MCP servers.[3] This framework addresses two distinct but related processes:
  - **User Authentication**: Verifying the identity of the human user who is interacting with the MCP Host application.
  - **AI Client Authorization**: Granting the LLM application (acting through an MCP Client) permission to access specific tools or resources on the MCP server on behalf of the authenticated user.[3]
- The **OAuth 2.1 Authorization Code Grant flow** (often with PKCE for public clients like desktop or mobile applications) is typically employed. This flow involves several exchanges:
  1. The client application redirects the user to the MCP server's authorization endpoint (e.g., /authorize).
  2. The user authenticates with the MCP server's identity provider (which could be the server itself or an external IdP) and grants consent for the requested scopes.
  3. The authorization server redirects the user back to the client's registered callback URI with a temporary authorization code.
  4. The client application exchanges this authorization code for an access token (and potentially a refresh token) at the MCP server's token endpoint (e.g., /token).[3]
  5. The client then includes the obtained access token (as a Bearer token in the Authorization HTTP header) in all subsequent requests to the MCP server's protected endpoints.[3]
- MCP servers acting as OAuth 2.1 providers must implement the necessary endpoints: /.well-known/oauth-protected-resource and /.well-known/oauth-authorization-server for discovery, /authorize for user authorization, /token for token issuance, and optionally /register for dynamic client registration and /revoke for token revocation.[3]
- The modelcontextprotocol/python-sdk (specifically mcp.server.auth) provides an OAuthServerProvider protocol (an interface). FastMCP can be initialized with an implementation of this protocol (auth_server_provider) and AuthSettings (which define issuer URL, scopes, client registration options, etc.) to enable OAuth 2.0/2.1 capabilities.[14]
- Once authenticated and authorized, MCP tool handler functions within the server typically receive an authInfo object containing details like the access token, client

ID, and granted scopes. This allows for fine-grained, per-user authorization decisions within the tool logic itself.[3]

Implementing a full, secure OAuth 2.1 provider from scratch is a complex and potentially error-prone undertaking. The OAuthServerProvider interface within the Python SDK [14] offers a crucial structural abstraction for this task. However, comprehensive, readily accessible examples demonstrating a complete implementation of this interface appear to be limited in the provided documentation snippets [[14] often show MyOAuthServerProvider() as a placeholder]. OAuth 2.1 involves numerous components and flows (authorization endpoint logic, token endpoint security, client credential validation, PKCE handling, various grant types, token validation, and revocation mechanisms). While an interface like OAuthServerProvider helps organize these elements, the intricate logic for each part must be correctly implemented by the developer. The scarcity of detailed, official examples for this provider within the SDK's public documentation or immediately accessible examples means that developers might face challenges or inadvertently introduce security vulnerabilities. This highlights a potential area where more detailed guidance or reference implementations would be beneficial. Developers might consider leveraging established Python OAuth libraries like Authlib [31] to implement the core logic required by the OAuthServerProvider interface, thereby reducing the risk of common OAuth pitfalls.

Windows 11 MCP Security Architecture Insights:
Microsoft's approach to integrating MCP into Windows 11 includes a comprehensive security architecture that, while specific to the Windows platform, offers valuable principles for any secure MCP deployment 8:

- **Proxy-Mediated Communication**: All MCP client-server interactions are routed through a trusted Windows proxy. This enables centralized enforcement of security policies, user consent management, authentication and authorization, and transparent auditing of operations performed on behalf of the user.
- **Tool-Level Authorization**: Users are required to explicitly approve each client-tool pairing, potentially with granular permissions per resource, ensuring user control.
- **Central Server Registry**: Windows plans to maintain a registry of MCP servers that meet baseline security criteria, aiding discoverability while maintaining trust.
- **Runtime Isolation and Least Privilege**: Servers are expected to adhere to the principle of least privilege, with Windows enforcing declarative capabilities and isolation to limit the potential impact of a compromised server.
- **Server Security Requirements**: To be listed, servers must meet criteria such as mandatory code signing, immutable tool definitions at runtime, security testing of

exposed interfaces, mandatory package identity, and explicit declaration of required privileges. Microsoft acknowledges the significant security implications of agentic computing, where MCP servers can become potent attack vectors.[8] Their multi-layered security strategy aims to mitigate these risks through centralized control and rigorous enforcement. Even for MCP deployments outside the Windows ecosystem, concepts such as a trusted intermediary proxy for policy enforcement or a curated registry of vetted MCP servers could substantially enhance security, particularly for enterprise or public-facing MCP services. This suggests a potential future trajectory where MCP security involves not only server-side hardening by individual developers but also the development of ecosystem-level infrastructure and governance frameworks.

General Vulnerabilities and Mitigations:
MCP servers, like any networked service, are susceptible to various attacks. Awareness and proactive mitigation are crucial 6:

- **Command Injection**: Can occur if inputs to tools (especially those executing shell commands or database queries) are not properly validated and sanitized. Mitigation involves rigorous input validation and using parameterized queries or safe APIs.
- **Data Exfiltration**: Maliciously crafted prompts could instruct an LLM to use MCP tools to access and leak sensitive data. Mitigations include implementing strict, granular access controls on tools and resources, applying the principle of least privilege, and minimizing the data exposed through any single tool.
- **Content/Tool Poisoning**: Attackers might try to inject malicious instructions into data that an MCP resource provides, or attempt to alter a server's tool definitions if they are mutable at runtime. Windows' requirement for immutable tool definitions [8] and general server hardening practices help mitigate this.
- **Access Control and Authentication Vulnerabilities**: Weak or improperly implemented authentication (e.g., flawed OAuth flows) can allow unauthorized access. Robust implementation of OAuth 2.1 and strong credential management are essential.
- **Server Hardening**: General security best practices for web servers apply, including keeping software up-to-date, minimizing exposed services, and secure configuration.
- **Network Security**: Firewalls should be configured to allow traffic only on necessary ports (e.g., 50051 for gRPC if used by an underlying tool, 8000 or 443 for HTTP/HTTPS MCP traffic). Strict IP whitelisting should be implemented where feasible. All external communication must use TLS/SSL (HTTPS) to protect data in transit.[13] For administrative access to servers, VPNs are recommended.[13]

The dual nature of authentication often present in MCP scenarios—where a user first authenticates to the MCP Host application (e.g., Claude Desktop), and then the Host application (via its MCP Client component) authenticates to the MCP Server on behalf of the user using OAuth—creates a chained trust model. This model requires careful management to maintain security. The user places trust in the MCP Host. The Host, through its MCP Client, then requests authorization from the MCP Server to perform actions on the user's behalf. The MCP Server, in turn, needs to be able to trust that the OAuth request genuinely originates from an authorized client application and that the user has indeed provided consent. This flow, while standard for OAuth 2.1, demands that both the client application and the MCP server correctly and securely implement their respective roles in the OAuth exchange to prevent vulnerabilities such as confused deputy attacks or the leakage of sensitive tokens.

E. Deployment Strategies and Considerations

Deploying an MCP server involves more than just running a Python script; it requires careful consideration of the target environment, scalability needs, performance requirements, and operational manageability.

Deployment Models:

- **Local Development/Testing**: For initial development, prototyping, and testing, a single-instance MCP server running directly on a developer's machine is common. This typically uses minimal resources and often relies on stdio or a simple HTTP server for communication.[13] The mcp.run() command from FastMCP is suited for this.[17]
- **Containerized Deployments (Docker/Kubernetes)**: For production and scalable cloud environments, containerization with Docker is highly recommended. Kubernetes can then be used to orchestrate these containers, providing scalability, resilience, and automated management.[13]
- **Distributed Deployments**: For enterprise-scale applications or those anticipating high traffic volumes, a distributed architecture with multiple server nodes, often behind a load balancer, is necessary.[13] This model requires careful attention to session management (as discussed previously) to ensure state consistency across nodes.

Cloud Platform Options:
Several cloud platforms offer suitable environments for hosting remote MCP servers 3:

- **Google Cloud Run**: A serverless platform that automatically scales instances (including to zero) and is well-suited for containerized applications.
- **Vercel**: Known for zero-configuration deployments, particularly strong for Node.js

applications but can host Docker containers. Offers an edge network for low-latency access.
- **Railway**: Provides automatic deployments from GitHub repositories and includes options for integrated databases.
- **DigitalOcean App Platform**: Simplifies deployment from Git repositories, manages SSL certificates, and is a good option for small to medium-sized applications.

Serverless Considerations:
The choice of transport protocol and session management strategy significantly impacts suitability for serverless platforms.

- The legacy HTTP+SSE transport's requirement for persistent SSE connections can prevent serverless platforms from scaling instances down to zero when idle, potentially leading to higher operational costs.[3]
- The newer Streamable HTTP protocol, with its single request/response model (even for streaming), is generally better aligned with the ephemeral nature of serverless functions.
- However, the inherently stateful design of MCP sessions can still pose a challenge. If the Python SDK's default in-memory session management is used, deploying to a stateless serverless environment will necessitate externalizing session state (e.g., to Redis or a similar service), which adds architectural complexity.[28]

Performance Tuning and Optimization:
Beyond basic deployment, ensuring optimal performance requires ongoing attention 13:

- **Dynamic Resource Allocation**: In containerized environments like Kubernetes, configure horizontal pod autoscalers to adjust the number of server instances based on CPU, memory, or custom metrics.
- **Caching Strategies**:
  - **Context Caching**: Implement application-level caching for frequently accessed data or tool results to reduce latency and load on backend systems. An LRU (Least Recently Used) cache is a common approach.
  - **Multi-Tier Caching**: For more demanding applications, consider using distributed caching systems like Redis.
- **Asynchronous Request Handling**: Leverage Python's asyncio capabilities to handle I/O-bound operations non-blockingly. This is crucial for MCP servers that interact with external APIs or databases, allowing the server to efficiently manage many concurrent client connections. Request queuing and prioritization can also be implemented for very high load scenarios.

The decision of which deployment strategy to adopt is therefore deeply interconnected with choices made earlier regarding transport protocols and session management. For instance, if a serverless architecture (like Google Cloud Run) is the target, Streamable HTTP is a more appropriate transport than HTTP+SSE due to the latter's issues with persistent connections.[3] Furthermore, given that serverless functions are typically stateless, using the Python SDK's default in-memory session management would necessitate the integration of an external, shared session store (e.g., Redis [28]), adding a layer of design and operational complexity. Conversely, a more traditional deployment on virtual machines or a Kubernetes cluster with fewer, longer-lived server instances might more readily accommodate stateful server designs or even the HTTP+SSE protocol, but this approach generally requires more direct infrastructure management.

