MTR MCP Concepts

Code

git clone https://github.com/enoch-sit/mtr-mcp-example-fastapi.git

or in Moodle

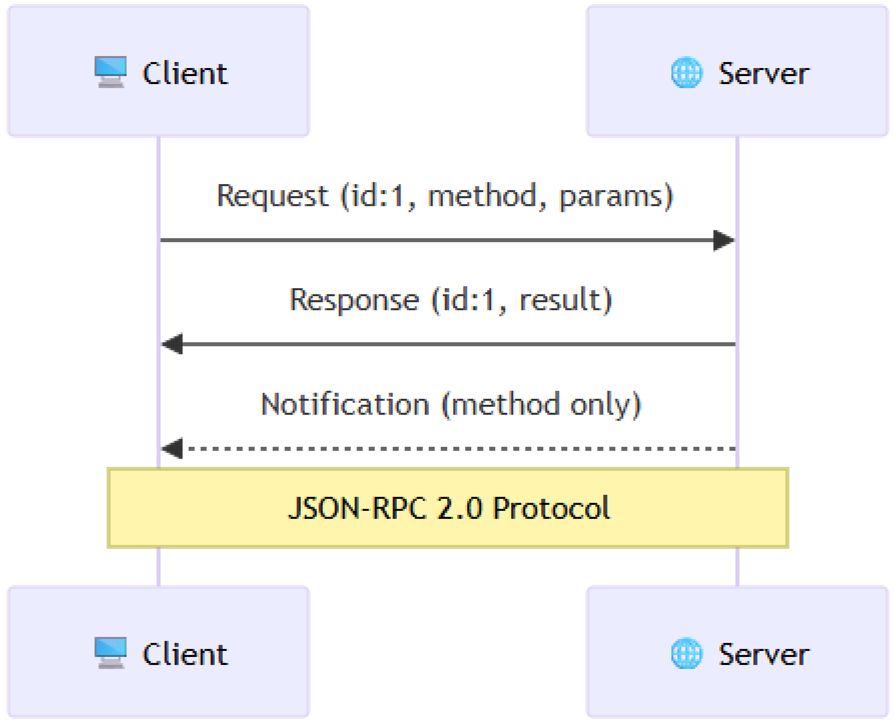
What is MCP (Model Context Protocol)?

- An open protocol created by Anthropic that standardizes how AI applications communicate with external data sources and tools.
- Like USB-C for AI - one universal interface for all context sources, enabling seamless integration.
- Uses JSON-RPC 2.0 over stdio or HTTP/SSE transports with a three-layer architecture: Application → Data → Transport.
-

What is JSON-RPC?

- JSON-RPC (JSON Remote Procedure Call) is a lightweight protocol for calling functions on a remote server as if they were local.
- Uses simple JSON format for requests (client asks), responses (server replies), and notifications (one-way messages).
- MCP chose JSON-RPC because it's simple, bidirectional, language-agnostic, standardized, and lightweight.

What is JSON-RPC?



```
//Request (Client asks server to do something)
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {"name": "calculator", "args": {"a": 5, "b": 3}}
}
//Response (Server replies with result)
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {"content": [{"type": "text", "text": "8"}]}
}
//Notification (One-way, no response expected)
{
  "jsonrpc": "2.0",
  "method": "notifications/tools/list_changed"
}
```

Understanding HTTP & SSE

- HTTP is the foundation of web communication - a text-based protocol using request-response model with methods (GET, POST, etc.).

- SSE (Server-Sent Events) enables real-time server→client streaming over HTTP, keeping connections open for continuous updates.

- HTTP is UTF-8 encoded text (not Base64), packaged into TCP packets for network transmission with automatic reliability.

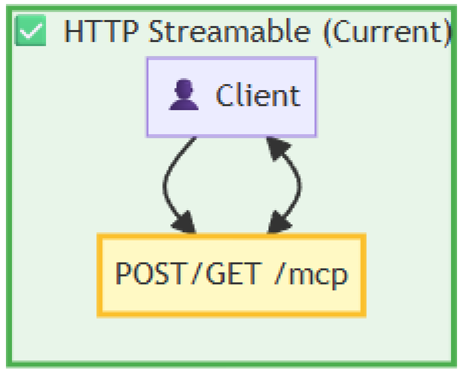
```
# Traditional HTTP (Request-Response)
Client: "GET /data HTTP/1.1"  →  Server
Client  ← "200 OK: Here's data"  Server
[Connection closes]

# SSE (Server Streaming)
Client: "GET /events HTTP/1.1"  →  Server
      "Accept: text/event-stream"
      ← "data: Update 1"      Server
      ← "data: Update 2"      Server
      ← "data: Update 3"      Server
[Connection stays open]
```

