AZAMZAM.AI

First Edition

A Practical Developer's Guide

Understanding MCP

Model Context Protocol — From Concepts to Implementation

```
{ "method": "tools/call",
  "params": { "name": "PingME" } }
→  result: "BISMILLAH"
```

PingME server source · HTTP transport · Wire format · LLM tool triggering

Ahmad Zam Zam Jusoh

MCP 2025-06-18 · Java / Tomcat

Understanding MCP

Model Context Protocol — from concepts to a working Java implementation

This guide walks through the Model Context Protocol (MCP) from first principles, explains how its HTTP transport actually works on the wire, and builds a complete **PingME** server in Java using bare servlets on Tomcat. It then digs into two follow-up questions: how clients interpret tool output without an output schema, and what happens to a tool result once it returns to the host.

*Use case: one tool, no arguments, returns the magic word **BISMILLAH**.*

Contents

	Preface	**3**
	Objectives · assumptions · structure · conventions	
Part 1	What is MCP, really?	**5**
	The three roles · what servers expose · JSON-RPC 2.0 · transports	
Part 2	How HTTP MCP works	**8**
	Single endpoint · the handshake · the four methods · wire format	
Part 3	Implementation — Java Servlet on Tomcat	**11**
	Project structure · pom.xml · web.xml · the servlet · walkthrough	
Part 4	Build, deploy, test	**16**
	Maven build · Tomcat deploy · curl tests · connecting to Claude.ai	
Part 5	Why no output schema?	**18**
	Fixed output contract · why the design · the optional outputSchema	
Part 6	What happens to the result?	**20**
	The full cycle · what the model sees · interception points	
Part 7	How the LLM decides to call a tool	**23**
	Trained behaviour · the description as trigger · tool_choice	
	Usage examples: Claude.ai and Claude Code · writing good descriptions	
	Appendix · Glossary of MCP terms · About the Author	**27**

Preface

The Model Context Protocol (MCP) is one of those specifications that looks deceptively simple on the surface — a few JSON messages over HTTP — but rewards careful study. Once you understand what is really happening at each step, a whole class of AI integration problems suddenly has a clean, standard solution.

This guide was written to provide exactly that understanding: not just *how* to implement MCP, but *why* it is designed the way it is, and what that design means for the applications you build on top of it.

Objectives

By the end of this guide you will be able to:

- Explain what MCP is, what problem it solves, and how it fits into the broader AI ecosystem.
- Describe the Streamable HTTP transport — the handshake, the four core methods, and the wire format for each.
- Build a working MCP server from scratch using Java servlets on Tomcat, with no framework dependencies beyond Jackson.
- Understand how and why the LLM decides to call a registered tool — and how to write tool descriptions that trigger reliably.
- Explain what happens to a tool result after it is returned: how the host translates it, how the model sees it, and where interception can occur.
- Identify real-world use cases where MCP adds genuine business value.

Assumptions

This guide assumes the following about the reader:

- **Java familiarity.** You are comfortable reading and writing Java. The implementation uses Java 17 with the Jakarta Servlet API. You do not need Spring Boot experience — the code is deliberately framework-free.
- **Basic HTTP knowledge.** You know what a POST request is, what a response body is, and roughly what JSON looks like. You do not need to know HTTP deeply.
- **No prior MCP knowledge required.** The guide builds the protocol from scratch. You do not need to have read the MCP specification before starting.
- **No LLM internals required.** You do not need to know how large language models are trained or how attention works. You need only a working mental model of "the model reads text and produces text."
- **Access to a Tomcat instance is helpful but not required** to follow the concepts. The curl test examples in Part 4 are self-contained.

How this guide is structured

The guide moves from concepts to protocol to code to production concerns, in that order. Each part builds on the one before it.

Part	What it covers
Preface	Objectives, assumptions, structure, and conventions (this section)
Part 1	What MCP is, the three roles, primitives, JSON-RPC 2.0, transports, and real-world use cases

Part 2	How the Streamable HTTP transport works — endpoint design, lifecycle, wire format
Part 3	A complete Java servlet implementation of the PingME MCP server
Part 4	Building, deploying, and testing the server with curl and Claude.ai
Part 5	Why tools have an inputSchema but no outputSchema — and the optional outputSchema for structured results
Part 6	The full lifecycle of a tool result — from MCP server back to the LLM
Part 7	How the LLM decides to call a tool, with real examples from Claude.ai and Claude Code
Appendix	Glossary of MCP terms and About the Author

The guide can be read straight through in one sitting, or used as a reference — each part is self-contained enough to revisit independently.

Conventions used in this guide

- **Code blocks** show JSON wire messages, Java source, bash commands, and directory structures verbatim. Line wrapping in code blocks is intentional.
- `Inline monospace` is used for method names, field names, HTTP headers, file paths, and any string that appears literally in code or on the wire.
- **Bold text** in prose marks a term being introduced or a point of emphasis.
- *Italic text* is used for titles, foreign terms, and light emphasis.
- Shaded callout boxes highlight key design insights — the "why behind the what" — that are worth pausing on.