Ultimately, achieving "production-readiness" for an MCP server extends far beyond simply writing the core server logic. It encompasses a holistic approach that includes robust deployment pipelines (CI/CD), comprehensive monitoring (for performance, errors, and resource utilization), thorough logging, and diligent security hardening—practices that are standard for any mission-critical web service.[13] While a simple mcp.run() [17] is adequate for local development and experimentation, transitioning to a production environment demands this more comprehensive and operationally mature strategy.

III. Enabling Client Connectivity to Remote MCP Servers

A primary goal of MCP is to enable broad interoperability, which necessitates clients being able to connect to MCP servers regardless of their physical or network location. This section details how clients, particularly those not on the same machine or local network as the server, can establish connections and interact with remote MCP servers using standard web protocols and URLs.

A. Standard Client Connection via URL (HTTP/HTTPS)

The fundamental mechanism for a client to connect to a remote MCP server is via an HTTPS URL.[3] This aligns MCP with standard web practices, allowing clients to reach servers deployed anywhere on the internet.

- **URL Structure**: The specific path within the URL that a client targets depends on the transport protocol implemented by the server.
    - For servers using the **modern Streamable HTTP** transport, clients will typically connect to a single endpoint, often /mcp (e.g., https://example-mcp-server.com/mcp).[3]

    - For servers using the **legacy HTTP+SSE** transport, clients need to interact with two endpoints: one for establishing the SSE stream (e.g., GET https://example-mcp-server.com/sse) and another for sending messages (e.g., POST https://example-mcp-server.com/messages).[3]
- **Communication Protocol**: Regardless of the specific transport, the underlying communication payload consists of JSON-RPC 2.0 messages exchanged over the established HTTP(S) connection.[4]
- **SDK Support**: The modelcontextprotocol/python-sdk provides client-side components and capabilities that enable Python applications to connect to any MCP server using these standard HTTP-based transports.[14]

The ability for clients to connect via a standard URL is pivotal to realizing MCP's vision of an interoperable ecosystem of AI tools and agents that are not constrained to local execution environments. While local stdio communication [3] is practical for personal tools or development scenarios, a vast number of valuable tools and data sources are, in fact, web services provided by third parties.[9] URL-based connectivity empowers AI agents to tap into this extensive ecosystem of remote services, thereby significantly expanding their potential capabilities far beyond what is available locally. This is a key enabler for the "USB-C for AI" vision, where any compliant client can seamlessly connect to any compliant server, irrespective of their respective locations.

However, client implementations must be cognizant of the specific transport protocol (Streamable HTTP versus HTTP+SSE) supported by the target server's URL. As detailed previously [3], these two protocols utilize different endpoint structures and communication patterns. A client application configured to use Streamable HTTP (expecting a single /mcp endpoint) will encounter errors if it attempts to connect to an older server that only exposes the dual-endpoint HTTP+SSE mechanism. Consequently, client SDKs or the applications using them must either incorporate a mechanism to automatically detect the protocol spoken by the server [72] or require explicit configuration from the user or developer that specifies the server type or provides the correct set of endpoints. This introduces a necessary layer of complexity in client configuration to ensure successful communication.

B. Leveraging Anthropic's MCP Connector with the Messages API

To simplify the integration of Claude with remote MCP servers, Anthropic offers an "MCP connector." This feature allows developers to connect Claude to these servers directly through the Messages API, obviating the need to build and manage a separate, standalone MCP client application.[2]

Key Characteristics and Usage:

- **Beta Feature**: Access to the MCP connector currently requires a specific beta header in API requests: "anthropic-beta": "mcp-client-2025-04-04".[26]
- **Core Functionality**: The connector provides direct API integration for MCP, supports tool calling, handles OAuth Bearer tokens for authentication with secure servers, and allows connections to multiple MCP servers within a single Messages API request.[26]
- **Current Limitations**: As of the available information, the connector primarily supports MCP tool calls; support for accessing MCP resources or invoking MCP prompts was not initially included. The target MCP server must be publicly accessible via HTTP(S); local stdio servers cannot be connected directly. Additionally, the connector was not supported on Amazon Bedrock and Google Vertex AI platforms at the time of documentation.[26]
- **API Invocation**: To use the connector, developers include an mcp_servers parameter in their Messages API request. This parameter takes an array of server objects, each specifying the server's url (which must be HTTPS), a unique name for identification, optional tool_configuration (to enable or restrict specific tools from that server), and an authorization_token if the server requires OAuth authentication.[26]
- **Abstraction**: The Anthropic API transparently handles the underlying MCP connection management, tool discovery (listing available tools from the server), and error handling related to the MCP interaction.[25]
- **Response Structure**: When Claude utilizes tools via the MCP connector, the Messages API response will include new content block types: mcp_tool_use (indicating a tool is being invoked, with its ID, name, server name, and input parameters) and mcp_tool_result (providing the outcome of the tool execution, including success/error status and content).[26]

Anthropic's MCP Connector significantly lowers the barrier to entry for developers seeking to integrate Claude with the growing ecosystem of existing remote MCP tools. It effectively abstracts away many of the complexities involved in building a full-fledged MCP client, such as handling protocol handshakes, managing session state for stateful interactions, dealing with the specifics of different transport protocols, and implementing client-side OAuth 2.1 flows.[3] By allowing developers to simply provide a server URL and an OAuth token (if required) within their standard Messages API call, the connector accelerates development cycles and makes it substantially easier to experiment with and consume capabilities from third-party MCP tools, such as those provided by Asana, Zapier, and others listed in [9], without

requiring deep expertise in MCP client implementation.

The documented limitations of the MCP Connector, such as its initial focus on tool calls to the exclusion of resources and prompts [26], suggest that it is an actively evolving feature. The MCP standard itself defines tools, resources, and prompts as its three key primitives.[7] While the initial emphasis of the connector on "tool calls" addresses a primary and highly valuable use case (i.e., endowing LLMs with the ability to perform actions), it does not yet encompass the full spectrum of interactions defined by MCP. As the MCP ecosystem continues to mature and the demand for richer interaction patterns—such as providing LLMs with large contextual resources or enabling users to invoke complex server-defined prompts—intensifies, it is probable that Anthropic will enhance the connector to include support for these other primitives. Developers who choose to rely on the MCP Connector should therefore remain cognizant of its current capabilities and monitor its evolution for new features and expanded support.

C. Using the mcp-remote Bridge Tool for Local Clients

The mcp-remote tool serves as an adapter or bridge, designed to enable MCP clients that were originally built for local (stdio-based) connections to communicate with remote MCP servers accessible via HTTP(S).[3] This is particularly useful for older client applications or environments, like earlier versions of Claude Desktop or other local AI tools, that may not have native support for direct remote MCP connections.

The bridge tool essentially acts as an intermediary: it receives MCP requests from the local client application (typically over stdio) and forwards these requests to the specified remote MCP server over the appropriate HTTP transport (HTTP+SSE or Streamable HTTP). Responses from the remote server are then relayed back to the local client via the bridge.[3] mcp-remote can also be a valuable utility for testing newer transport protocols like Streamable HTTP with clients that do not yet natively support them.[23]

Configuration for using mcp-remote typically involves modifying the local client application's settings (e.g., the claude_desktop_config.json file for Claude Desktop) to point its MCP command to the mcp-remote executable. The mcp-remote tool itself is then configured with the URL of the actual target remote MCP server.[3]

The mcp-remote tool plays a crucial role as a compatibility layer, effectively bridging the gap between the earlier, often local-first, MCP ecosystem and the increasingly important remote-first ecosystem. Early adoption of MCP frequently centered on local tools accessed via stdio.[3] As remote MCP servers gained prominence and

demonstrated significant value [9], existing local clients required a mechanism to access these remote capabilities without necessitating a complete rewrite. mcp-remote provides this translation layer, making a remote server appear, from the perspective of the local client application, as if it were a local stdio-based server. This facilitates a smoother transition for users and developers, allowing them to leverage remote functionalities with their established local setups.

The existence and utility of mcp-remote also underscore the diversity found within MCP client implementations and their varying levels of native support for different transport protocols and advanced features. Not all MCP clients, especially older ones or those deeply embedded within specific applications, will immediately or simultaneously adopt all new MCP features, such as native Streamable HTTP support or direct remote connections incorporating complex OAuth 2.1 flows. In such scenarios, mcp-remote serves as a practical stopgap or a useful testing utility.[23] This implies that developers building MCP servers should be aware that their clients might be connecting through such intermediaries. However, the server implementation itself should ideally remain agnostic to the presence of a bridge tool, as long as the MCP protocol messages it receives are correctly formatted and adhere to the standard.

D. Client-Side Authentication: Handling the OAuth 2.1 Flow

When a client application attempts to connect to a secure remote MCP server that protects its tools or resources, it must typically engage in an OAuth 2.1 flow to obtain an access token.[3] This token serves as proof of authorization, allowing the client to make requests on behalf of the user.

The client-side OAuth 2.1 Authorization Code Grant flow generally involves the following steps:

1. **Initiate Authorization**: The client application constructs an authorization URL pointing to the MCP server's /authorize endpoint. This URL includes parameters such as the client ID, requested scopes, redirect URI, state parameter (for CSRF protection), and code challenge (for PKCE). The user is then redirected to this URL, typically by opening it in a web browser.
2. **User Authentication and Consent**: At the MCP server's authorization endpoint, the user authenticates themselves (e.g., by logging in) and grants consent for the client application to access the requested scopes (permissions).
3. **Receive Authorization Code**: If authentication and consent are successful, the MCP server's authorization server redirects the user's browser back to the client application's pre-registered callback URI. This redirection includes the temporary authorization code and the original state parameter in the query string.

4. **Exchange Code for Tokens**: The client application receives the authorization code from the callback. It then makes a secure, direct (server-to-server) POST request to the MCP server's /token endpoint. This request includes the authorization code, client ID, client secret (for confidential clients), redirect URI, and code verifier (for PKCE).
5. **Receive and Store Tokens**: If the code exchange is successful, the token endpoint responds with an access token and, typically, a refresh token. The client application must securely store these tokens. Access tokens are usually short-lived, while refresh tokens are longer-lived and can be used to obtain new access tokens without requiring the user to re-authenticate.
6. **Make Authenticated Requests**: The client includes the access token as a Bearer token in the Authorization HTTP header of all subsequent requests to the MCP server's protected tool or resource endpoints.[3]
7. **Token Refresh**: When an access token expires, the client uses the stored refresh token to request a new access token from the /token endpoint (using the refresh_token grant type).