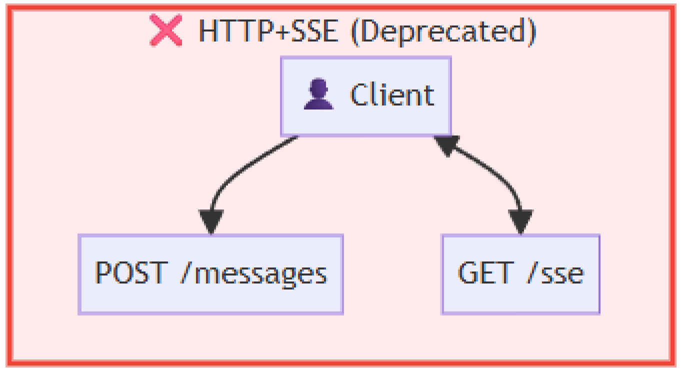
HTTP+SSE vs HTTP Streamable Evolution

- OLD (HTTP+SSE, 2024-11-05): Required TWO separate endpoints - /messages (POST) and /sse (GET) - complex to configure.
- NEW (HTTP Streamable, 2025-06-18): ONE unified /mcp endpoint handles both POST and GET - simpler, more secure.
- Benefits: 50% fewer endpoints, unified security, easier CORS, better load balancing, flexible responses (JSON or SSE).

HTTP+SSE vs HTTP Streamable Evolution

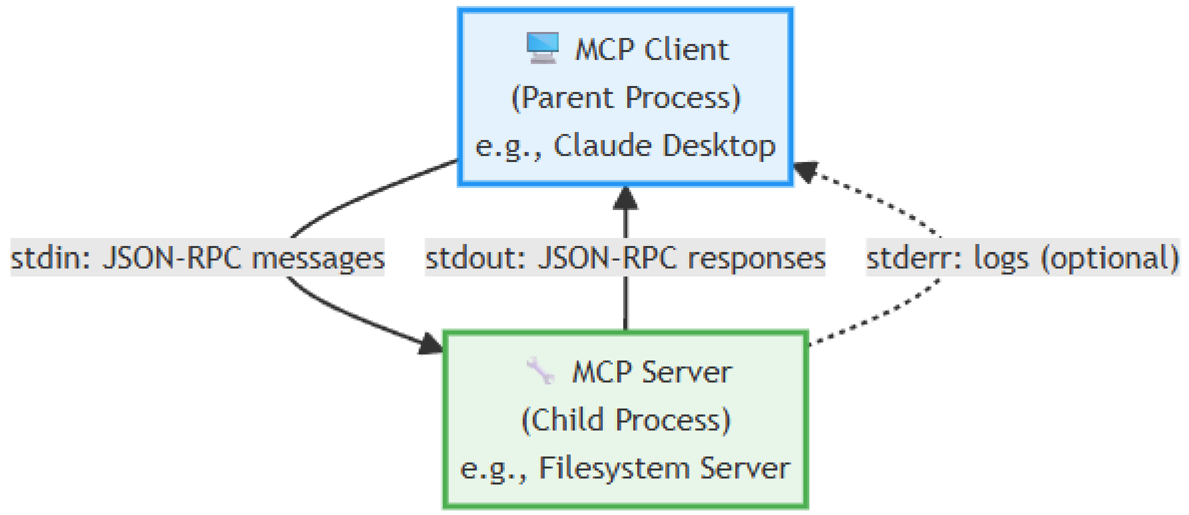
```python
# OLD: HTTP+SSE (DEPRECATED)
@app.post("/messages")
async def handle_messages(request):
    # Handle JSON-RPC messages

@app.get("/sse")
async def handle_sse(request):
    # Handle SSE stream

# NEW: HTTP Streamable (CURRENT)
@app.api_route("/mcp", methods=["GET", "POST"])
async def handle_mcp(request):
    if request.method == "POST":
        # Can return JSON or SSE stream
    elif request.method == "GET":
        # Return SSE stream
```





MCP Transport: stdio

- stdio transport enables local process communication - client launches MCP server as subprocess and communicates via stdin/stdout.
- Messages are newline-delimited JSON-RPC on single lines, UTF-8 encoded. Server can write logs to stderr (not protocol).
- Best for: Desktop apps (Claude, VS Code), local integrations, low latency. NOT for: remote servers, web apps, multiple clients.