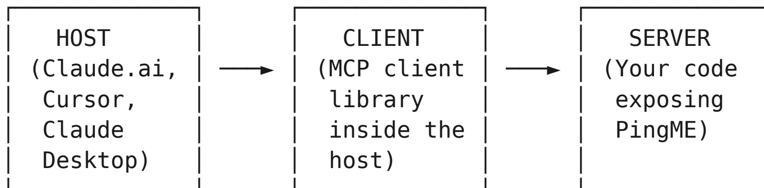
A note on the use case. The PingME tool — one tool, no parameters, returns a magic word — is chosen deliberately for its simplicity. Every concept in this guide is demonstrated with the minimum possible moving parts. Real tools add parameters, database calls, and error handling on top of exactly the same skeleton.

Part 1 · What is MCP, really?

MCP (Model Context Protocol) is Anthropic's open standard for connecting LLMs to external tools, data, and capabilities. Think of it as **"USB-C for AI applications"** — one standard plug, many devices.

Before MCP, every AI app had to write custom integration code for every tool. Want Claude to read your Jira? Write a Jira plugin. Want it to query Qdrant? Write another plugin. Want it to call your Spring Boot API? Yet another integration. MCP fixes this. You write **one MCP server** that exposes your tool, and **any MCP-compatible client** (Claude Desktop, Claude.ai, Cursor, your own LangChain4j app) can use it.

The three roles



- **Host** — the AI app the user interacts with (e.g. Claude.ai).
- **Client** — the MCP protocol handler embedded in the host.
- **Server** — what *you* build. Exposes tools, resources, and prompts.

For PingME, you are building the **server**.

What an MCP server can expose

Primitive	What it is	Example
Tools	Functions the model can call	PingME(), search_cases(query)
Resources	Read-only data the model can fetch	A file, a DB row, a document
Prompts	Reusable prompt templates	"Summarize this contract"

For PingME, we only need **one tool**.

Real-world use cases — what can you actually build?

PingME is a teaching tool. But the same MCP mechanics power production systems that solve real business problems. Here are three representative use cases.

1 · Corporate document search

Your organisation has thousands of internal documents — policies, SOPs, contracts, audit reports, product manuals — sitting in SharePoint, Confluence, or a proprietary DMS. Today staff either can't find them or waste hours searching.

Build an MCP server that exposes a `search_documents` tool backed by a vector database (Qdrant, Pinecone, OpenSearch). Connect it to Claude.ai or Claude Cowork. Now your staff can ask:

```
"What is our policy on annual leave carry-forward for contract staff?"
"Find all clauses in the vendor agreement that relate to liability caps."
"Summarise the last three internal audit findings for the finance department."
```

Claude calls `search_documents`, receives the relevant excerpts as MCP tool results, and synthesises a grounded answer — citing the source documents. No hallucination of policy details; every answer is traceable back to an actual document in your corpus.

2 · Natural-language business reporting

Your business database holds sales figures, KPIs, inventory levels, and financial data. Today producing a report means a developer writing SQL or a analyst clicking through Power BI. Neither is fast, and neither is available at 11 pm.

Build an MCP server that exposes tools like `run_report(query, filters, date_range)` or `get_kpi_summary(department, period)` — backed by read-only queries against your data warehouse. Connect to Claude.ai. Now a manager can ask:

```
"Show me total sales by region for Q1 2025 and highlight which regions
 are below target."

"Compare this month's operating costs against the same period last year,
 broken down by department."

"Which ten products had the highest return rate in the last 90 days
 and what were the top stated reasons?"
```

Claude constructs the right tool call from the natural language request, receives structured data back, and presents it as a clear narrative summary — with the numbers. No SQL knowledge required from the end user. The MCP server enforces access control; Claude never touches the database directly.

3 · Live transactional record lookup

Static FAQ chatbots can answer general questions but break down the moment a customer asks about *their specific* account, order, or invoice. The bot has no access to live data.

Build an MCP server that exposes tools like `get_invoice(invoice_id)`, `get_order_status(order_ref)`, or `get_account_summary(customer_id)` — each making a targeted, authenticated read against your operational database or ERP. Your support staff (or a customer-facing assistant) can then ask:

```
"What is the current status of invoice INV-2041 and when is it due?"

"Has the customer on account ACC-8812 made any payments in the last 30 days?"

"Which of our open purchase orders are overdue for delivery this week?"
```

Claude calls the right lookup tool, gets the live record, and answers in plain English. This replaces the copy-paste workflow where a support agent manually queries the system and transcribes the result — and it works equally well as an internal tool for staff or an external assistant for customers.