Developers building client applications (or using tools like Anthropic's MCP Connector) are responsible for correctly implementing this OAuth flow, including the secure handling and refreshing of tokens.[26] The MCP Inspector tool can be used to guide users through the process of obtaining an access token for testing purposes with an MCP server.[26] Some SDKs, like the Cloudflare Agents SDK, aim to simplify this by integrating the OAuth flow directly, allowing agents to securely connect to remote MCP servers with less boilerplate code.[37]

While the modelcontextprotocol/python-sdk mentions client-side authentication and has an mcp.client.auth module, specific, comprehensive examples of a Python client fully implementing the OAuth 2.1 flow were not readily apparent in the browsed documentation snippets.[38] Community examples or leveraging third-party Python OAuth libraries may be necessary for developers building such clients from scratch.[5]

Implementing client-side OAuth 2.1 correctly, particularly the Authorization Code Flow with PKCE (which is the recommended standard for public clients like desktop or mobile applications that cannot securely store a client secret), is a non-trivial task critical for maintaining security. The level of abstraction provided by tools like Anthropic's MCP Connector or SDKs such as Cloudflare's Agents SDK is therefore highly valuable for developers, as these tools can manage much of this complexity. OAuth 2.1 involves multiple intricate steps, secure handling of sensitive credentials like codes and tokens, and careful management of state (e.g., the state parameter to prevent Cross-Site Request Forgery attacks). Errors in the implementation of these

flows can lead to serious security vulnerabilities, such as token leakage or unauthorized access. For developers who are building Python clients from the ground up using the modelcontextprotocol/python-sdk, the apparent scarcity of clear, robust examples or dedicated helper libraries for OAuth within the easily accessible SDK documentation represents a potential challenge. This underscores the importance of either seeking out well-vetted community solutions or relying on established Python OAuth client libraries (e.g., requests-oauthlib or Authlib [31]) to handle the complexities of the OAuth dance.

Beyond the initial acquisition of tokens, ongoing token management is a persistent responsibility for the client application. Access tokens are typically designed to be short-lived to limit the window of exposure if they are compromised.[33] Refresh tokens, which are longer-lived, are used to obtain new access tokens without requiring the user to go through the full authentication and consent process again. Client applications need to securely store these refresh tokens (e.g., using platform-specific secure storage mechanisms) and implement logic to automatically use them when an access token expires. Furthermore, clients should also be prepared to handle token revocation, for instance, if the user explicitly logs out of the application or revokes the granted consent through the authorization server's interface. These aspects of token lifecycle management add further to the complexity of client-side development, extending beyond just the initial OAuth exchange. Frameworks or libraries that assist with these token management tasks can significantly simplify development and improve the security posture of the client application.

IV. Practical Guide: An API-Agnostic Framework with the Python MCP SDK

This section provides a detailed, practical guide to designing and implementing an API-agnostic framework using the modelcontextprotocol/python-sdk. The goal is to create a system where various third-party APIs can be wrapped as MCP tools and exposed through a single, remotely accessible MCP server. This framework will leverage architectural best practices and concrete Python implementation strategies to achieve flexibility, extensibility, and security.

A. Introduction to the modelcontextprotocol/python-sdk and FastMCP

The modelcontextprotocol/python-sdk is the official Python Software Development Kit for the Model Context Protocol. It is a comprehensive library designed to facilitate the development of both MCP clients and MCP servers.[14] A key feature of the SDK is its support for the standard MCP transport protocols, including stdio (for local

communication), the legacy HTTP+SSE (Server-Sent Events) transport, and the modern Streamable HTTP transport for remote communication.[14]

Within this SDK, **FastMCP** stands out as a high-level abstraction specifically engineered to simplify and accelerate the development of MCP servers.[11] FastMCP achieves this simplification through several mechanisms:

- **Decorator-based Primitive Registration**: It allows developers to use Python decorators (@mcp.tool(), @mcp.resource(), @mcp.prompt()) to easily register functions as MCP tools, resources, or prompts. These decorators handle much of the boilerplate involved in defining these primitives according to the MCP specification.[14]
- **Automated Protocol Handling**: FastMCP takes care of underlying MCP connection management, ensures protocol compliance in message formatting and exchange, and handles the routing of incoming messages to the appropriate registered handlers.[19]
- **Simplified Server Execution**: A FastMCP server can often be run with a simple command like mcp.run(), which typically defaults to stdio transport but can be configured for HTTP transports to enable remote access.[17]
- **Integrated Authentication Support**: FastMCP supports the integration of OAuth 2.0/2.1 authentication by allowing developers to provide an auth_server_provider (an object that implements the OAuthServerProvider protocol defined by the SDK) and AuthSettings during its initialization. This enables the server to protect its tools and resources.[14]

While FastMCP offers a streamlined development experience, the SDK also provides access to lower-level server and client components. This allows for more direct control and customization if the abstractions provided by FastMCP are insufficient for highly specific or advanced use cases.[14]

Installation of the SDK is typically done using a modern package manager like uv (uv add "mcp[cli]") or pip (pip install "mcp[cli]"). The [cli] extra installs additional command-line tools, such as the MCP Inspector, which is useful for debugging.[14]

The FastMCP component significantly lowers the barrier to entry for creating basic MCP servers, making the protocol more accessible to a broad range of Python developers. Its declarative approach to defining tools, resources, and prompts, coupled with its handling of protocol intricacies, allows developers to focus more on the business logic of their integrations rather than on MCP boilerplate. This is excellent for rapid prototyping, building simple tools, and for developers new to MCP. However, for advanced production systems with unique requirements—such as deep

integration with existing web frameworks beyond basic Starlette/FastAPI (which FastMCP likely uses internally), implementation of highly customized transport configurations, or very specific OAuth 2.1 provider behaviors not easily mapped to the OAuthServerProvider protocol—it might become necessary to descend to the lower-level components of the SDK. The availability of these "low-level server implementations" [14] suggests that the SDK is designed with escape hatches for such scenarios. This flexibility, however, also implies a steeper learning curve if the conveniences of FastMCP prove insufficient for a particular use case. A key consideration for this report is to delineate when FastMCP is the appropriate choice versus when leveraging the lower-level APIs might be more advantageous.

The Python SDK, with FastMCP as a prominent feature, is evidently central to Anthropic's strategy for cultivating a vibrant and expanding MCP ecosystem. Given Python's dominance as a language in the fields of AI and machine learning, the robustness, feature set, and ease of use of this SDK directly influence the rate at which developers can build, deploy, and innovate with MCP-enabled services. Features like FastMCP, the provision of clear examples (though some gaps appear to exist for advanced authentication scenarios), and the embrace of modern packaging tools like uv all contribute to this strategic objective. The ongoing development, versioning, and feature enhancements of the SDK [73] are clear indicators of active maintenance and an evolutionary path responsive to community needs and the maturing MCP specification.

B. Designing the API-Agnostic Wrapper Architecture

The objective is to create a versatile framework where diverse third-party APIs can be seamlessly wrapped and exposed as standardized MCP tools. These tools will be served by a single, configurable MCP server, built using the modelcontextprotocol/python-sdk, and designed for remote access by various MCP clients.

Core Architectural Components:

1. **Plugin Manager**: This central component is responsible for the discovery, loading, and management of individual API wrapper plugins. A robust approach for plugin discovery is to utilize Python's standard importlib.metadata.entry_points mechanism.[42] This allows plugins to be developed as independent installable packages that register themselves under a predefined entry point group (e.g., my_mcp_framework.api_plugins).
2. **API Adapters (Plugins)**: These are individual Python modules or classes, each dedicated to interacting with a specific third-party API. They serve as the

translation layer, converting incoming MCP tool call requests into the API-specific format and transforming the API's responses back into a structure compatible with MCP. Each plugin will effectively implement the Adapter design pattern.

3. **Configuration System**: A comprehensive system for managing configurations is essential. This includes settings for the core framework itself (e.g., server host, port, logging levels) and for each individual plugin (e.g., API base URLs, authentication credentials, specific operational parameters). Pydantic is an ideal choice for defining these configurations in a typed and validated manner.[44] Secure management of secrets (API keys, tokens) is a critical sub-component of this system.
4. **Central MCP Server (based on FastMCP)**: This is the public-facing component of the framework. It will be built using FastMCP from the modelcontextprotocol/python-sdk. Its primary role is to expose the collective set of MCP tools defined by all loaded and active plugins. It will also handle the intricacies of the MCP protocol, transport layer (supporting both Streamable HTTP and HTTP+SSE for broad compatibility), session management (potentially with a persistent backend for scalability), and authentication (implementing OAuth 2.1 provider logic).
5. **Shared API Client/Session Manager (Optional but Recommended)**: To promote consistency and efficiency, the framework could provide a shared utility for managing HTTP client instances (e.g., using httpx.AsyncClient). This utility would handle common concerns like setting default timeouts, retry strategies, persistent connections (if applicable), and shared HTTP headers. Plugins could then receive an instance of this pre-configured client via dependency injection, rather than each plugin creating and managing its own client.[50]

Key Design Patterns to Employ:

The design of this framework will benefit from the application of several established software design patterns [[25] (search for design patterns), [56]]:

- **Adapter Pattern** [56]: Each API plugin will embody the Adapter pattern. It adapts the unique, often idiosyncratic, interface of a specific third-party API to a standardized internal interface that the framework's tool registration mechanism or the Plugin Manager expects. This is the cornerstone of achieving API agnosticism.
- **Strategy Pattern** [56]: If the framework needs to support multiple ways of interacting with a general *type* of service (e.g., different cloud storage providers, each with its own API but serving the same functional purpose), the Strategy pattern can be employed. Each plugin implementing access to a specific provider

for that service type would represent a concrete strategy. The framework could then allow selection of the desired strategy at runtime or via configuration.
- **Facade Pattern** [56]: The FastMCP server, in conjunction with the Plugin Manager, can be viewed as implementing the Facade pattern. It provides a simplified, unified MCP interface (a collection of tools) to the potentially complex underlying system of diverse, individually managed API integrations.
- **Plugin Architecture** [42]: This is the core architectural style enabling extensibility. It allows new API integrations (plugins) to be added to the framework without requiring modifications to the core server codebase.
- **Dependency Injection** [50]: This pattern will be highly beneficial for providing shared resources—such as configured HTTP client instances, logging services, or access to the central configuration object—to the individual plugins in a decoupled and testable manner.