MCP Transport: stdio

```
# Client launches server as subprocess
$ /path/to/mcp-server


# Client sends to stdin
{"jsonrpc":"2.0","id":1,"method":"initialize",...}


# Server responds via stdout
{"jsonrpc":"2.0","id":1,"result":{...}}


# Server logs to stderr (optional)
[2025-10-25 10:30:15] INFO: Server initialized


# Claude Desktop config example
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-filesystem",
"/Users/data"]
    }
  }
}
```

MCP Transport: HTTP Streamable

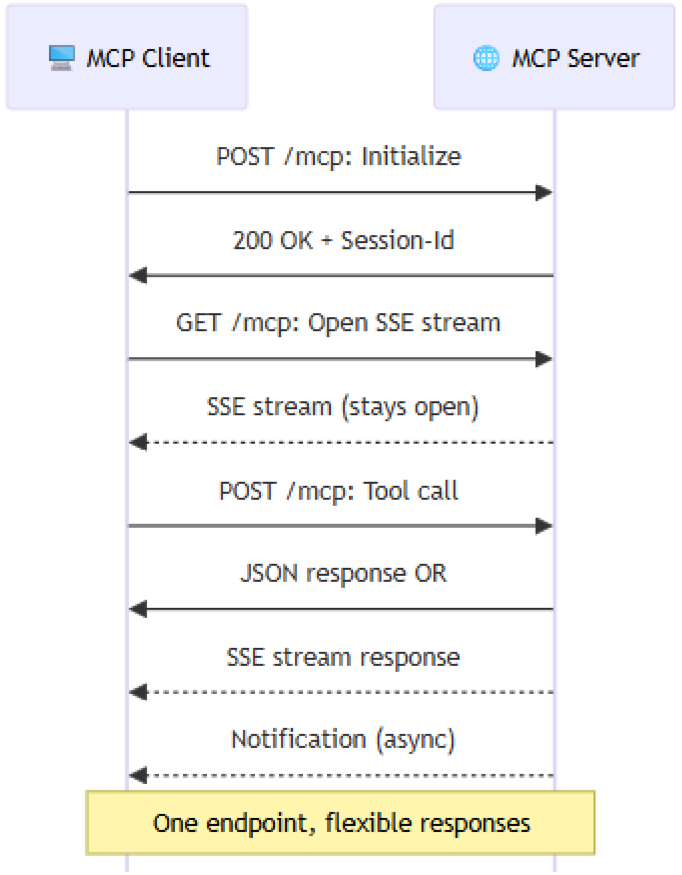
- HTTP Streamable enables remote web service communication with one unified /mcp endpoint for POST (send) and GET (receive SSE).

- Server can respond with direct JSON (simple requests) or SSE stream (complex requests with progress updates).

- Session management via Mcp-Session-Id header links all requests, enables stateful multi-turn interactions.

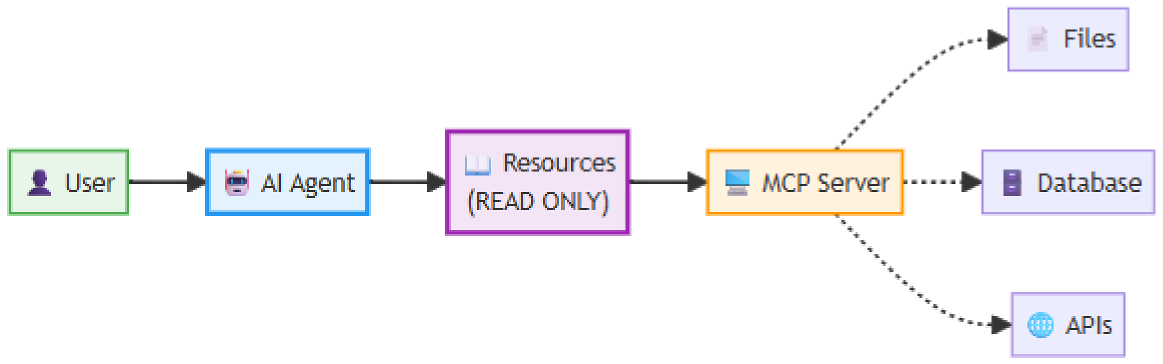
MCP Transport: HTTP Streamable

```
# 1. Initialize (POST /mcp)
POST /mcp HTTP/1.1
{"jsonrpc":"2.0","id":1,"method":"initialize",...}


Response:
Mcp-Session-Id: session-abc-123
{"jsonrpc":"2.0","id":1,"result":{...}}


# 2. Open SSE Stream (GET /mcp)
GET /mcp HTTP/1.1
Mcp-Session-Id: session-abc-123
Accept: text/event-stream


# 3. Send Tool Call (POST /mcp)
POST /mcp HTTP/1.1
Mcp-Session-Id: session-abc-123
{"jsonrpc":"2.0","id":2,"method":"tools/call",...}
```