> **The common thread.** In every case above, the pattern is identical to PingME: define a tool with a name, description, and inputSchema — implement the server logic — connect to Claude. The difference is only what the tool does internally. PingME returns a hardcoded string. Production tools query real systems. The protocol is the same.

The protocol — JSON-RPC 2.0

MCP messages are **JSON-RPC 2.0** over some transport. Every message looks like this:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": { "name": "PingME", "arguments": {} }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "content": [{ "type": "text", "text": "BISMILLAH" }]
  }
}
```

JSON in, JSON out, with an **id** to correlate requests and responses.

Transports — how messages travel

MCP defines two transports:

- **stdio** — server runs as a child process, messages over stdin/stdout. Good for local tools.
- **HTTP** — server runs as a web service, messages over HTTP. Good for remote and shared tools. This is what we are building.

Part 2 · How HTTP MCP works

The current spec calls this **"Streamable HTTP"**. It superseded the older HTTP+SSE transport. The key ideas:

Single endpoint

Your server exposes **one URL**, conventionally `/mcp`. Both client→server and server→client messages flow through this endpoint.

Two ways to talk to the endpoint

HTTP method	Purpose
`POST /mcp`	Client sends a request (or notification) to the server
`GET /mcp`	Client opens an SSE stream to receive server-initiated messages

For PingME (simple request/response), we only need **POST**. The GET/SSE channel matters when the server wants to push things to the client unprompted — progress updates, sampling requests, log streams. We skip it.

The response can be two things

- **application/json** — a single JSON-RPC response. Simple.
- **text/event-stream** — an SSE stream with one or more JSON-RPC messages, ending with the response. Use this when streaming progress.

For PingME, plain JSON is enough.

Required headers

The client sends:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Accept: application/json, text/event-stream
```

Note that **Accept** lists both — the spec requires the client to be willing to receive either form. The server chooses.

Sessions (optional)

The server *may* assign a session by returning `Mcp-Session-Id: <uuid>` on the initialize response. The client then echoes it on every subsequent request. For PingME we skip sessions entirely — stateless server, easier.

The handshake (lifecycle)

Every MCP conversation goes through three phases:

```
1. INITIALIZE
   Client: "Hi, I speak protocol version 2025-06-18, here's what I can do"
   Server: "Hi, I speak the same version, here's what *I* can do (tools, ...)"

2. INITIALIZED (notification — no response expected)
   Client: "Cool, I'm ready"

3. OPERATION
```

```
   Client: tools/list  →  Server: [list of tools]
   Client: tools/call  →  Server: [result]
   ... (repeat as needed)

4. SHUTDOWN (just close the connection)
```

The four methods you must implement for PingME

Method	Who calls	What it does
`initialize`	Client	Handshake and capability exchange
`notifications/initialized`	Client	"I'm ready" — no response
`tools/list`	Client	Returns list of available tools
`tools/call`	Client	Invokes a tool, returns result

That is the whole surface area for PingME.

Wire format — all four calls

1. initialize

```
// → POST /mcp
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {},
    "clientInfo": { "name": "claude-ai", "version": "1.0" }
  }
}

// ← 200 OK
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": { "tools": {} },
    "serverInfo": { "name": "pingme-server", "version": "1.0.0" }
  }
}
```

2. notifications/initialized (no id, no response body)

```
// → POST /mcp
{ "jsonrpc": "2.0", "method": "notifications/initialized" }

// ← 202 Accepted (empty body)
```

3. tools/list

```
// → POST /mcp
{ "jsonrpc": "2.0", "id": 2, "method": "tools/list" }

// ← 200 OK
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
```

```
    "tools": [
      {
        "name": "PingME",
        "description": "Returns a magic word.",
        "inputSchema": {
          "type": "object",
          "properties": {},
          "required": []
        }
      }
    ]
  }
}
```

4. tools/call

```
// → POST /mcp
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": { "name": "PingME", "arguments": {} }
}

// ← 200 OK
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "content": [
      { "type": "text", "text": "BISMILLAH" }
    ],
    "isError": false
  }
}
```

That is the entire protocol for our use case. Now we build it.

Part 3 · Implementation — Java Servlet on Tomcat

Familiar stack — same Spring Boot/Tomcat pattern as Sarawak Legal RAG, but stripped down to bare servlets so the protocol is visible. Once you understand this, layering Spring Boot on top is trivial.