The choice of plugin discovery mechanism carries implications for both deployment flexibility and the ease of plugin development. Utilizing importlib.metadata.entry_points [42] aligns with standard Python packaging practices. This method is robust and integrates seamlessly with Python's installation and packaging tools (like pip and uv). However, it typically requires that plugins be developed and installed as distinct Python packages. An alternative, such as scanning a designated directory for plugin modules, might offer a simpler setup for local development or less formal plugin structures. However, directory scanning can be less secure if the source of plugins is not strictly controlled and can become more challenging to manage in complex deployment scenarios with multiple environments or plugin versions. For a framework intended to be robust, shareable, and potentially used by a wider community, the entry points mechanism is generally the preferred approach due to its explicitness and integration with the Python packaging ecosystem.

A well-defined and stable interface for the API adapter plugins themselves is paramount for the long-term success, maintainability, and extensibility of the framework. Each plugin must conform to a clear contract that dictates how it registers its MCP tools with the central server, how it receives its specific configuration settings (including secrets), how it should handle incoming tool execution requests, and the format in which it must return results or report errors. This plugin interface should be meticulously documented and, where feasible, enforced programmatically (e.g., by requiring plugins to inherit from a common Abstract Base Class that defines the necessary methods). A thoughtfully designed and stable plugin interface minimizes the likelihood of breaking changes as the core framework evolves. It also significantly

lowers the barrier for third-party developers or other teams within an organization to contribute new plugins, thereby enriching the ecosystem of APIs accessible through the framework.

The following table summarizes the application of these design patterns:

Table IV.B.1: Application of Design Patterns in the API-Agnostic MCP Framework

Design Pattern	Role in Framework	Key Benefit(s)	Relevant Snippets
Adapter	Each API plugin adapts a third-party API to a common internal interface for MCP tool registration and execution.	Enables API agnosticism; decouples framework core from specific API implementations.	56
Strategy	Allows interchangeable concrete API implementations (plugins) for a general service type (e.g., different weather APIs).	Provides flexibility in choosing service providers; configurable behavior.	56
Facade	The FastMCP server and Plugin Manager provide a simplified MCP tool interface to complex underlying API integrations.	Simplifies client interaction; hides internal complexity of managing multiple APIs.	56
Plugin Architecture	Core mechanism for discovering and loading API adapter plugins dynamically.	Extensibility without core modification; modularity; promotes independent development of API integrations.	42
Dependency Injection	Provides shared resources (e.g., HTTP clients, configuration,	Decoupling; improved testability of plugins; centralized	50

	loggers) to plugins.	management of shared resources.	

This explicit mapping of design patterns to their roles within the framework aids in understanding the architectural rationale, promotes sound design principles, and offers a clear conceptual model of how the framework's components are intended to interact.

C. Implementing a Dynamic Plugin System for API Connectors

The dynamism of the framework hinges on its ability to discover, load, and integrate API connector plugins at runtime. This allows the MCP server to expose a growing set of tools without requiring code changes to its core.

Plugin Discovery:
The recommended method for plugin discovery is Python's entry points mechanism, specifically importlib.metadata.entry_points.42 Plugins (developed as separate installable packages) would register themselves under a well-defined group name, for example, my_mcp_framework.api_plugins. Each entry point within this group would typically point to the main class of an API adapter plugin.

Python

```
# Conceptual Plugin Manager Snippet (Discovery)
import importlib.metadata

PLUGIN_ENTRY_POINT_GROUP = "my_mcp_framework.api_plugins"

class PluginManager:
    def __init__(self):
        self.plugins = {}

    def discover_plugins(self):
        discovered_entry_points =
importlib.metadata.entry_points(group=PLUGIN_ENTRY_POINT_GROUP)
        for entry_point in discovered_entry_points:
            try:
                plugin_class = entry_point.load()
                # Further initialization and registration would happen here
                print(f"Discovered plugin '{entry_point.name}': {plugin_class}")
```

```
            # Store or instantiate plugin_class
        except Exception as e:
            print(f"Error loading plugin '{entry_point.name}': {e}")
```

Plugin Loading and Initialization:
The Plugin Manager iterates through the discovered entry points. For each valid entry point, it loads the specified module and instantiates the plugin class. During this initialization phase, each plugin instance must receive its specific configuration settings (retrieved from the central configuration system) and any shared resources (such as a pre-configured httpx.AsyncClient instance or a logger) that the framework provides. Dependency injection is an effective pattern for supplying these to the plugin.
Tool Registration by Plugins:
Once a plugin is loaded and initialized, it becomes responsible for registering its MCP tools with the central FastMCP server instance. This registration can occur in a couple of ways:

1. **Direct Decoration (if feasible):** If plugins are loaded early enough and have access to the FastMCP instance, they might be able to use the @mcp.tool() decorator directly on their methods. This is the simplest approach from a plugin developer's perspective.
2. **Programmatic Registration:** More commonly, the plugin will implement a specific method (e.g., register_tools(mcp_server_instance)) defined by the plugin interface. The Plugin Manager calls this method, passing the FastMCP instance, and the plugin then programmatically calls a registration function on the FastMCP object for each tool it wants to expose.

Regardless of the method, the plugin must provide all necessary information for each tool: its unique name (within the server's namespace), a clear and concise description (this is crucial, as LLMs rely on it for tool discovery and selection), an input schema (preferably defined using Pydantic models, which FastMCP can use for automatic request validation and documentation), and the actual handler function that will be executed when the tool is called. This handler function contains the logic to interact with the underlying third-party API.

Plugin Lifecycle Management (Simplified):
For this framework, a basic lifecycle management approach might suffice, focusing primarily on load-time initialization. When the MCP server starts, the Plugin Manager discovers and loads all available plugins. Each plugin is initialized and registers its tools. These tools then remain available for the lifetime of the server process. More complex lifecycle operations, such as dynamically enabling/disabling plugins at runtime or unloading them without restarting the server, could be considered as future enhancements. If such features were required, a hook-based system 75 could be designed, allowing plugins to respond to events like on_enable, on_disable, or on_unload.

The dynamic loading of plugins is a powerful feature. It allows the framework to be extended with new API integrations simply by installing a new plugin package and providing its configuration. The core MCP server code remains unchanged, promoting modularity, maintainability, and enabling different teams or even a community to contribute API connectors independently.

However, robust error handling during the plugin discovery, loading, and tool execution phases is critically important. Since plugins are essentially external pieces of code, they can introduce errors. A faulty plugin—due to import errors, misconfiguration, bugs in its tool handlers, or issues with the third-party API it wraps—should not be allowed to bring down the entire MCP server. The Plugin Manager and the FastMCP server must implement comprehensive error handling mechanisms. Exceptions originating from plugins should be caught, logged appropriately (providing enough context for debugging), and handled gracefully. For example, if a plugin fails to load correctly, it might be skipped, and its tools would not be registered, but the server should continue to operate with other successfully loaded plugins. Similarly, if a tool call within a plugin fails, the error should be reported back to the MCP client according to the JSON-RPC error specification, without crashing the server process.

D. Managing Configuration and Secrets Securely

A robust and secure configuration system is vital for an API-agnostic framework that relies on numerous plugins, each potentially requiring its own set of credentials and operational parameters. This system must balance ease of use for developers with stringent security practices, especially concerning API keys and other secrets.

Typed Configuration with Pydantic:
Pydantic 44 is an excellent choice for managing configurations due to its strong data validation, type hinting capabilities, and ability to create clear, structured settings models.

- **Global and Plugin-Specific Models**: Define Pydantic models for:
  - Global framework settings (e.g., server host/port, logging configuration, default timeouts).
  - A base configuration model for all plugins (if there are common settings).
  - Specific configuration models for each individual API plugin, inheriting from the base if applicable, to define its unique parameters (e.g., api_key, base_url, user_specific_settings).
- **Loading Configuration**: Configuration data can be loaded from various sources, such as YAML, TOML, or JSON files. Libraries like PydaConf [49] demonstrate patterns for loading Pydantic models from such files and can even offer pluggable secret injection. Environment variables are another common source, especially for

overriding settings in different deployment environments.

- **Validation and Defaults**: Pydantic automatically validates loaded configurations against the defined models, ensuring that all required fields are present and that values conform to the specified types and constraints (e.g., min_length for strings, gt for numbers [46]). Default values can be specified in the Pydantic models, simplifying configuration for common cases.[46]
- **Strictness**: Pydantic's ConfigDict (or the older Config class) allows for settings like extra='forbid' to prevent unknown or misspelled configuration keys from being silently ignored, which can help catch errors early.[48]

Python

```
# Conceptual Pydantic Configuration Snippet
from pydantic import BaseModel, SecretStr, HttpUrl
from typing import Dict, Optional

class BasePluginConfig(BaseModel):
    enabled: bool = True
    timeout_seconds: int = 30

class WeatherApiPluginConfig(BasePluginConfig):
    api_key: SecretStr # Pydantic's SecretStr obscures the value in logs/repr
    base_url: HttpUrl = "https://api.weatherprovider.com/v1" # type: ignore

class TaskApiPluginConfig(BasePluginConfig):
    api_token: SecretStr
    user_id: Optional[str] = None

class GlobalFrameworkConfig(BaseModel):
    server_host: str = "0.0.0.0"
    server_port: int = 8000
    log_level: str = "INFO"
    plugins: Dict # Holds specific plugin configs, e.g., {"weather": WeatherApiPluginConfig(...)}

# Configuration loading logic would populate GlobalFrameworkConfig
# For example, from a YAML file:
# global:
#   server_host: "127.0.0.1"
```

```
# plugins:
#   weather:
#     api_key: "env_var:WEATHER_API_KEY" # Placeholder for env var or secrets manager
#     enabled: True
#   task_manager:
#     api_token: "secret:task_manager/api_token" # Placeholder for secrets manager
#     enabled: False
```

Secrets Management Best Practices:
The secure handling of API keys, tokens, and other sensitive credentials is non-negotiable.64

- **Never Hardcode Secrets**: API keys or any sensitive credentials must **never** be embedded directly in source code or committed to version control systems (Git, etc.).[64] Configuration files containing plaintext secrets should also be excluded from version control (e.g., by adding them to .gitignore).[64]
- **Use Environment Variables**: Storing secrets in environment variables is a common practice, especially for separating configuration from code.[64] For local development, libraries like python-dotenv can load these variables from a .env file (which itself should be in .gitignore).[67]
- **Dedicated Secrets Managers (Production)**: For staging and production environments, using a dedicated secrets management service is strongly recommended. Options include HashiCorp Vault, AWS Secrets Manager, Google Cloud Secret Manager, Azure Key Vault, or commercial solutions like Keeper Secrets Manager (which provides a Python SDK [66]). These tools offer secure, encrypted storage, fine-grained access control, audit logging, and often automated secret rotation capabilities.
- **Configuration Management Tools**: Tools like Ansible, Chef, or Puppet can be used to securely deploy configurations, including injecting secrets retrieved from a secrets manager into the application environment at deployment time.[64]
- **Principle of Least Privilege**: Each API key or token used by a plugin should be granted only the minimum necessary permissions required for that plugin's specific tasks. Avoid using overly permissive or master keys.[64]
- **Regular Key Rotation**: Implement a schedule for regularly rotating API keys (e.g., every 30-90 days, depending on sensitivity). Automated rotation, if supported by the secrets manager and the third-party API, is ideal.[64]
- **Access Control and Monitoring**: If the third-party APIs support it, utilize features like IP address whitelisting (restricting key usage to specific server IPs) and rate limiting to further secure API keys and detect anomalous activity.[64] Comprehensive logging of API calls made by plugins can also aid in auditing and incident response.

