**MCP Server Architecture and Components**

**MCP servers** act as bridges between AI hosts (like IDEs or desktop tools) and external data sources or APIs. The architecture typically involves:

* **Hosts**: Applications that require external tool access.
* **Clients**: Maintain connections with MCP servers.
* **MCP Servers**: Expose specific functionalities, connecting to local or remote data sources.
* **Local/Remote Data Sources**: Files, databases, or APIs accessed by the server.

This modular approach ensures maintainability, scalability, and clear separation of concerns.

**Approaches to Building MCP Servers**

**1. Python SDK**

* **Setup**: Create a project directory, initialize a virtual environment, and install the MCP SDK.
* **Development**: Define server logic using classes (e.g., FastMCP), register tools, and implement async handlers for external API calls.
* **Example**: A weather server using Python and httpx for async requests.

**2. TypeScript/JavaScript SDK**

* **Setup**: Use Node.js, import the McpServer core class, and select a transport (stdio or HTTP).
* **Development**: Register prompts, resources, and tools, then instantiate and run the server.
* **Example**: Minimal server in src/index.ts with capability registration and a main loop.

**3. Other Languages/Frameworks**

* **Spring AI**: Java-based quick-starts are available for rapid prototyping.
* **Adapters**: Tools exist to bridge stdio and HTTP transports for compatibility.

**MCP Server Best Practices**

* **Resource Allocation**: Use dynamic scaling and workload-specific configurations.
* **Network Optimization**: Segment networks, use secure routing, and minimize latency.
* **Security**: Employ secure, non-deterministic session IDs, robust access controls, and encrypted communication.
* **Logging & Observability**: Log every prompt execution with metadata in append-only, tamper-evident systems for traceability and auditability.
* **Versioning**: Maintain clean, versioned repositories for models and tools to ensure reproducibility and rollback capability.
* **Performance Tuning**: Regular benchmarking, adaptive scaling, and continuous monitoring are essential for high-performance environments.

**Transport Mechanisms: Stdio vs. Streamable HTTP**

**Stdio Transport**

Server communicates via standard input/output streams, running as a subprocess on the local machine.

* **Strengths**:
  + Simple to implement and debug.
  + Fastest for local, single-user scenarios.
  + No network setup or authentication required.
  + Ideal for command-line tools and experiments.
* **Weaknesses**:
  + Limited to local use; not suitable for remote or multi-user/cloud scenarios.
  + No live streaming—responses are delivered only after computation completes.
  + Scalability is restricted to the host machine.

**Streamable HTTP Transport**

Server exposes an HTTP endpoint, handling POST/GET requests and streaming responses as they are generated.

* **Strengths**:
  + Supports both local and remote/cloud deployments.
  + Enables real-time, incremental response streaming—users see output as it's generated.
  + Scales efficiently to multiple users and integrates with standard HTTP infrastructure.
  + Supports token-based authentication and secure headers.
  + Backward compatible with SSE for legacy clients
* **Weaknesses**:
  + More complex to set up and secure.
  + Requires additional infrastructure (network, authentication, etc.).
  + Slightly higher operational overhead compared to stdio for simple use cases.

**Feature Comparison Table**

|  |  |  |
| --- | --- | --- |
| Feature/Transport | Stdio | Streamable HTTP |
| **Deployment** | Local only | Local or remote/cloud |
| **Setup Complexity** | Very low | Moderate to high |
| **Security** | OS-level only | Token/headers, HTTPS |
| **Streaming Support** | No (response at end) | Yes (incremental, real-time) |
| **Scalability** | Single user/process | Multi-user, scalable |
| **Use Cases** | CLI tools, experiments | Production, cloud, teams |
| **Debuggability** | Simple, direct | Standard HTTP tooling |
| **Integration** | Local-only, subprocess | Web, cloud, cross-platform |

**Recommendations**

* **Choose stdio** for rapid prototyping, local scripts, or single-user tools where simplicity and speed are paramount.
* **Choose streamable HTTP** for production, remote, or multi-user scenarios, especially where real-time feedback, security, and scalability are required.
* **Adapters** are available to bridge stdio and HTTP, allowing gradual migration or compatibility with clients that support only one transport.