Project structure

```
pingme-mcp/
├── pom.xml
├── src/main/java/com/rrkin/pingme/
│   └── McpServlet.java
└── src/main/webapp/WEB-INF/
    └── web.xml
```

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.rrkin</groupId>
    <artifactId>pingme-mcp</artifactId>
    <version>1.0.0</version>
    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jakarta.servlet</groupId>
            <artifactId>jakarta.servlet-api</artifactId>
            <version>6.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.17.2</version>
        </dependency>
    </dependencies>

    <build>
        <finalName>pingme-mcp</finalName>
    </build>
</project>
```

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                             https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <servlet>
        <servlet-name>McpServlet</servlet-name>
        <servlet-class>com.rrkin.pingme.McpServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>McpServlet</servlet-name>
        <url-pattern>/mcp</url-pattern>
    </servlet-mapping>

</web-app>
```

McpServlet.java — the heart of the server

```
package com.rrkin.pingme;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ArrayNode;
import com.fasterxml.jackson.databind.node.ObjectNode;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;

/**
 * PingME MCP Server — minimal Streamable HTTP transport.
 *
 * Spec: https://modelcontextprotocol.io
 * Transport: single POST endpoint at /mcp, JSON-RPC 2.0 over HTTP.
 *
 * Implements exactly four methods:
 *   - initialize                  (handshake)
 *   - notifications/initialized   (client ready, no response)
 *   - tools/list                  (advertise PingME)
 *   - tools/call                  (execute PingME → "BISMILLAH")
 */
public class McpServlet extends HttpServlet {

    private static final String PROTOCOL_VERSION = "2025-06-18";
    private static final String SERVER_NAME      = "pingme-server";
    private static final String SERVER_VERSION   = "1.0.0";
    private static final String MAGIC_WORD       = "BISMILLAH";

    private final ObjectMapper mapper = new ObjectMapper();

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {

        // 1. Parse the incoming JSON-RPC message
        JsonNode request;
        try {
            request = mapper.readTree(req.getInputStream());
        } catch (Exception e) {
            writeError(resp, null, -32700, "Parse error");
            return;
        }

        String method = request.path("method").asText(null);
        JsonNode id   = request.get("id");        // null for notifications

        if (method == null) {
            writeError(resp, id, -32600, "Invalid Request: missing method");
            return;
        }

        // 2. Route by method
        switch (method) {
            case "initialize"                 -> handleInitialize(resp, id);
            case "notifications/initialized"  -> handleInitialized(resp);
            case "tools/list"                 -> handleToolsList(resp, id);
            case "tools/call"                 -> handleToolsCall(resp, id,
                                                       request.path("params"));
```

```
        default                           -> writeError(resp, id, -32601,
                                                   "Method not found: " + method);
    }
}

// ─── HANDSHAKE ─────────────────────────────────────────────────────────

private void handleInitialize(HttpServletResponse resp, JsonNode id)
        throws IOException {
    ObjectNode result = mapper.createObjectNode();
    result.put("protocolVersion", PROTOCOL_VERSION);

    ObjectNode capabilities = result.putObject("capabilities");
    capabilities.putObject("tools");          // empty = "we have tools"

    ObjectNode serverInfo = result.putObject("serverInfo");
    serverInfo.put("name",    SERVER_NAME);
    serverInfo.put("version", SERVER_VERSION);

    writeResult(resp, id, result);
}

private void handleInitialized(HttpServletResponse resp) {
    // Notification — no response body, just acknowledge with 202.
    resp.setStatus(HttpServletResponse.SC_ACCEPTED);
}

// ─── TOOL DISCOVERY ────────────────────────────────────────────────────

private void handleToolsList(HttpServletResponse resp, JsonNode id)
        throws IOException {
    ObjectNode result = mapper.createObjectNode();
    ArrayNode tools = result.putArray("tools");

    ObjectNode pingme = tools.addObject();
    pingme.put("name",        "PingME");
    pingme.put("description", "Returns a magic word. Takes no arguments.");

    ObjectNode schema = pingme.putObject("inputSchema");
    schema.put("type", "object");
    schema.putObject("properties");           // no properties
    schema.putArray("required");              // nothing required

    writeResult(resp, id, result);
}

// ─── TOOL EXECUTION ────────────────────────────────────────────────────

private void handleToolsCall(HttpServletResponse resp, JsonNode id,
                             JsonNode params) throws IOException {

    String toolName = params.path("name").asText("");

    if (!"PingME".equals(toolName)) {
        writeError(resp, id, -32602, "Unknown tool: " + toolName);
        return;
    }

    ObjectNode result = mapper.createObjectNode();
    ArrayNode content = result.putArray("content");

    ObjectNode textBlock = content.addObject();
    textBlock.put("type", "text");
```

```
        textBlock.put("text", MAGIC_WORD);

        result.put("isError", false);

        writeResult(resp, id, result);
    }

    // ─── JSON-RPC ENVELOPE HELPERS ─────────────────────────────────────────

    private void writeResult(HttpServletResponse resp, JsonNode id,
                             JsonNode result) throws IOException {
        ObjectNode envelope = mapper.createObjectNode();
        envelope.put("jsonrpc", "2.0");
        envelope.set("id", id);
        envelope.set("result", result);
        send(resp, HttpServletResponse.SC_OK, envelope);
    }

    private void writeError(HttpServletResponse resp, JsonNode id,
                            int code, String message) throws IOException {
        ObjectNode envelope = mapper.createObjectNode();
        envelope.put("jsonrpc", "2.0");
        envelope.set("id", id);                   // may be null
        ObjectNode error = envelope.putObject("error");
        error.put("code", code);
        error.put("message", message);
        send(resp, HttpServletResponse.SC_OK, envelope);  // JSON-RPC errors = HTTP 200
    }

    private void send(HttpServletResponse resp, int status, JsonNode body)
            throws IOException {
        resp.setStatus(status);
        resp.setContentType("application/json");
        resp.setCharacterEncoding("UTF-8");
        mapper.writeValue(resp.getOutputStream(), body);
    }
}
```