MCP Core Primitive: Resources

- Resources are passive data sources that clients can READ (like files, database records, API responses) - context for LLMs.
- Resources have URIs (e.g., file:///, db://), MIME types (text/plain, application/json), and optional metadata.
- Flow: Client requests resources/list → Server returns available resources → Client reads specific resource by URI.



MCP Core Primitive: Resources

```
//1. List available resources
Request:  {"jsonrpc":"2.0","id":1,"method":"resources/list"}
Response: {
  "resources": [
    {
      "uri": "file:///project/README.md",
      "name": "Project README",
      "mimeType": "text/plain"
    },
    {
      "uri": "db://users/12345",
      "name": "User Profile",
      "mimeType": "application/json"
    }
  ]
}

//2. Read specific resource
Request:  {"jsonrpc":"2.0","id":2,"method":"resources/read",
           "params":{"uri":"file:///project/README.md"}}
Response: {"contents":[{"uri":"...", "text":"..."}]}
```

MCP Core Primitive: Prompts

- Prompts are reusable templates/workflows (like 'code review template', 'SQL query generator') with placeholders for arguments.
- Server defines prompts with names, descriptions, and argument schemas. Client can request and use them with custom values.
- Flow: Client requests prompts/list → Server returns templates → Client gets prompt with args → Server returns filled template.

MCP Core Primitive: Prompts

```
// 1. List available prompts
Request:  {"jsonrpc":"2.0","id":1,"method":"prompts/list"}
Response: {
  "prompts": [
    {
      "name": "code-review",
      "description": "Review code for best practices",
      "arguments": [
        {"name": "language", "description": "Programming language"}
      ]
    }
  ]
}


// 2. Get prompt with arguments
Request:  {"jsonrpc":"2.0","id":2,"method":"prompts/get",
           "params":{"name":"code-review",
                     "arguments":{"language":"Python"}}}
Response: {
  "messages": [
    {"role": "user", "content": "Review this Python code..."}
  ]
}
```

MCP Core Primitive: Tools

- Tools are executable functions that agents can CALL to perform actions (like calculations, file writes, API calls) - active operations.
- Tools have names, descriptions (critical for LLM decision), and input schemas (JSON Schema for validation).
- Flow: Client requests tools/list → Server returns available tools → Client calls tool → Server executes & returns result.

MCP Core Primitive: Tools

```
// 1. List available tools
Request:  {"jsonrpc":"2.0","id":1,"method":"tools/list"}
Response: {
  "tools": [
    {
      "name": "calculator",
      "description": "Perform arithmetic calculations",
      "inputSchema": {
        "type": "object",
        "properties": {
          "operation": {"type": "string"},
          "a": {"type": "number"},
          "b": {"type": "number"}
        }
      }
    }
  ]
}

// 2. Call tool
Request:  {"jsonrpc":"2.0","id":2,"method":"tools/call",
           "params":{"name":"calculator",
                     "arguments":{"operation":"add","a":5,"b":3}}}
Response: {"content":[{"type":"text","text":"8"}]}
```

MCP Primitives: When to Use Which?

