🔧 MCP Architecture (Flow Diagram)

1. Participants

- MCP Host → The main AI application (e.g., Claude Desktop, VS Code)
- MCP Client → Manages a one-to-one connection with a single MCP server
- MCP Server → Provides context data/tools/resources to clients

📌 Example:
VS Code (Host) → connects to Sentry Server (Server 1) & Filesystem Server (Server 2) via separate Clients.

text

```
      ┌────────────┐
      │  MCP Host  │    (AI App: Claude, VS Code, etc.)
      └──────┬─────┘
             │ manages
             ▼ 1-1 mapping
┌────────────────────────┐
│    MCP Clients (N)     │
│ (connector per server) │
└───┬────────┬───────────┘
    │        │
    │        │
┌───▼───┐ ┌──▼────┐  ...
│Server1│ │Server2│  (etc.)
│ Tools │ │Filesys│
│ Resrc │ │DB, API│
│Prompt │ │Custom │
└───────┘ └───────┘
```

📑 Layered Model (Concept Stack)

2. Layers

- Data Layer → Inner layer, JSON-RPC 2.0 protocol
  - Manages lifecycle (init/terminate)
  - Provides primitives (tools, resources, prompts, etc.)
  - Supports client features (sampling, elicitation, logging)
  - Utility (notifications, progress updates)
- Transport Layer → Outer communication layer
  - Stdio Transport → Local process, fast, no network overhead
  - Streamable HTTP Transport → Remote, supports HTTP, SSE, OAuth tokens, API Keys, etc.
  - Same JSON-RPC messages reused across transports

text

```
+----------------------+
| Transport Layer      |   (stdio OR Streamable HTTP)
| - Connection open    |
| - Message framing    |
| - Auth (OAuth, API)  |
+----------------------+
| Data Layer           |   (JSON-RPC 2.0 protocol)
| - Lifecycle mgmt     |
| - Primitives (tools/resources/prompts)
| - Notifications      |
+----------------------+
```

---

3. Primitives

Server-side primitives:

- Tools → Executable actions (queries, APIs, file ops)
- Resources → Context data (schemas, file contents, API responses)
- Prompts → Reusable templates (system prompts, examples)

Client-side primitives:

- Sampling → Server asks AI client for model completion
- Elicitation → Server asks for user input/confirmation
- Logging → Server sends logs/debug info

🎯 MCP Primitives

Server → Client

- 🛠️ **Tools** → Actions (file ops, queries, APIs)
- 📚 **Resources** → Data/context (schemas, records)
- 📝 **Prompts** → Templates/examples

Client → Server

- 🔮 **Sampling** → Ask LLM for a completion
- 🙋 **Elicitation** → Ask user for input/approval
- 📜 **Logging** → Send logs/debug info

---

4. Lifecycle Management

- Protocol is stateful → requires init, negotiation, termination
- Handshake steps:
  1. Initialize → exchange capabilities + protocol version
  2. Capabilities → each declares what features/notifications they support (tools/resources/etc.)
  3. Identity exchange (clientInfo, serverInfo)
  4. Send `notifications/initialized`

🔄 Lifecycle Flow (Step by Step)

text

```
Client → Server : initialize (protocolVersion, capabilities,
clientInfo)
Server → Client : response (supported primitives, serverInfo)
Client → Server : notification "initialized"
```

```
Now session is READY → supported tools/resources/prompts registered
```

---

5. Workflow Example

1. Initialization
   → Client sends `initialize` → Server responds with supported capabilities → Client signals "ready"
2. Tool Discovery
   → Client sends `tools/list` → gets back tools metadata (name, desc, inputSchema)
3. Tool Execution
   → Client sends `tools/call` → server executes and returns structured response (content array: text, img, resources)
4. Real-time Updates (Notifications)
   → Server sends, e.g., `notifications/tools/list_changed` → Client refreshes tool list

⚙ Example Interaction

1. **Tool Discovery**

text

```
Client → Server : "tools/list"
Server → Client : [ { name:"weather_current", schema:{...}, ... } ]
```

2. **Tool Execution**

text

```
Client → Server : "tools/call" {name:"weather_current", args:{loc:"SF"} }
Server → Client : {content:[ {type:"text", value:"72°F"} ]}
```

3. **Notification**

text

```
Server → Client : "notifications/tools/list_changed"
Client → Server : "tools/list" (refresh!)
```

---

6. Notifications

- JSON-RPC notifications (no response expected)
- Keep clients & servers in sync dynamically
- Use cases: tool availability changes, resource updates
- Ensures efficiency (no polling), consistency, real-time collaboration

---

7. How It Works Inside AI Applications

- Host AI app registers MCP servers + their capabilities
- LLM can use available tools/resources/prompts from multiple servers
- When servers update tools → client updates registry → LLM adapts mid-conversation

🧠 How It Feels in an AI App

- Host (Claude Desktop, VS Code) = **control tower**

- Each MCP Client = **bridge** to one MCP server

- Servers = **data/action providers**

- LLM = **orchestrator** that picks which tools/prompts/resources to use

- Notifications = **live sync** so the system adapts on the fly

---