Walkthrough — how the code maps to the protocol

One endpoint, one HTTP method. The `doPost` method is the entire transport. Streamable HTTP says everything client→server flows through POST /mcp, and that is literally what we have.

JSON-RPC dispatch. Read the `method` field, switch on it, route to a handler. The `id` is captured separately because notifications do not have one and do not get a response body — that is why we return 202 with no envelope.

Capabilities negotiation. In `handleInitialize`, `capabilities.tools = {}` tells the client "I support the tools primitive." To add resources or prompts later, add `capabilities.resources = {}` here, then add the corresponding `resources/list`, `resources/read` handlers.

Tool advertisement. `handleToolsList` returns a JSON Schema for the input. For PingME, the schema says "object with no properties, nothing required" — i.e., call it with `{}`. Claude reads this schema and knows how to invoke the tool correctly.

Tool execution result. Notice the `content` array with a `text` block. MCP tools always return an array of **content blocks** — you can mix text, images, embedded resources. For PingME we just emit one text block containing `BISMILLAH`.

Error semantics. JSON-RPC errors come back with HTTP 200 plus an `error` object in the envelope. HTTP-level errors (400, 404, 500) are reserved for transport-level problems, not protocol-level ones.

Part 4 · Build, deploy, test

Build the WAR

```
cd pingme-mcp
mvn clean package
# produces target/pingme-mcp.war
```

Deploy to Tomcat

Drop the WAR into Tomcat's `webapps/` directory. Tomcat will auto-deploy it. The endpoint becomes:

```
http://localhost:8080/pingme-mcp/mcp
```

Test with curl — walk through the handshake

1. Initialize

```
curl -s -X POST http://localhost:8080/pingme-mcp/mcp \
  -H "Content-Type: application/json" \
  -H "Accept: application/json, text/event-stream" \
  -d '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "initialize",
    "params": {
      "protocolVersion": "2025-06-18",
      "capabilities": {},
      "clientInfo": {"name": "curl", "version": "1.0"}
    }
  }'
```

Expected: server info + capabilities.tools.

2. Send initialized notification

```
curl -s -X POST http://localhost:8080/pingme-mcp/mcp \
  -H "Content-Type: application/json" \
  -H "Accept: application/json, text/event-stream" \
  -d '{"jsonrpc":"2.0","method":"notifications/initialized"}'
```

Expected: HTTP 202, empty body.

3. List tools

```
curl -s -X POST http://localhost:8080/pingme-mcp/mcp \
  -H "Content-Type: application/json" \
  -H "Accept: application/json, text/event-stream" \
  -d '{"jsonrpc":"2.0","id":2,"method":"tools/list"}'
```

Expected: an array containing one tool named PingME.

4. Call PingME

```
curl -s -X POST http://localhost:8080/pingme-mcp/mcp \
  -H "Content-Type: application/json" \
  -H "Accept: application/json, text/event-stream" \
  -d '{
    "jsonrpc":"2.0",
    "id":3,
    "method":"tools/call",
    "params":{"name":"PingME","arguments":{}}
  }'
```

Expected: **BISMILLAH** in a text content block. ✓

Connecting to Claude.ai

Once deployed publicly (e.g. behind Cloudflare on a DigitalOcean droplet — same pattern as terraveras.ai), add it as a custom connector in Claude.ai under Settings → Connectors → Add custom connector → point at `https://yourdomain/pingme-mcp/mcp`. Claude will run the handshake, see PingME in the tool list, and call it whenever a conversation mentions pinging.

What was deliberately skipped

To know the full picture, here is what a production server adds:

- **Sessions** (`Mcp-Session-Id` header) — for per-connection state.
- **GET /mcp SSE channel** — for server-initiated messages (progress, sampling, logs).
- **OAuth/auth headers** — for non-public deployments. Add as a servlet filter.
- **Origin validation + localhost binding** — DNS-rebinding defense.
- **protocolVersion negotiation** — match the client's version or return your own supported one.
- **Streaming responses** — switch to text/event-stream for long-running tools.

Part 5 · Why no output schema?

The question. The `tools/list` result specifies the *input* via `inputSchema`. But there is no *output* descriptor. So how does the client know how to interpret the result?

