MCP Architecture

MCP standardizes how AI applications connect to external tools and data sources, replacing one-off integrations with a universal protocol. Understanding the host-client-server architecture is the foundation for building or consuming any MCP service.

Key Roles

Role	Responsibility	Example
Host	Application that embeds MCP clients	Claude Desktop, VS Code, custom agent
Client	Maintains 1:1 connection with a server	SDK-provided, runs inside host
Server	Exposes tools, resources, prompts	File system server, database server, API server

Three Primitives

MCP exposes exactly three primitives -- tools, resources, and prompts -- each with a different control model. Knowing which primitive to use for a given capability determines whether the model, the application, or the user drives the interaction.

Primitive	Direction	Purpose	User Action
Tools	Server -> Model	Functions the LLM can call	Model-controlled (with user approval)
Resources	Server -> Client	Data/content for context	Application-controlled
Prompts	Server -> Client	Templated prompt workflows	User-controlled (slash commands)

Tools

- Model discovers available tools and decides when to call them
- Each tool has a name, description, and JSON Schema for inputs
- Server executes the tool and returns results
- Requires user approval before execution (security boundary)

Resources

- Provide context data without requiring a tool call
- Can be static (file contents) or dynamic (query results)
- Identified by URI (e.g., `file:///path/to/doc.md`)
- Support subscriptions for real-time updates

Prompts

- Predefined templates that combine instructions + context
- Surfaced as slash commands or menu items in the host
- Can accept arguments to customize behavior
- User explicitly triggers them

JSON-RPC Transport

MCP uses JSON-RPC 2.0 over two transport options: stdio for local servers and streamable HTTP for remote ones. Your transport choice determines deployment topology, latency characteristics, and security boundaries.

Stdio Transport

```
Host process <-- stdin/stdout --> Server process (child)
```

- Server runs as a child process of the host
- Communication via stdin (requests) and stdout (responses)
- Best for local servers

Streamable HTTP (SSE) Transport

```
Client <-- HTTP POST + SSE --> Server (remote)
```

- Server runs as an HTTP endpoint
- Client sends requests via POST
- Server streams responses via Server-Sent Events
- Best for remote/shared servers

Protocol Messages

These are the core JSON-RPC messages that every MCP client and server exchange. Understanding the initialize handshake and tool call flow is essential for debugging connection issues and building custom implementations.

Initialize

```
// Client -> Server
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": { "roots": { "listChanged": true } },
    "clientInfo": { "name": "MyApp", "version": "1.0.0" }
  }
}

// Server -> Client
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "tools": { "listChanged": true },
      "resources": { "subscribe": true },
      "prompts": { "listChanged": true }
    },
    "serverInfo": { "name": "MyServer", "version": "1.0.0" }
  }
}
```

List Tools

```
// Client -> Server
{ "jsonrpc": "2.0", "id": 2, "method": "tools/list" }

// Server -> Client
{
  "jsonrpc": "2.0", "id": 2,
  "result": {
    "tools": [{
      "name": "query_database",
      "description": "Run a read-only SQL query against the database",
      "inputSchema": {
        "type": "object",
        "properties": {
          "sql": { "type": "string", "description": "SQL SELECT query" }
        },
        "required": ["sql"]
      }
    }]
```

```
  }
}
```

Call Tool

```
// Client -> Server
{
  "jsonrpc": "2.0", "id": 3,
  "method": "tools/call",
  "params": {
    "name": "query_database",
    "arguments": { "sql": "SELECT count(*) FROM users" }
  }
}

// Server -> Client
{
  "jsonrpc": "2.0", "id": 3,
  "result": {
    "content": [{
      "type": "text",
      "text": "Count: 1,247"
    }],
    "isError": false
  }
}
```

Server Implementation (Python SDK)

The Python SDK provides a decorator-based API that turns ordinary functions into MCP tools, resources, and prompts with minimal boilerplate. This is the fastest way to expose existing Python code to any MCP-compatible host.

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("my-server")

# Define a tool
@mcp.tool()
def search_docs(query: str, limit: int = 5) -> str:
    """Search the documentation by keyword.