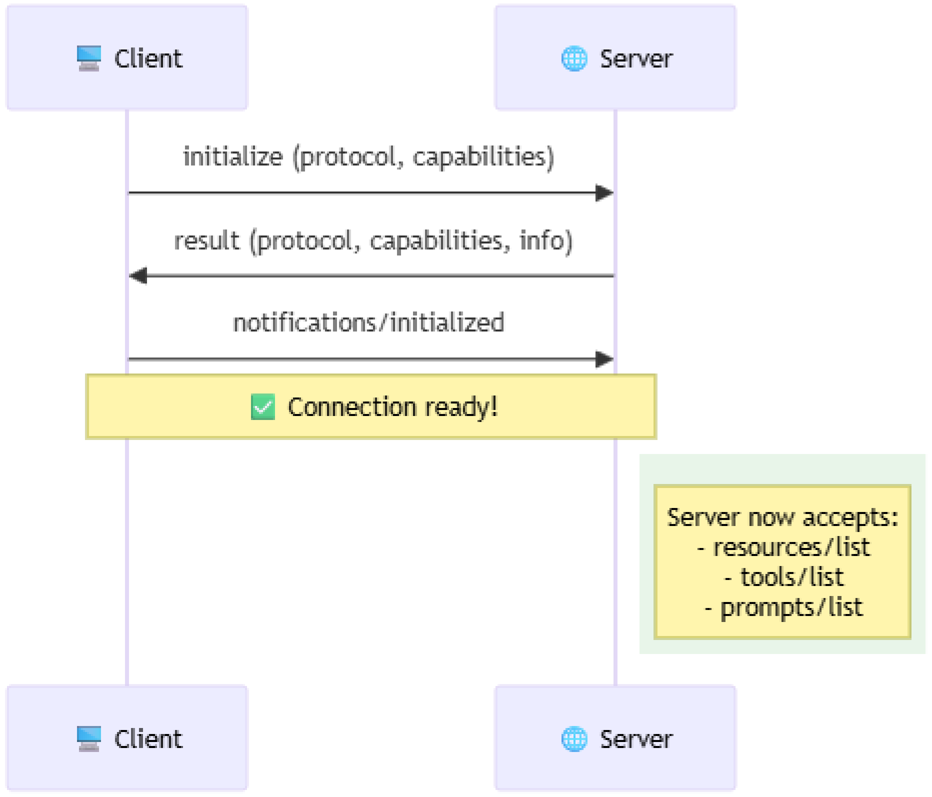
- Use Resources when: You need to READ data (files, docs, DB records) for context without modification - passive data access.

- Use Prompts when: You want reusable templates/workflows with placeholders - structured guidance for LLMs.

- Use Tools when: You need to EXECUTE actions (calculations, writes, API calls) that change state - active operations.

MCP Lifecycle: Initialization Phase

- Client sends initialize request with protocol version and capabilities (experimental, roots, sampling).
- Server responds with its protocol version, capabilities (resources, tools, prompts, logging), and server info.
- Client sends initialized notification to confirm completion. Connection is now ready for interactions.



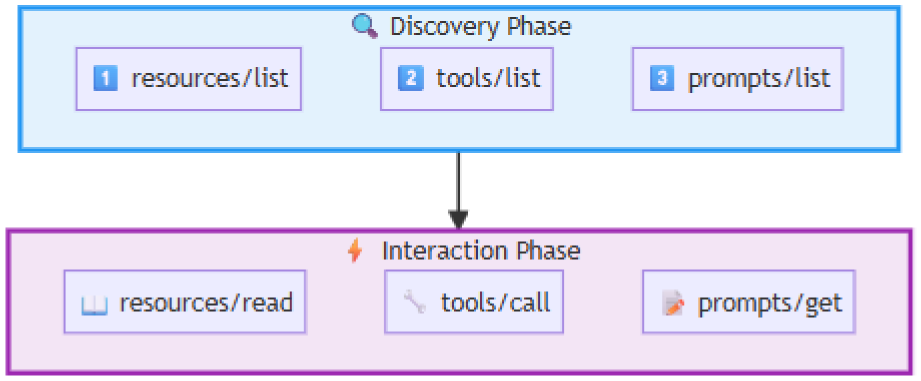
MCP Lifecycle: Initialization Phase

```
// 1. Client → Server: Initialize request
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "experimental": {},
      "roots": {"listChanged": true},
      "sampling": {}
    },
    "clientInfo": {"name": "MyClient", "version": "1.0.0"}
  }
}
```

```
// 2. Server → Client: Initialize response
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "resources": {"subscribe": true, "listChanged": true},
      "tools": {"listChanged": true},
      "prompts": {"listChanged": true},
      "logging": {}
    },
    "serverInfo": {"name": "MyServer", "version": "1.0.0"}
  }
}


// 3. Client → Server: Initialized notification
{"jsonrpc": "2.0", "method": "notifications/initialized"}
```

MCP Lifecycle: Discovery & Interaction

- Discovery: Client lists available resources/prompts/tools from server to understand capabilities.
- Interaction: Client reads resources (for context), gets prompts (for templates), calls tools (for actions).
- Server can send notifications when lists change (resources/tools/prompts updated) - client can re-query.