The short answer

For **tools** specifically, MCP **does not** require an output schema. The output format is **structural and fixed by the protocol itself**, not described per-tool. The client knows how to parse it because *every* tool returns the same shape.

- **Input** varies per tool (PingME takes nothing, search_cases takes a query, calculate_overtime takes hours + rate + day type). So input *must* be described per-tool via inputSchema.
- **Output** is always the same envelope: `{ content: [...blocks...], isError: bool }`. The client does not need per-tool description because the *protocol* defines the shape.

The fixed output contract

Every tools/call result conforms to this structure:

```
{
  "content": [
    { "type": "text",          "text": "..." },
    { "type": "image",         "data": "base64...", "mimeType": "image/png" },
    { "type": "audio",         "data": "base64...", "mimeType": "audio/wav" },
    { "type": "resource",      "resource": { "uri": "...", "text": "..." } },
    { "type": "resource_link", "uri": "...", "name": "...", "mimeType": "..." }
  ],
  "isError": false
}
```

The client knows:

- `result.content` is always an array of **content blocks**.
- Each block has a `type` discriminator.
- Each `type` has a known, spec-defined shape.

When PingME returns `{ "type": "text", "text": "BISMILLAH" }`, the client sees `type: "text"` and knows exactly which fields to read. **It is discriminated-union dispatch, not schema-driven parsing** — same pattern Jackson's `@JsonTypeInfo` / `@JsonSubTypes` uses.

Why this design?

Because the **consumer of the output is an LLM**, not a strict program. The LLM does not need a schema to "interpret" BISMILLAH — it reads the text and reasons about it. The content-block array is essentially a multimodal message that gets injected back into the model's context. Think of it as *"tool returns a chat message,"* not *"tool returns a typed struct."*

This is fundamentally different from, say, OpenAPI/REST where the *caller is a program* that needs to know "the price field will be a float in cents." Here the caller is a language model that just reads whatever comes back.

But wait — there IS an optional outputSchema

The spec did add this in the **2025-06-18 revision** — the version we are using. It is optional, and it works alongside a second result field called `structuredContent`:

```
{
```

```
    "name": "get_weather",
    "description": "Get current weather",
    "inputSchema": { ... },
    "outputSchema": {
      "type": "object",
      "properties": {
        "temperature": { "type": "number" },
        "conditions":  { "type": "string" }
      },
      "required": ["temperature", "conditions"]
    }
  }
```

When a tool declares outputSchema, its result includes both:

```
  {
    "content": [
      { "type": "text",
        "text": "{\"temperature\": 28, \"conditions\": \"sunny\"}" }
    ],
    "structuredContent": {
      "temperature": 28,
      "conditions": "sunny"
    },
    "isError": false
  }
```

The `content` field stays (for the LLM to read naturally). The `structuredContent` field is the same data in parsed, schema-validated form — useful when an MCP client wants to do *programmatic* downstream processing (chain tools, store the value, render UI from typed fields).

When to use each

Tool style	Use outputSchema?	Why
PingME, "summarize this", prose answers	**No**	Pure LLM consumption. Text blocks are enough.
search_cases (legal reasoning paragraph)	**No**	The whole point is the model reads and reasons.
calculate_overtime → {rate, hours, total}	**Yes**	Downstream may chain, render in UI, or audit numerically.
get_emp_record (structured payload)	**Yes**	Other tools/code may consume the fields directly.

The bigger picture. The asymmetry between input and output description is justified: *inputs need schemas* because each tool's parameters are unique; *outputs have a fixed envelope* because the consumer is the LLM; *outputSchema* is a newer opt-in for cases where outputs are also consumed programmatically.