Framework Approach to Configuration and Secrets:

The framework's central configuration loader should be responsible for reading the main configuration file(s) and environment variables. It will then parse and validate this configuration using the defined Pydantic models. For secrets, the configuration system should ideally integrate with a secrets management backend. Instead of plugins directly accessing environment variables or files for their secrets, they should receive their fully resolved, typed configuration objects (with secrets populated) from the framework, potentially via dependency injection. This centralizes control over configuration and secrets, ensures consistency, and makes it easier to enforce security best practices across all plugins. For example, a configuration value for an API key might be specified in a config file as a placeholder like "env:MY_PLUGIN_API_KEY" or "vault:path/to/secret". The framework's config loader would then resolve this placeholder by fetching the actual secret from the environment or the secrets manager before injecting it into the plugin's Pydantic config model.
A unified configuration system that leverages Pydantic for structure, validation, and type safety, when combined with a pluggable or abstracted secrets management backend, offers a powerful combination of developer-friendliness and production-grade security. Pydantic makes the definition and validation of potentially complex configurations straightforward and less error-prone.[44] Different deployment environments (development, staging, production) invariably have distinct requirements for how secrets are managed (e.g., a simple .env file is often sufficient and convenient for local development, while a robust secrets manager like HashiCorp Vault is essential for production). A configuration system that can abstract the actual source of secrets—perhaps using a library like PydaConf [49], which supports pluggable secret injectors, or by designing a similar abstraction layer within the framework—allows plugins to be written agnostically of the specific secret storage mechanism being used in a given environment. The core framework would handle the loading of the Pydantic configuration models and the resolution of any secret values before the typed configuration object is passed to the respective plugin.

Managing secrets for a dynamically growing number of plugins, where each plugin might require multiple API keys or credentials for different third-party services, presents a significant operational security challenge. This complexity must be proactively addressed in the framework's design. As the number of integrated plugins increases, so does the total number of secrets that need to be managed, which in turn expands the potential attack surface and complicates tasks like regular rotation and auditing. Therefore, the framework's strategy for secrets management must be inherently scalable and capable of consistently enforcing security best practices, such as the principle of least privilege for each plugin's keys. This consideration strongly argues for designing the framework with direct integration capabilities for centralized secrets management solutions, especially in production deployments.

E. Step-by-Step Implementation using modelcontextprotocol/python-sdk

This subsection outlines the core implementation steps for the API-agnostic MCP server framework and a corresponding Python client, leveraging the modelcontextprotocol/python-sdk. Due to the complexity and length of a full implementation, conceptual Python snippets will be provided to illustrate key components and integration points.

1. Building the FastMCP Server Core

The server will be built around FastMCP, integrating the plugin system, OAuth 2.1 provider, multi-transport support, and session management.

- Core Setup with FastMCP:
  The heart of the server is an instance of FastMCP. This instance will be responsible for managing MCP primitives (tools, resources, prompts) registered by the plugins.

```
Python
# server_core.py
from mcp.server.fastmcp import FastMCP
from mcp.server.auth import AuthSettings, OAuthServerProvider # Assuming OAuthServerProvider is a protocol
# from.oauth_provider import MyCustomOAuthProvider # To be implemented
# from.plugin_manager import PluginManager
# from.config import load_global_config, GlobalFrameworkConfig

# global_config: GlobalFrameworkConfig = load_global_config()

# oauth_provider = MyCustomOAuthProvider(config=global_config.oauth_settings)
# auth_settings = AuthSettings(
#     issuer_url=str(global_config.oauth_settings.issuer_url),
#     #... other AuthSettings based on global_config
# )

# mcp_server = FastMCP(
#     name="API_Agnostic_MCP_Framework_Server",
#     version="1.0.0",
#     # auth_server_provider=oauth_provider, # Enable once MyCustomOAuthProvider is implemented
#     # auth=auth_settings, # Enable once MyCustomOAuthProvider is implemented
#     # transport_settings relevant for Streamable HTTP / SSE might be needed
# )

# plugin_manager = PluginManager(mcp_server_instance=mcp_server, config=global_config.plugins)
```

```
# plugin_manager.load_and_register_plugins()

# app = mcp_server # FastMCP instance is an ASGI app
```

FastMCP itself is an ASGI application, which can be run with an ASGI server like Uvicorn. If custom non-MCP routes are needed (e.g., for the OAuth endpoints), FastMCP can often be mounted as a sub-application within a larger FastAPI or Starlette application.

- Implementing OAuthServerProvider for OAuth 2.1:
  This is a critical and complex part. The SDK provides an OAuthServerProvider protocol 14, which your custom class must implement. This class will handle the logic for OAuth 2.1 endpoints (/.well-known/..., /authorize, /token, etc.).3
  Given the lack of detailed official examples for OAuthServerProvider 77, a robust implementation would likely involve using a dedicated OAuth library like Authlib 31 internally to manage the complexities of token issuance, validation, client registration, and grant flows.

```
Python
# oauth_provider.py (Conceptual Structure)
from mcp.server.auth import (
    OAuthServerProvider,
    # Import necessary request/response types from mcp.server.auth.types if available
    # or define them based on OAuth 2.1 spec and MCP requirements.
    # For example: AuthorizeRequest, TokenRequest, ClientInfo, AuthCode, AccessToken
)
# from authlib.integrations.starlette_oauth2 import AuthorizationServer # Example using Authlib
# from starlette.requests import Request
# from starlette.responses import HTMLResponse, RedirectResponse

class MyCustomOAuthProvider(OAuthServerProvider):
    def __init__(self, config): # config for database connections, secrets etc.
        # self.auth_server = self._init_authlib_server() # Example
        pass

    # --- Methods required by OAuthServerProvider protocol ---
    # These method signatures are illustrative and need to be based on the
    # actual OAuthServerProvider protocol definition in the SDK.

    async def get_client_info(self, client_id: str) -> # ClientInfo | None:
        # Retrieve client details from storage
        # Return None if client_id is invalid
```

```
        pass

    async def save_authorization_code(self, code: str, request: # AuthorizeRequest) -> None:
        # Store authorization code with associated client_id, redirect_uri, scopes, user_id,
pkce_challenge
        pass

    async def authenticate_user_for_authorize(self, request: # StarletteRequest) -> # User | None:
        # Logic to authenticate the resource owner (user)
        # This might involve session checks, redirecting to a login page, etc.
        # Returns user object or None if not authenticated.
        # This is highly dependent on your application's user authentication system.
        pass

    async def confirm_authorization_consent(self, request: # StarletteRequest, user: # User) ->
bool:
        # Logic to confirm user consent for the requested scopes.
        # Might involve rendering a consent page.
        return True # Placeholder

    async def create_access_token(self, token_request: # TokenRequest, client: # ClientInfo) -> #
AccessToken:
        # Validate authorization_code or refresh_token, client credentials
        # Issue access token (and optionally refresh token)
        # Store token details
        pass

    async def validate_access_token(self, token: str, required_scopes: list[str] | None) -> #
AuthenticatedPrincipal | None:
        # Validate the bearer token from an incoming MCP tool request
        # Check signature, expiry, scopes
        # Return an object representing the authenticated principal (user/client) or None
        pass

    async def revoke_token(self, token: str, token_type_hint: str | None, client: # ClientInfo) -> None:
        # Invalidate the given token (access or refresh)
        pass

    # --- Helper methods for FastAPI/Starlette routes ---
    # These would be called by your /authorize, /token, etc. HTTP endpoints

    # async def handle_authorize_endpoint(self, request: Request) -> Response:
        # 1. Authenticate user (call self.authenticate_user_for_authorize)
        # 2. Confirm consent (call self.confirm_authorization_consent)
        # 3. If all good, generate and save auth code (call self.save_authorization_code)
        # 4. Redirect to client's redirect_uri with code and state
        # Use Authlib's server.create_authorization_response(grant_user=user) for this
```

```
        # return HTMLResponse("Authorize endpoint logic here") # Placeholder

    # async def handle_token_endpoint(self, request: Request) -> Response:
        # Use Authlib's server.create_token_response() for this
        # This will internally call your registered grant types (e.g., authorization_code, refresh_token)
        # which in turn would call methods like self.create_access_token
        # return HTMLResponse("Token endpoint logic here") # Placeholder

    # async def handle_revocation_endpoint(self, request: Request) -> Response:
        # Use Authlib's server.create_revocation_response()
        # return HTMLResponse("Revocation endpoint logic here") # Placeholder

    #... methods for.well-known endpoints...
```

The FastMCP instance would then be initialized with auth_server_provider=MyCustomOAuthProvider(...) and appropriate AuthSettings. The HTTP routes for /authorize, /token, etc., would need to be added to the main ASGI application, and these route handlers would call the respective methods on the MyCustomOAuthProvider instance.

- Supporting Multiple Transports (Streamable HTTP & HTTP+SSE):
  A single FastMCP server instance should ideally handle both modern Streamable HTTP clients (at /mcp) and legacy HTTP+SSE clients (at /sse and /messages). The modelcontextprotocol/python-sdk documentation and examples 23 are crucial here.
  If FastMCP does not natively support routing for both transport types to the same toolset when mounted as a single ASGI app, an approach involving ASGI routing middleware might be necessary. For instance, using Starlette's routing capabilities, one could define routes that direct traffic based on the path:
  - Requests to /mcp (for Streamable HTTP) are handled directly by the FastMCP ASGI app.
  - Requests to /sse and /messages (for HTTP+SSE) might need to be handled by a separate SSEServerTransport instance from mcp.server.sse [3], which is then connected to the *same underlying MCP server logic/tool registry* as the Streamable HTTP part. This ensures tool consistency. The Cloudflare example [23] shows an McpAgent class with distinct serveSSE('/sse') and serve('/mcp') methods, suggesting that SDKs can provide abstractions for this. The Python SDK's FastMCP or lower-level components might offer similar configuration options. Without definitive examples from the SDK, this remains an area requiring careful implementation to ensure the same set of registered tools (from plugins) is accessible via both transport mechanisms.
- Integrating a Persistent Session Store (e.g., Redis):

FastMCP's default session management is likely in-memory, which is unsuitable for scalable, distributed deployments.28 To support persistence and sharing of session state across multiple server instances, an external store like Redis is needed.3

The key question is whether FastMCP provides direct hooks to plug in a custom session backend for its own internal MCP protocol session objects, or if session persistence is primarily the responsibility of the tool implementations themselves if they require state across calls.