MCP Lifecycle: Discovery & Interaction

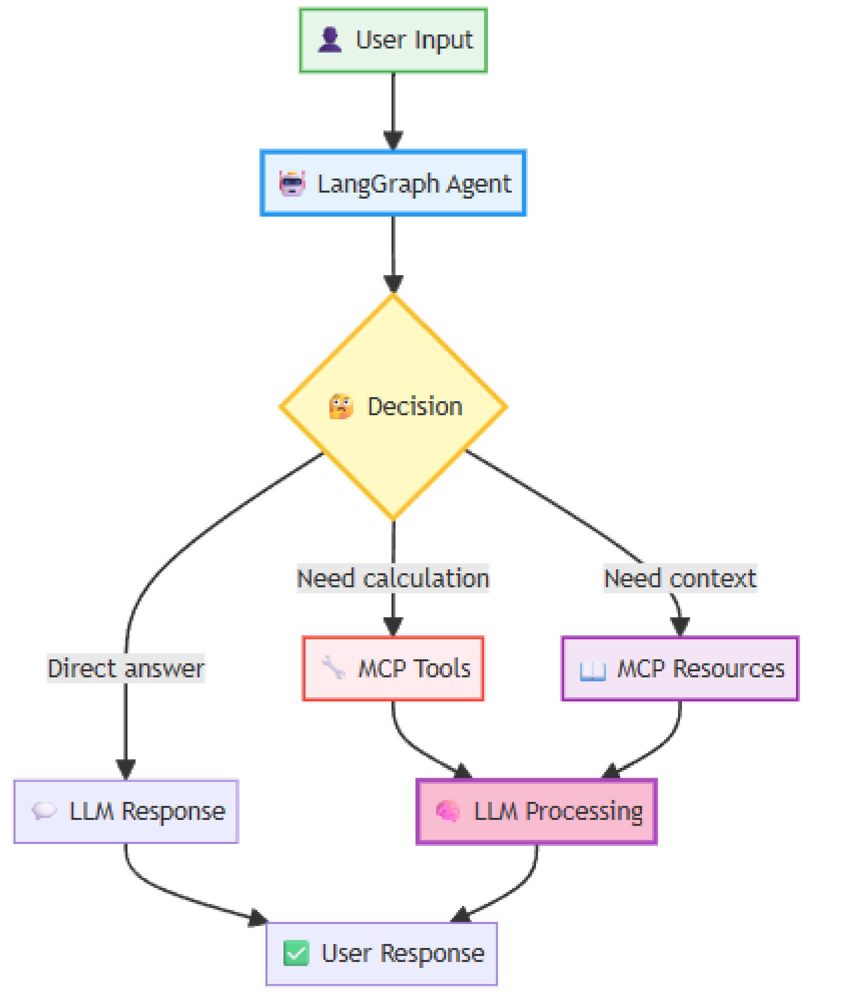
```
// DISCOVERY PHASE
// List resources
{"jsonrpc":"2.0","id":2,"method":"resources/list"}
→ {"resources":[{"uri":"file:///data","name":"Data"}]}


// List tools
{"jsonrpc":"2.0","id":3,"method":"tools/list"}
→ {"tools":[{"name":"calculator","description":"..."}]}


// List prompts
{"jsonrpc":"2.0","id":4,"method":"prompts/list"}
→ {"prompts":[{"name":"review","description":"..."}]}


// INTERACTION PHASE
// Read resource
{"jsonrpc":"2.0","id":5,"method":"resources/read",
 "params":{"uri":"file:///data"}}


// Call tool
{"jsonrpc":"2.0","id":6,"method":"tools/call",
 "params":{"name":"calculator","arguments":{...}}}


// Get prompt
{"jsonrpc":"2.0","id":7,"method":"prompts/get",
 "params":{"name":"review","arguments":{...}}}
```

MCP + LangGraph Integration

- LangGraph agents use MCP to access external tools and data during decision-making loops.
- Flow: User input → Agent decides → Calls MCP tools OR reads MCP resources → LLM processes → Responds to user.
- MCP provides standardized interface for tools/resources, LangGraph provides agent orchestration and state management.