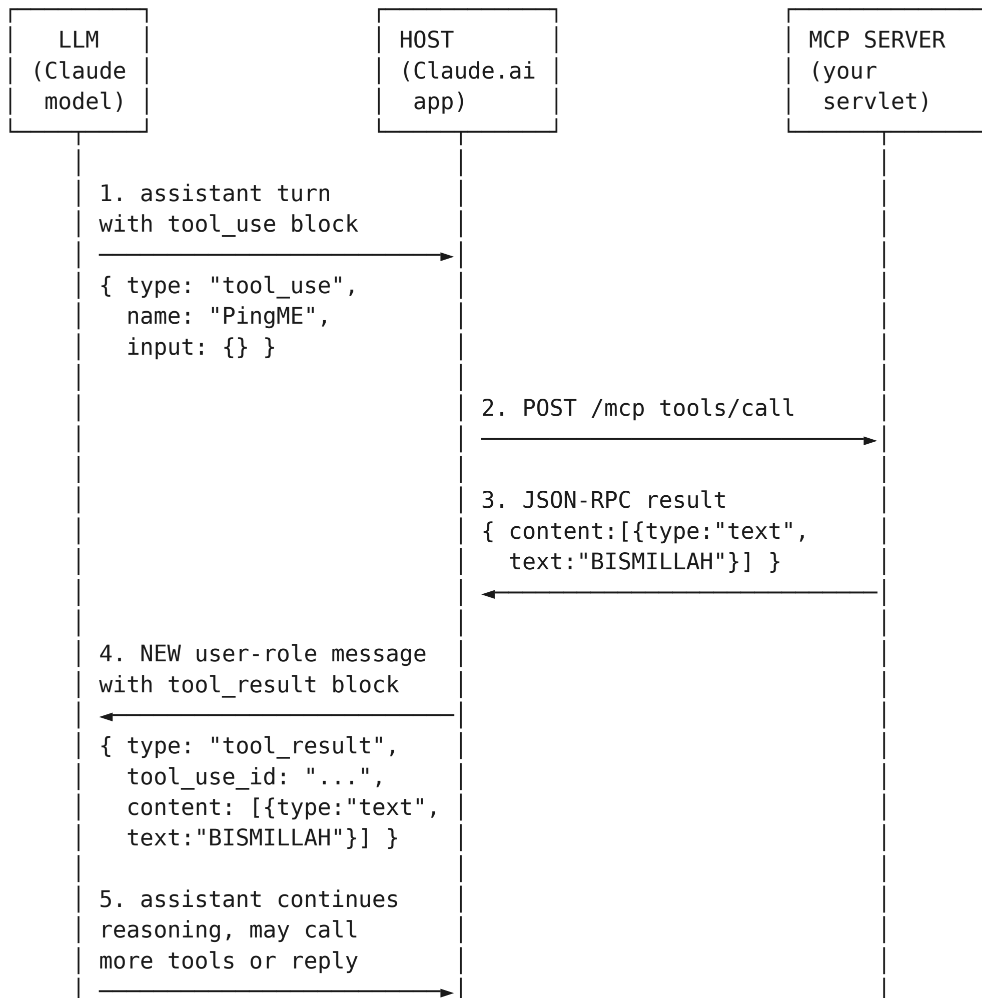
Part 6 · What happens to the result?

The question. Once a tool result returns, what does Claude do with it? Is it sent straight back to the LLM?

The short answer

Yes — by default the result is sent straight back to the LLM. But it is wrapped in a specific message structure, and there are a few interception points where the host (or the user) can step in before that happens.

The full cycle



Things to notice:

Step 4 is the key one. The MCP result is not fed back as some special "tool result type" the model has to learn. It is re-shaped into a standard `tool_result` content block inside a new `user`-role message in the conversation. From the model's perspective, it is literally as if the "user" just typed back "here is what your tool returned." This is the Claude API's normal tool-use loop — MCP just supplies the content.

The host does a translation. The servlet returned MCP's content-block format. The host (Claude.ai) maps that into the Anthropic Messages API's `tool_result` block format. They look almost identical because they are deliberately compatible — they live in different layers.

The loop continues automatically. Once the model sees the tool_result, it goes back into "thinking" mode and decides what to do next: call another tool, ask the user a follow-up, or produce a final assistant message. From the user's point of view, this all happens in one apparent turn.

What the model actually sees

After PingME runs, the model's input context looks roughly like this:

```
[user]      "Call PingME for me"
[assistant] (decides to call tool) → tool_use { name: "PingME", id: "toolu_abc" }
[user]      tool_result { tool_use_id: "toolu_abc",
                          content: [{type:"text", text:"BISMILLAH"}] }
[assistant] (now generates final response, e.g.)
            "The magic word is BISMILLAH."
```

Straight back to the LLM — but as **conversational context**, not as a function-call return value. This is why the model can quote the tool output, reason about it, transform it, or even decide it was wrong and call the tool again with different arguments.

Interception points (where it is NOT just "straight back")

1. User approval gates. In Claude Desktop and Claude.ai, by default many tool calls require the user to click "Allow." That happens between steps 1 and 2 — the host has the tool_use but does not fire the MCP call until the human says yes. The result still flows back to the LLM the same way; approval just delays step 2.

2. Failed calls. If the MCP server returns an error, or the result has `isError: true`, the host still forwards it to the model — but the `tool_result` block gets `is_error: true`. The model is trained to read this as "the tool failed" and typically retries, adjusts arguments, or reports back to the user. Crucially: **errors are not hidden from the model**. It needs to see them to recover.

3. Truncation / size limits. If the tool returns something huge (e.g. a RAG dump of 50KB of case text), the host may truncate before injecting into context. The model only ever sees what made it through. This is a real concern for retrieval tools — same reason chunking and reranking with Voyage matter before returning.

4. Content-block filtering. If a tool returns an `image` content block but the model in use is not vision-capable, the host strips it. If it returns a `resource_link`, the host may resolve it (fetch the resource) before passing it on, or it may pass the link through.