    Args:
        query: Search query string
        limit: Maximum number of results to return
    """
    results = db.search(query, limit=limit)
    return "\n".join(f"- {r.title}: {r.snippet}" for r in results)

# Define a resource
@mcp.resource("config://settings")
def get_settings() -> str:
    """Current application settings."""
    return json.dumps(load_settings())
```

```
# Dynamic resource with URI template
@mcp.resource("docs://{doc_id}")
def get_document(doc_id: str) -> str:
    """Retrieve a specific document by ID."""
    return db.get_document(doc_id).content

# Define a prompt
@mcp.prompt()
def review_code(language: str, code: str) -> str:
    """Review code for best practices and bugs."""
    return f"Review this {language} code for bugs, security issues, and style:\n\n```{language}\n{

# Run the server
if __name__ == "__main__":
    mcp.run()  # Default: stdio transport
```

Server Implementation (TypeScript SDK)

The TypeScript SDK uses Zod for schema validation and supports the same tool/resource/prompt primitives. Use this when your server logic is in Node.js or when you need tighter integration with TypeScript tooling.

```
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

const server = new McpServer({
  name: "my-server",
  version: "1.0.0",
});

// Define a tool
server.tool(
  "search_docs",
  "Search documentation by keyword",
  { query: z.string(), limit: z.number().default(5) },
  async ({ query, limit }) => {
    const results = await db.search(query, limit);
    return {
      content: [{ type: "text", text: JSON.stringify(results) }],
    };
  }
);

// Define a resource
server.resource("settings", "config://settings", async (uri) => ({
  contents: [{ uri: uri.href, mimeType: "application/json", text: JSON.stringify(settings) }],
}));

// Run
const transport = new StdioServerTransport();
await server.connect(transport);
```

Configuration (Claude Desktop Example)

MCP servers are registered in a JSON config file that tells the host how to launch or connect to each server. This is the entry point for adding new capabilities to Claude Desktop or any MCP-compatible application.

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem", "/Users/me/docs"],
      "env": {}
    },
    "database": {
      "command": "python",
      "args": ["-m", "my_db_server"],
      "env": {
        "DATABASE_URL": "postgresql://localhost/mydb"
      }
    },
    "remote-api": {
      "url": "https://api.example.com/mcp",
      "headers": {
        "Authorization": "Bearer ${API_TOKEN}"
      }
    }
  }
}
```

Available Community Servers

Before building a custom server, check this list -- there is likely an existing community server for common integrations like file systems, databases, and popular APIs.

Server	Capabilities	Transport
Filesystem	Read/write/search files	stdio
PostgreSQL	Query, schema inspection	stdio
GitHub	Issues, PRs, repos, search	stdio
Slack	Channels, messages, search	stdio
Google Drive	Read, search files	stdio
Puppeteer	Browser automation	stdio
Brave Search	Web search	stdio
Memory (knowledge graph)	Store/retrieve facts	stdio
Fetch	HTTP requests	stdio

Security Considerations

MCP servers execute real actions on real systems -- a misconfigured server can leak data, delete files, or run arbitrary code. Treat every MCP server as a security boundary that requires explicit trust and least-privilege access.

Concern	Mitigation
Tool execution	Require user approval before tool calls
Data exposure	Scope server access to minimum needed
Injection via tool results	Treat all tool output as untrusted
Credential leakage	Use environment variables, not hardcoded keys
Over-permissioned servers	Follow least-privilege principle
Remote server trust	Verify server identity, use HTTPS
Resource exfiltration	Monitor and log all tool invocations
Cross-server attacks	Isolate servers, no shared state

Security Checklist

- [ ] Server only accesses resources it needs
- [ ] All credentials passed via environment variables
- [ ] User approval required for destructive operations
- [ ] Tool inputs are validated and sanitized
- [ ] Tool outputs are treated as untrusted by the host
- [ ] Remote servers use HTTPS with proper certificates
- [ ] Access logs are maintained
- [ ] Rate limiting is configured for remote servers

Common Pitfalls

Most MCP integration failures come from vague tool descriptions, missing error handling, or over-permissioned servers. Fix these before you debug the LLM's behavior.

Pitfall	Problem	Fix
Too many tools	LLM confused about which to use	Keep under 15 tools per server, use clear names
Vague tool descriptions	Wrong tool calls	Write specific descriptions, include examples
No error handling in tools	Crashes or silent failures	Return error in content with `isError: true`
Blocking operations	Server hangs	Use async operations, add timeouts

Pitfall	Problem	Fix
Large response payloads	Token waste, slow responses	Paginate results, summarize
No input validation	Security vulnerabilities, crashes	Validate all inputs against schema
Missing capabilities declaration	Client doesn't know what's available	Declare all capabilities in initialize
Hardcoded paths/URLs	Not portable across environments	Use configuration and environment variables