MCP + LangGraph Integration

```python
from langgraph.prebuilt import create_react_agent
from langchain_aws import ChatBedrockConverse
from langchain_core.tools import tool


# Define MCP-backed tool
@tool
def mcp_calculator(operation: str, a: float, b: float) -> float:
    """Calculate using MCP calculator tool."""
    # Calls MCP server's calculator tool
    response = mcp_client.call_tool(
        "calculator",
        {"operation": operation, "a": a, "b": b}
    )
    return response["content"][0]["text"]


# Create LangGraph agent with MCP tools
llm = ChatBedrockConverse(model="amazon.nova-lite-v1:0")
tools = [mcp_calculator]
agent = create_react_agent(llm, tools)


# Use agent
result = agent.invoke({
    "messages": [{"role": "user", "content": "What is 15 + 27?"}]
})
```

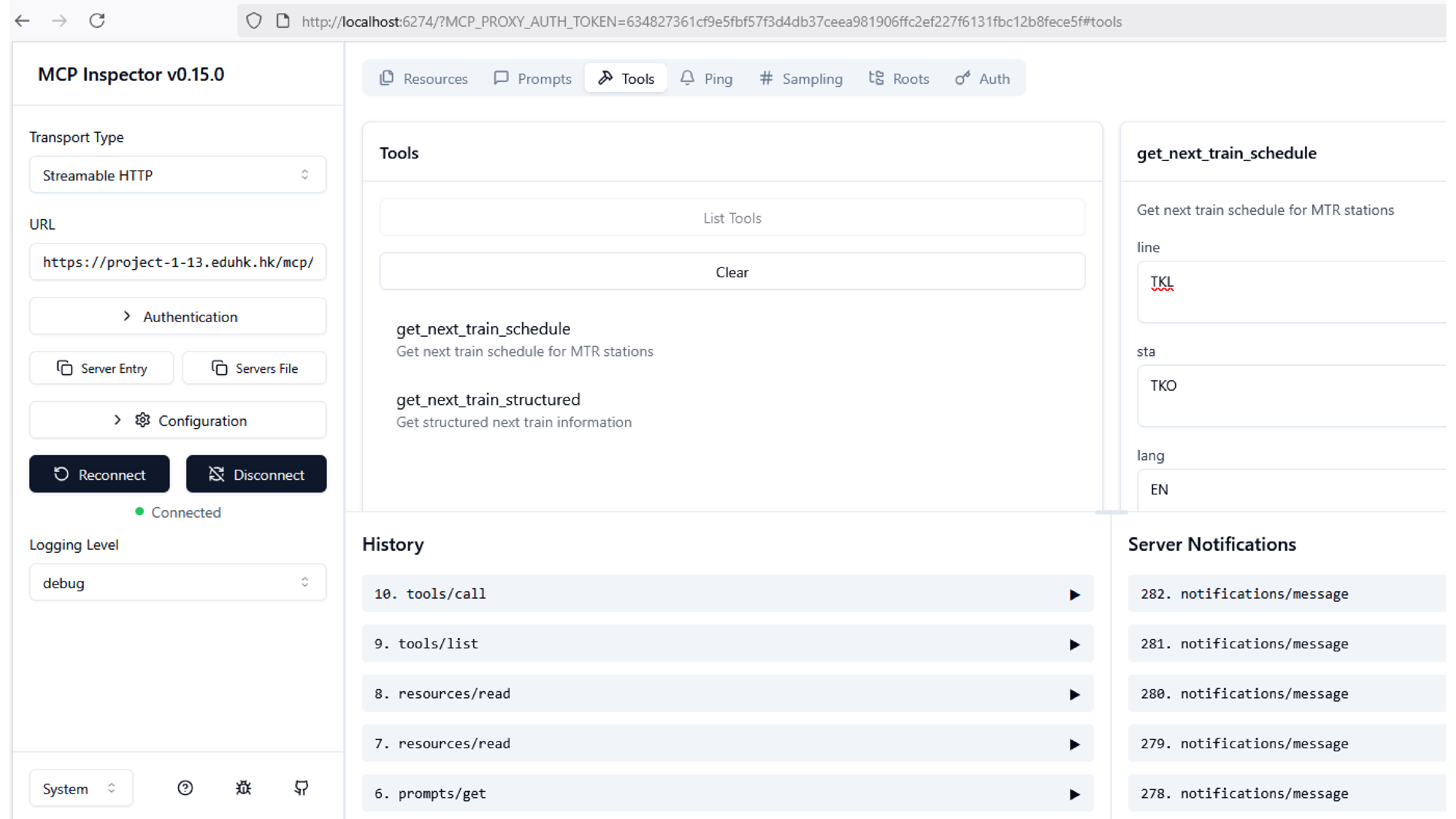


Example: MTR MCP Server

- MTR (Morning-Tea-Research) MCP server deployed at https://project-1-04.eduhk.hk/mcp/ using HTTP Streamable transport.
- MCP inspector to evaluate and inspect

MCP Inspector

- npx @modelcontextprotocol/inspector


http://localhost:6274/?MCP_PROXY_AUTH_TOKEN=634827361cf9e5fbf57f3d4db37ceea981906ffc2ef227f6131fbc12b8fece5f#tools

MCP Inspector v0.15.0

Transport Type: Streamable HTTP

URL: https://project-1-13.eduhk.hk/mcp/

Connected

Logging Level: debug

Tools

get_next_train_schedule
Get next train schedule for MTR stations

get_next_train_structured
Get structured next train information

get_next_train_schedule

Get next train schedule for MTR stations

line: TKL

sta: TKO

lang: EN

History

```
10. tools/call
9. tools/list
8. resources/read
7. resources/read
6. prompts/get
```

Server Notifications

```
282. notifications/message
281. notifications/message
280. notifications/message
279. notifications/message
278. notifications/message
```

Nginx Config Concepts

- location /mcp/ {}
  - Proxy to Docker container on localhost:8080 proxy_pass http://localhost:8080/mcp/;

- Essential for SSE (Server-Sent Events)
  - proxy_buffering off;   # CRITICAL: Must disable buffering for SSE
  - proxy_cache off;   # Disable caching for real-time streams

Nginx Config Concepts

- proxy_buffering off;
  - Disables response buffering. Normally, Nginx buffers responses for efficiency, but for SSE, this must be off to allow immediate, chunked streaming of events (e.g., tool results). Buffering would delay or break the real-time flow.
- proxy_cache off;