- If FastMCP manages MCP-level session objects (e.g., tracking client connections, subscriptions), and these are stored in memory by default, then overriding this behavior would require specific extension points in FastMCP's session handling.[83]
- If FastMCP is largely stateless regarding tool calls once a connection is established (or if its session objects are lightweight and primarily for transport management), then stateful tools developed as plugins would need to manage their own state using an external store. The Context object (ctx: Context) passed to tool handlers [14] might be a place to provide access to a shared session store interface.

The mcp-CodeAnalysis server [29] implements its own session store (Redis/in-memory) and a "stateful tool helper," suggesting that tool-level state management is a common pattern. The framework should provide a SessionStore protocol and concrete implementations (e.g., RedisSessionStore, InMemorySessionStore), making an instance available to plugins via dependency injection.Python

```
# session_store.py (Conceptual)
from abc import ABC, abstractmethod
import redis.asyncio as redis # Example using redis-py

class SessionStore(ABC):
    @abstractmethod
    async def get_session_data(self, session_id: str, key: str) -> any:
        pass

    @abstractmethod
    async def set_session_data(self, session_id: str, key: str, value: any, expiry_seconds: int = 3600) ->
None:
        pass

    @abstractmethod
    async def delete_session_key(self, session_id: str, key: str) -> None:
        pass

class RedisSessionStore(SessionStore):
    def __init__(self, redis_url: str):
```

```
        self.redis_client = redis.from_url(redis_url)

    async def get_session_data(self, session_id: str, key: str) -> any:
        # Implement logic to get data from Redis, e.g., using HGET or GET
        # Remember to deserialize if necessary
        pass

    #... other methods...
```

The FastMCP tool handlers would then use this injected session store.

2. Developing the API-Agnostic Python Client

The client application will connect to the remote MCP server, handle OAuth 2.1 for authentication, discover available tools, and invoke them.

- Core Client Setup:
  The client will use components from mcp.client provided by the modelcontextprotocol/python-sdk.14
  The OpenAI Agents SDK provides classes like MCPServerSse and MCPServerStreamableHttp for connecting to remote servers 71, which serve as a good conceptual model for how the Anthropic Python SDK client might be structured for remote connections.

  ```
  Python
  # mcp_api_client.py (Conceptual)
  from mcp.client.fast_client import FastMCPClient # Assuming such a high-level client exists
  # Or using lower-level components:
  # from mcp.client.sse import SSEClientTransport
  # from mcp.client.streamable_http import StreamableHTTPClientTransport
  # from mcp.client.session import ClientSession
  # from.oauth_client_handler import OAuthClientHandler # To be implemented

  class MyRemoteMCPClient:
      def __init__(self, server_base_url: str, client_id: str, client_secret: str, redirect_uri: str):
          self.server_base_url = server_base_url
          # self.oauth_handler = OAuthClientHandler(
          #     server_base_url=server_base_url,
          #     client_id=client_id,
          #     client_secret=client_secret, # For confidential clients
          #     redirect_uri=redirect_uri,
          #     # token_endpoint=f"{server_base_url}/token", # etc.
          # )
          self.mcp_session = None # Will be ClientSession or similar
  ```

```
    async def connect(self):
        # access_token = await self.oauth_handler.get_access_token() # Handles full OAuth flow
        # if not access_token:
        #     raise Exception("Failed to obtain access token")

        # Determine transport (Streamable HTTP preferred, fallback to SSE)
        # For Streamable HTTP:
        # transport = StreamableHTTPClientTransport(url=f"{self.server_base_url}/mcp")
        # For SSE:
        # transport = SSEClientTransport(
        #     sse_url=f"{self.server_base_url}/sse",
        #     post_url=f"{self.server_base_url}/messages"
        # )

        # self.mcp_session = ClientSession(transport=transport)
        # await self.mcp_session.initialize(
        #     client_info={"name": "MyAPIClient", "version": "1.0"},
        #     # headers={"Authorization": f"Bearer {access_token}"} # Pass token
        # )
        print("Connected and initialized with MCP server.")

    async def list_tools(self) -> list:
        # if not self.mcp_session:
        #     raise Exception("Not connected")
        # return await self.mcp_session.list_tools()
        pass # Placeholder

    async def call_tool(self, tool_name: str, params: dict) -> any:
        # if not self.mcp_session:
        #     raise Exception("Not connected")
        # return await self.mcp_session.call_tool(name=tool_name, params=params)
        pass # Placeholder
```

- Implementing OAuth 2.1 Client-Side Authentication:
  This is crucial for interacting with the secured MCP server. The client needs to manage the full Authorization Code Grant flow.3
  The mcp.client.auth module in the Python SDK might provide helpers.38 If not, libraries like requests-oauthlib or Authlib's client components 31 are essential for

robustly handling:
    - Constructing the authorization URL and initiating the redirect (e.g., by printing the URL for the user to open in a browser for a CLI client, or handling the redirect in a web app client).
    - Running a local temporary HTTP server to handle the callback from the authorization server to capture the authorization code and state.
    - Exchanging the authorization code for an access token and refresh token at the MCP server's token endpoint.
    - Securely storing tokens (in-memory for CLI session, or using OS keychain/encrypted storage for longer-lived clients).
    - Automatically adding the access token as a Bearer token to the Authorization header for all MCP requests.
    - Implementing logic to use the refresh token to obtain a new access token when the current one expires.
- Dynamically Invoking Tools:
  Once connected and authenticated, the client interacts with the MCP server:
    1. **List Tools**: The client calls the MCP tools/list method (or its SDK equivalent, e.g., await server.list_tools() [71]) to retrieve the list of tools currently available from the server. These tools are dynamically registered by the server's plugins.
    2. **Tool Selection**: The client application (or an LLM it's interacting with) uses this list to select an appropriate tool.
    3. **Call Tool**: To invoke the selected tool, the client sends an MCP tools/call message (or SDK equivalent, e.g., await server.call_tool(name=..., params=...) [71] or await client_session.call_tool(...) [21]) containing the tool's name and the required parameters.
    4. **Process Result**: The client receives the tool's result (or an error) from the server and processes it accordingly.

3. Illustrative Example: Wrapping Two Hypothetical APIs as Plugins

Let's outline how two distinct APIs could be wrapped as plugins for this framework.

- **Plugin A: Weather API Plugin**
    - **Configuration (WeatherApiPluginConfig from Pydantic example):**
        - api_key: SecretStr
        - base_url: HttpUrl (e.g., "https://api.openweathermap.org/data/2.5")
        - enabled: bool
    - **MCP Tool Definition:**
        - Name: get_current_weather

- Description: "Fetches the current weather conditions for a specified location."
- Input Schema (Pydantic model): class WeatherInput(BaseModel): location: str
- Output Schema (Pydantic model): class WeatherOutput(BaseModel): temperature: float description: str humidity: int

- **Plugin Adapter Logic (weather_plugin.py):**

```
Python
# weather_plugin.py (Conceptual)
from pydantic import BaseModel
# from..shared_http_client import SharedHttpClient # Injected
# from..config_models import WeatherApiPluginConfig # Injected config
# from mcp.server.fastmcp import FastMCP # Passed for registration

class WeatherInput(BaseModel):
    location: str

class WeatherOutput(BaseModel):
    temperature: float
    description: str
    humidity: int

class WeatherApiPlugin:
    def __init__(self, config: # WeatherApiPluginConfig, http_client: # SharedHttpClient):
        self.config = config
        self.http_client = http_client

    def register_tools(self, mcp_server: # FastMCP):
        # Option 1: If FastMCP allows programmatic registration
        # mcp_server.register_tool(
        #     name="get_current_weather",
        #     description="Fetches the current weather conditions for a specified location.",
        #     input_model=WeatherInput,
        #     output_model=WeatherOutput, # Or handler returns Pydantic model
        #     handler=self.get_current_weather_handler
        # )
        # Option 2: If using decorators, this class might need to be structured differently
        # or decorators applied to its methods if FastMCP instance is accessible at definition
time.
        pass # Placeholder for actual registration logic

    async def get_current_weather_handler(self, params: WeatherInput) -> WeatherOutput:
        # actual_api_url =
f"{self.config.base_url}/weather?q={params.location}&appid={self.config.api_key.get_secret_v
```

```
alue()}"
        # response = await self.http_client.get(actual_api_url)
        # response.raise_for_status() # Raise exception for bad status codes
        # data = response.json()
        # return WeatherOutput(
        #     temperature=data['main']['temp'],
        #     description=data['weather']['description'],
        #     humidity=data['main']['humidity']
        # )
        return WeatherOutput(temperature=25.0, description="Sunny", humidity=60) #
Placeholder
```

    - **Entry Point (in plugin's pyproject.toml):**
      Ini, TOML
      ```
      [project.entry-points."my_mcp_framework.api_plugins"]
      weather = "my_weather_plugin_package.weather_plugin:WeatherApiPlugin"
      ```

- **Plugin B: Task Management API Plugin**
  - **Configuration (TaskApiPluginConfig):**
    - api_token: SecretStr
    - base_url: HttpUrl (e.g., "https://api.taskmanager.com/v2")
    - user_id: Optional[str]
    - enabled: bool
  - **MCP Tool Definitions:**
    1. create_task: "Creates a new task."
       - Input: class CreateTaskInput(BaseModel): title: str description: Optional[str] = None
       - Output: class TaskOutput(BaseModel): id: str title: str status: str
    2. list_tasks: "Lists tasks, optionally filtered by status."
       - Input: class ListTasksInput(BaseModel): status: Optional[str] = "open"
       - Output: list
  - **Plugin Adapter Logic (task_plugin.py):** Similar structure to WeatherApiPlugin, with methods for create_task_handler and list_tasks_handler that make appropriate authenticated POST/GET requests to the task management API.
  - **Entry Point:** Registered similarly in its own package.

The Plugin Manager in the server core would discover these plugins via their entry points, instantiate them with their respective configurations (secrets resolved), and call their register_tools method, passing the central FastMCP instance. The tools would then be available via the MCP server. The Python client, after connecting and authenticating, would be able to list get_current_weather, create_task, and list_tasks

as available tools and invoke them.

The practical implementation of OAuth 2.1, for both the server-side OAuthServerProvider and the client-side authentication handler, represents the most intricate aspect of this framework, primarily due to the apparent lack of comprehensive, easily accessible, and complete examples within the modelcontextprotocol/python-sdk documentation snippets for these specific components. While the SDK clearly provides interfaces and mentions modules like mcp.server.auth and mcp.client.auth [14], detailed "how-to" guides or full reference implementations for these seem to be located in less accessible parts of the GitHub repository (e.g., specific example directories that were not browsable) or are presented at a high level. This implies that the report must, to some extent, extrapolate the expected behavior and structure from the OAuth 2.1 standard itself [3] and from the hints provided by the SDK's AuthSettings parameters and the conceptual use of OAuthServerProvider. It may be necessary to strongly recommend or demonstrate the integration of well-established external Python OAuth libraries, such as Authlib, to manage the significant underlying complexities of the OAuth protocol within both the server provider and client implementations. This is a notable practical challenge for developers aiming to build secure, remote MCP services and clients using the current SDK.

Furthermore, the interplay between the abstractions offered by FastMCP and the potential need for custom behaviors—such as sophisticated multi-transport support or the integration of custom session stores—will largely define the actual implementation complexity of the server. FastMCP undoubtedly simplifies many common scenarios. However, if it does not natively support, for instance, the elegant routing of different transport protocols (Streamable HTTP and HTTP+SSE) to the same underlying toolset when operating as a single, unified server, developers might find themselves needing to work at a lower ASGI level to compose these behaviors. This could involve bypassing some of FastMCP's conveniences. Similarly, if FastMCP's internal session management mechanisms are not designed to be easily pluggable with external stores like Redis, then developers of stateful tools will need to manage their own state persistence independently of any MCP-level session objects. This is a feasible approach but one that needs to be clearly understood and architected.

Despite these complexities, the true power of the proposed API-agnostic framework emerges from the synergistic combination of dynamic plugin loading, typed configuration management with Pydantic, and the standardized exposure of all integrated functionalities as MCP tools. Dynamic loading ensures that new APIs can be incorporated with relative ease, often just by adding a new plugin package. Typed

Pydantic configurations enhance the robustness and safety of plugin integration by ensuring that plugins receive valid and expected settings. Finally, exposing all functionalities via MCP tools means that any MCP-compatible client—including sophisticated LLM agents—can then leverage this diverse array of backend APIs through a single, consistent, and discoverable interface. This creates a powerful abstraction layer that has the potential to significantly accelerate the development and enhance the capabilities of AI agents.

V. Conclusion and Future Outlook for MCP

The Model Context Protocol is rapidly establishing itself as a crucial enabler for the next generation of AI applications, particularly those involving sophisticated agentic behaviors. Its core value lies in standardizing the way LLMs discover, interact with, and leverage external tools, data sources, and services. This report has provided a deep dive into MCP's fundamentals, the intricacies of building secure remote MCP servers, mechanisms for client connectivity over the web, and a practical architectural guide for developing an API-agnostic framework using the modelcontextprotocol/python-sdk.

A. Summary of MCP's Significance and Implementation Strategies

MCP's significance stems from its potential to solve the N x M integration problem, fostering an interoperable ecosystem where AI agents can seamlessly connect to a vast array of capabilities.[1] By providing a "USB-C port for AI" [2], it simplifies development, enhances LLM autonomy, and offers a path towards more standardized governance and security.

Building robust remote MCP servers requires careful consideration of several key areas:

- **Transport Protocols**: While legacy HTTP+SSE is still prevalent, the modern Streamable HTTP protocol offers a simpler, more efficient single-endpoint approach, better suited for contemporary web architectures, including serverless deployments.[3] Servers aiming for broad compatibility should consider supporting both.
- **Authentication**: OAuth 2.1 is the standard for securing remote MCP servers, enabling proper user authentication and client authorization.[3] Implementing a full OAuth 2.1 provider is complex; the modelcontextprotocol/python-sdk provides an OAuthServerProvider interface, but leveraging dedicated OAuth libraries like Authlib within this implementation is advisable for robustness and security.[14]
- **Session Management**: Stateful interactions are common in MCP. For scalable

and resilient deployments, especially those with multiple server instances, session state must be managed externally using persistent stores like Redis, moving beyond simple in-memory solutions.[3]
- **Security**: Beyond authentication, servers must be hardened against common web vulnerabilities, employ TLS/SSL for all communications, and implement granular access controls for tools and resources.[8]

The modelcontextprotocol/python-sdk, particularly its FastMCP component, significantly lowers the barrier to entry for developing MCP servers in Python.[14] The proposed API-agnostic framework, built upon this SDK and leveraging design patterns like Plugin Architecture and Adapter, offers a powerful and extensible solution for integrating diverse third-party APIs. By dynamically loading API connector plugins, managing configurations with Pydantic, and ensuring secure secret handling, such a framework can provide a unified MCP interface to a multitude of backend services, making them readily accessible to LLM agents.

B. The Evolving Landscape of MCP and its Ecosystem

The Model Context Protocol is not a static specification but rather a dynamic and rapidly evolving field of research and development.[6] Its trajectory points towards an increasingly sophisticated and integral role in the future of AI.

Roadmap and Future Developments:
The official roadmap for MCP includes several key initiatives aimed at maturing the protocol and its ecosystem 6:

- **Tooling and Standardization**: Development of validation tools and compliance test suites to ensure consistency across implementations. Creation of reference client implementations and more example applications to aid developers.
- **Discovery**: Establishment of a centralized MCP Registry. This aims to make MCP servers more easily discoverable and installable by clients. The development of such a registry could be a pivotal moment for MCP adoption, making it far simpler for clients to find and connect to relevant tools. However, as Microsoft's experience with its own planned Windows MCP registry suggests [8], such a central directory also introduces significant governance challenges. Vetting servers for security, reliability, and trustworthiness will be paramount to prevent the registry from inadvertently becoming a distribution vector for malicious or poorly implemented servers. This implies a future need for clear submission guidelines, robust security review processes, and potentially community-driven reputation systems for listed MCP servers.
- **Advanced Capabilities**: Longer-term goals for MCP include support for complex, multi-agent workflows, often referred to as "Agent Graphs".[6] This

suggests a vision where MCP (or a related protocol) could serve as the communication backbone not just for agent-to-tool interactions, but also for agent-to-agent collaboration and task delegation.[6] Furthermore, the planned addition of multimodal capabilities would allow agents to process and interact with images, audio, and video through MCP tools, significantly broadening their operational scope.

- **Enhanced Security and Governance**: Continuous enhancements to security features, permission models, and the eventual establishment of formal governance structures and standardization bodies are anticipated.

Ecosystem Growth:
The MCP ecosystem has shown remarkable growth since its introduction. Reports of over 1,000 MCP servers becoming available within months of the protocol's launch 6 and significant corporate adoption by major players like Microsoft (integrating MCP into Windows 11) 8, Asana 9, Atlassian 9, Zapier 9, Cloudflare 9, and others, signal strong industry momentum. This is complemented by active community engagement and a growing number of open-source contributions.6
The long-term vision hinted at by concepts like "Agent Graphs" and multimodal support [6] indicates that MCP is positioned to be more than just a tool integration protocol. It aims to become a fundamental communication fabric for increasingly complex, capable, and collaborative AI systems. These advancements would dramatically expand the potential of what can be achieved with MCP, further cementing its role at the core of future AI architectures. As MCP continues to mature and its ecosystem expands, proficiency in its principles and implementation will become increasingly indispensable for developers and architects working at the forefront of artificial intelligence.

Works cited

1. What you need to know about the Model Context Protocol (MCP) - Merge.dev, accessed May 24, 2025, https://www.merge.dev/blog/model-context-protocol
2. Model Context Protocol (MCP) - Anthropic API, accessed May 24, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/mcp
3. How to MCP - The Complete Guide to Understanding Model Context ..., accessed May 24, 2025, https://simplescraper.io/blog/how-to-mcp
4. How to Use Model Context Protocol the Right Way | Boomi, accessed May 24, 2025, https://boomi.com/blog/model-context-protocol-how-to-use/
5. Model Context Protocol (MCP): A comprehensive introduction for developers - Stytch, accessed May 24, 2025, https://stytch.com/blog/model-context-protocol-introduction/
6. Model Context Protocol: What You Need To Know - Gradient Flow, accessed May 24, 2025,

https://gradientflow.com/model-context-protocol-what-you-need-to-know/
7. A beginners Guide on Model Context Protocol (MCP) - OpenCV, accessed May 24, 2025, https://opencv.org/blog/model-context-protocol/
8. Securing the Model Context Protocol: Building a safer agentic future ..., accessed May 24, 2025, https://blogs.windows.com/windowsexperience/2025/05/19/securing-the-model-context-protocol-building-a-safer-agentic-future-on-windows/
9. Remote MCP servers - Anthropic, accessed May 24, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/remote-mcp-servers
10. Model Context Protocol (MCP) - Everything You Need to Know, accessed May 24, 2025, https://zencoder.ai/blog/model-context-protocol
11. Model Context Protocol (MCP) an overview - Philschmid, accessed May 24, 2025, https://www.philschmid.de/mcp-introduction
12. How is JSON-RPC used in the Model Context Protocol? - Milvus, accessed May 24, 2025, https://milvus.io/ai-quick-reference/how-is-jsonrpc-used-in-the-model-context-protocol
13. Model Context Protocol Server Setup Guide | Step-by-Step Tutorial, accessed May 24, 2025, https://www.byteplus.com/en/topic/541333
14. The official Python SDK for Model Context Protocol servers and clients - GitHub, accessed May 24, 2025, https://github.com/modelcontextprotocol/python-sdk
15. Model Context Protocol (MCP) Hands-On Walkthrough for the Unstructured Workflow Endpoint, accessed May 24, 2025, https://docs.unstructured.io/examplecode/tools/mcp
16. Model Context Protocol (MCP): A Guide With Demo Project - DataCamp, accessed May 24, 2025, https://www.datacamp.com/tutorial/mcp-model-context-protocol
17. MCP Server in Python — Everything I Wish I'd Known on Day One | DigitalOcean, accessed May 24, 2025, https://www.digitalocean.com/community/tutorials/mcp-server-python
18. How to Build an MCP Server in Python: A Complete Guide - Scrapfly, accessed May 24, 2025, https://scrapfly.io/blog/how-to-build-an-mcp-server-in-python-a-complete-guide/
19. FastMCP Tutorial: Building MCP Servers in Python From Scratch - Firecrawl, accessed May 24, 2025, https://www.firecrawl.dev/blog/fastmcp-tutorial-building-mcp-servers-python
20. How to Implement a Model Context Protocol (MCP) Server with SSE - ni18 Blog, accessed May 24, 2025, https://blog.ni18.in/how-to-implement-a-model-context-protocol-mcp-server-with-sse/
21. MCP Python SDK Client Appends sessionId as Query Parameter · Issue #236 - GitHub, accessed May 24, 2025, https://github.com/modelcontextprotocol/python-sdk/issues/236
22. Use Google ADK and MCP with an external server | Google Cloud Blog, accessed