  - Disables caching of responses. SSE data is dynamic and real-time (e.g., live train schedules), so caching could serve stale content.

Nginx Config Concepts

- proxy_set_header Host $host;
  - Sets the Host header to the original client's requested host (e.g., project-1-04.eduhk.hk). This helps the backend server know the original domain.

- proxy_set_header X-Real-IP $remote_addr;
  - Adds a header with the client's real IP address (from Nginx's $remote_addr variable). Useful for logging or access control on the backend.

Nginx Config Concepts

- proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  - Appends the client's IP to the X-Forwarded-For header chain, tracking the request path through proxies. $proxy_add_x_forwarded_for automatically handles appending.
- proxy_set_header X-Forwarded-Proto $scheme;
  - Sets a header indicating the original protocol (e.g., https if the client used HTTPS). This informs the backend if the connection was secure, even though the proxy uses HTTP internally.

Nginx Config Concepts

- proxy_read_timeout 86400s;
    - Sets the timeout for reading a response from the backend to 86,400 seconds (24 hours). If no data is received within this time, the connection times out. This prevents indefinite hangs for idle SSE streams.
- proxy_send_timeout 86400s;
    - Sets the timeout for sending data to the backend to 24 hours. Similar to above, it accommodates slow or infrequent SSE events.
- gzip off;
    - Disables GZIP compression for responses in this location. SSE streams are text-based and often small; compression can interfere with chunked streaming or cause buffering issues.

Nginx Config Concepts

- add_header 'Access-Control-Allow-Origin' '*' always;
  - Allows requests from any origin (*). The always ensures the header is added even on error responses.
- add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS' always;
  - Permits these HTTP methods for cross-origin requests.
- add_header 'Access-Control-Allow-Headers' 'Content-Type, Accept' always;
  - Allows these specific headers in cross-origin requests.

Nginx Config Concepts

- if ($request_method = 'OPTIONS') { ... }
  - Conditional block that checks if the request method is OPTIONS (using Nginx's $request_method variable).

- return 204;
  - Responds with HTTP 204 (No Content) for preflight requests, indicating approval without sending a body. This satisfies CORS without proxying to the backend.