May 24, 2025, https://cloud.google.com/blog/topics/developers-practitioners/use-google-adk-and-mcp-with-an-external-server
23. Bringing streamable HTTP transport and Python language support to MCP servers, accessed May 24, 2025, https://blog.cloudflare.com/streamable-http-mcp-servers-python/
24. MCP Integration - GitHub Pages, accessed May 24, 2025, https://langchain-ai.github.io/langgraph/agents/mcp/
25. New capabilities for building agents on the Anthropic API, accessed May 24, 2025, https://www.anthropic.com/news/agent-capabilities-api
26. MCP connector - Anthropic, accessed May 24, 2025, https://docs.anthropic.com/en/docs/agents-and-tools/mcp-connector
27. Building and Using MCP Servers in Amazon Q CLI - DEV Community, accessed May 24, 2025, https://dev.to/welcloud-io/building-and-using-mcp-servers-in-amazon-q-cli-3im
28. MCP Server Session Lost in Multi-Worker Environment #520 - GitHub, accessed May 24, 2025, https://github.com/modelcontextprotocol/python-sdk/issues/520
29. CodeAnalysis MCP Server | Glama, accessed May 24, 2025, https://glama.ai/mcp/servers/@0xjcf/MCP_CodeAnalysis
30. State, and long-lived vs. short-lived connections #102 - GitHub, accessed May 24, 2025, https://github.com/modelcontextprotocol/modelcontextprotocol/discussions/102
31. OAuth Libraries for Python, accessed May 24, 2025, https://oauth.net/code/python/
32. Integrating Google Authentication with FastAPI: A Step-by-Step Guide - FutureSmart AI Blog, accessed May 24, 2025, https://blog.futuresmart.ai/integrating-google-authentication-with-fastapi-a-step-by-step-guide
33. Implementing OAuth 2.0 in REST APIs: Complete Guide, accessed May 24, 2025, https://blog.dreamfactory.com/implementing-oauth-2.0-in-rest-apis-complete-guide
34. Build and deploy Remote Model Context Protocol (MCP) servers to Cloudflare, accessed May 24, 2025, https://blog.cloudflare.com/es-la/remote-model-context-protocol-servers-mcp/
35. Build and deploy Remote Model Context Protocol (MCP) servers to Cloudflare, accessed May 24, 2025, https://blog.cloudflare.com/id-id/remote-model-context-protocol-servers-mcp/
36. Introducing Atlassian's Remote Model Context Protocol (MCP) Server, accessed May 24, 2025, https://www.atlassian.com/blog/announcements/remote-mcp-server
37. Piecing together the Agent puzzle: MCP, authentication & authorization, and Durable Objects free tier - The Cloudflare Blog, accessed May 24, 2025, https://blog.cloudflare.com/building-ai-agents-with-mcp-authn-authz-and-durable-objects/
38. github.com, accessed May 24, 2025,

https://github.com/modelcontextprotocol/python-sdk/tree/main/examples/clients/simple-auth-client
39. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/mcp/client/auth.py
40. Model Context Protocol : A New Standard for Defining APIs - DEV Community, accessed May 24, 2025, https://dev.to/epam_india_python/model-context-protocol-a-new-standard-for-defining-apis-19ih
41. MCP Client - Step by Step Guide to Building from Scratch - Composio, accessed May 24, 2025, https://composio.dev/blog/mcp-client-step-by-step-guide-to-building-from-scratch/
42. localstack/plux: A dynamic code loading framework for building pluggable Python distributions - GitHub, accessed May 24, 2025, https://github.com/localstack/plux
43. python - How to dynamically add and load entry points? - Stack Overflow, accessed May 24, 2025, https://stackoverflow.com/questions/40514205/how-to-dynamically-add-and-load-entry-points
44. Mypy - Pydantic, accessed May 24, 2025, https://docs.pydantic.dev/latest/integrations/mypy/
45. Mypy plugin - Pydantic, accessed May 24, 2025, https://docs.pydantic.dev/1.10/mypy_plugin/
46. A Complete Guide to Pydantic | Better Stack Community, accessed May 24, 2025, https://betterstack.com/community/guides/scaling-python/pydantic-explained/
47. Configuration - Pydantic, accessed May 24, 2025, https://docs.pydantic.dev/latest/concepts/config/
48. Configuration - Pydantic, accessed May 24, 2025, https://docs.pydantic.dev/latest/api/config/
49. Welcome to Pydaconf, accessed May 24, 2025, https://varadinov.github.io/pydaconf/
50. di : pythonic dependency injection, accessed May 24, 2025, https://adriangb.com/di/0.36.0/
51. Dependencies - FastAPI, accessed May 24, 2025, https://fastapi.tiangolo.com/tutorial/dependencies/
52. Python Dependency Injection: How to Do It Safely - Xygeni, accessed May 24, 2025, https://xygeni.io/blog/python-dependency-injection-how-to-do-it-safely/
53. Plugin Architecture in Java and Python - Scientific Research and Community, accessed May 24, 2025, https://www.onlinescientificresearch.com/articles/plugin-architecture-in-java-and-python.pdf
54. Dependency Injection in Python Programming - Custom Software Development - NG Logic, accessed May 24, 2025, https://nglogic.com/dependency-injection-python/
55. Dependencies - PydanticAI, accessed May 24, 2025,

https://ai.pydantic.dev/dependencies/
56. Design Patterns in Python - Refactoring.Guru, accessed May 24, 2025, https://refactoring.guru/design-patterns/python
57. faif/python-patterns: A collection of design patterns/idioms in Python - GitHub, accessed May 24, 2025, https://github.com/faif/python-patterns
58. Adapter in Python / Design Patterns - Refactoring.Guru, accessed May 24, 2025, https://refactoring.guru/design-patterns/adapter/python/example
59. Adapter Design Pattern - Definition and Examples | Belatrix Blog - Globant, accessed May 24, 2025, https://belatrix.globant.com/us-en/blog/tech-trends/adapter-design-pattern/
60. Strategy in Python / Design Patterns - Refactoring.Guru, accessed May 24, 2025, https://refactoring.guru/design-patterns/strategy/python/example
61. Become a Python Design Strategist using the Strategy Pattern - DEV Community, accessed May 24, 2025, https://dev.to/fayomihorace/become-a-python-design-strategist-using-the-strategy-pattern-6ad
62. Plugin Architecture for Python - Binary Coders - WordPress.com, accessed May 24, 2025, https://binarycoders.wordpress.com/2023/07/22/plugin-architecture-for-python/
63. How to build API integrations in Python - Merge.dev, accessed May 24, 2025, https://www.merge.dev/blog/api-integration-python
64. How to Store API Keys Securely - Strapi, accessed May 24, 2025, https://strapi.io/blog/how-to-store-API-keys-securely
65. Secure API Key Management: Best Practices - Lucid Financials, accessed May 24, 2025, https://lucid.now/blog/secure-api-key-management-best-practices/
66. Python SDK | Keeper Documentation, accessed May 24, 2025, https://docs.keeper.io/en/keeperpam/secrets-manager/developer-sdk-library/python-sdk
67. How to Handle Secrets in Python - GitGuardian Blog, accessed May 24, 2025, https://blog.gitguardian.com/how-to-handle-secrets-in-python/
68. Best Practices for API Key Safety | OpenAI Help Center, accessed May 24, 2025, https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety
69. Is there an official or community driven SDK for creating MCP clients? : r/ClaudeAI - Reddit, accessed May 24, 2025, https://www.reddit.com/r/ClaudeAI/comments/1h9kevw/is_there_an_official_or_community_driven_sdk_for/
70. For Client Developers - Model Context Protocol, accessed May 24, 2025, https://modelcontextprotocol.io/quickstart/client
71. Model context protocol (MCP) - OpenAI Agents SDK, accessed May 24, 2025, https://openai.github.io/openai-agents-python/mcp/
72. Guide to Using the Responses API's MCP Tool - OpenAI Cookbook, accessed May 24, 2025, https://cookbook.openai.com/examples/mcp/mcp_tool_guide
73. Releases · modelcontextprotocol/python-sdk - GitHub, accessed May 24, 2025, https://github.com/modelcontextprotocol/python-sdk/releases
74. MCP Python SDK 1.8.0 Streamable HTTP release : r/modelcontextprotocol -

Reddit, accessed May 24, 2025, https://www.reddit.com/r/modelcontextprotocol/comments/1kj1yb8/mcp_python_sdk_180_streamable_http_release/
75. Lifecycle and Hooks - PDM, accessed May 24, 2025, https://pdm-project.org/en/latest/usage/hooks/
76. Lifecycle and Hooks - PDM, accessed May 24, 2025, https://pdm-project.org/latest/usage/hooks/
77. github.com, accessed May 24, 2025, https://github.com/modelcontextprotocol/python-sdk/tree/main/examples/servers/simple-auth
78. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/mcp/server/auth/provider.py
79. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/docs/transports.md
80. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/tree/main/examples/servers/advanced-server-multiple-transports
81. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/issues?q=is%3Aissue+FastMCP+multiple+transports+label%3Aenhancement
82. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/mcp/server/fastmcp/server.py
83. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/docs/sessions.md
84. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/mcp/server/fastmcp/fastmcp.py
85. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/issues?q=is%3Aissue+FastMCP+session+store+redis+label%3Aenhancement
86. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/blob/main/mcp/server/fastmcp/session.py
87. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/tree/main/examples/clients
88. accessed January 1, 1970, https://github.com/modelcontextprotocol/python-sdk/tree/main/mcp/client
89. Model Context Protocol (MCP) Guide: What It Is & How to Use It - Leanware, accessed May 24, 2025, https://www.leanware.co/insights/model-context-protocol-guide
90. Model Context Protocol (MCP) 101: A Hands-On Beginner's Guide! - DEV

Community, accessed May 24, 2025, https://dev.to/pavanbelagatti/model-context-protocol-mcp-101-a-hands-on-beginners-guide-47ho