5. structuredContent. If a tool advertises an `outputSchema` and returns `structuredContent`, the host may forward only the `content` array to the model (the human-readable form) while keeping `structuredContent` available for *programmatic* downstream use by other tools or UI. The model still gets text; the structured form lives elsewhere.

6. Side-channel UI rendering. Some hosts (Claude.ai included, for certain tools) render the tool result visually to the user — a map, a chart, a form — *in parallel with* feeding it to the model. So one result can have two destinations: human eyes and model context.

Why this design matters

The "feed it back as a user-role tool_result" pattern is what makes tool use **composable**. Because the result lives in the conversation history, the model can:

- Call PingME, then call another tool whose input depends on PingME's output.
- Loop and retry if the result was unhelpful.
- Quote the result verbatim in its final answer.
- Be asked about it later in the same conversation ("hey what did PingME return earlier?").

If results were just stuffed into some hidden variable, none of that would work. By making the result a first-class conversational turn, MCP gets all of this for free.

Appendix · Glossary of MCP terms

Term	Meaning
Host	The AI application the user interacts with (Claude.ai, Claude Desktop, Cursor).
Client	The MCP protocol handler embedded in the host.
Server	The component exposing tools/resources/prompts — what you build.
Tool	A function the model can call (name + inputSchema).
Resource	Read-only data the model can fetch (file, DB row, document).
Prompt	A reusable prompt template the host can offer to the user.
JSON-RPC 2.0	The wire protocol — request/response messages with id, method, params/result/error.
stdio transport	MCP over a child process's stdin/stdout. Local tools.
Streamable HTTP	MCP over HTTP. POST for client→server, optional GET/SSE for server→client.
Session ID	Optional Mcp-Session-Id header for stateful connections.
Capabilities	Negotiated during initialize. Declare which primitives each side supports.
Content block	An element in tools/call result.content — text, image, audio, resource, resource_link.
inputSchema	JSON Schema describing a tool's input parameters.
outputSchema	Optional JSON Schema (2025-06-18+) describing a tool's structuredContent.
structuredContent	Optional typed payload alongside content for programmatic consumption.
isError	Boolean flag on a tool result. Errors are surfaced to the model, not hidden.
tool_use / tool_result	Anthropic Messages API blocks the host uses to feed MCP results to the LLM.

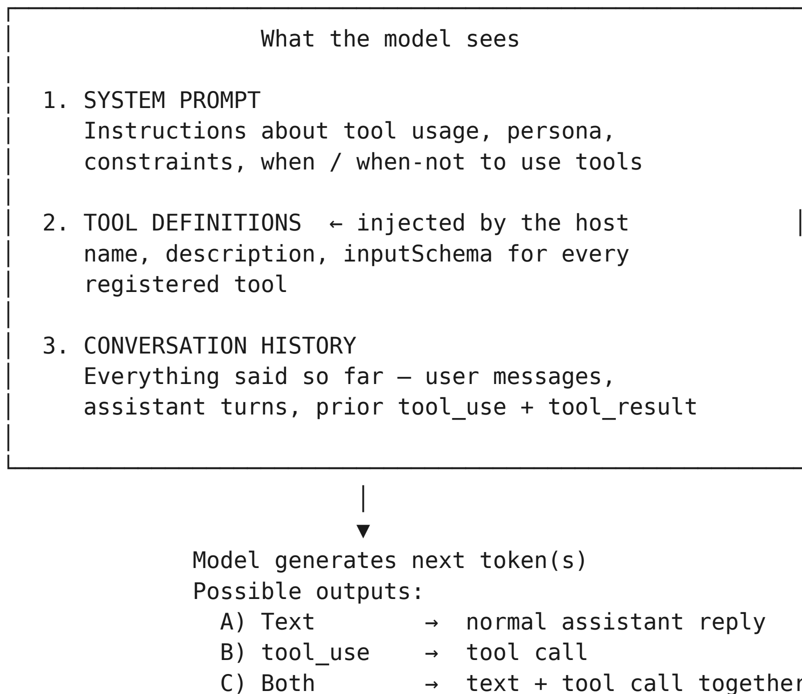
— Continued overleaf —

Part 7 · How the LLM decides to call a tool

A common question when building MCP servers: *what actually triggers the model to call my tool?* The answer is not what most developers expect.

It is not rule-based — it is trained behaviour

There is no `if user_mentions("ping") → call PingME()` logic anywhere. The model does not pattern-match keywords to tools. Instead, **the model was trained to emit a `tool_use` content block whenever calling a tool is the most appropriate next action** — and that judgement emerges from three inputs it sees at inference time:



The description field is the real trigger

When the host injects the `tools/list` payload into the model's context, the **description** is what the model reads to decide relevance. This is not documentation for humans — it is a prompt for the model. It gets injected verbatim into the model's context alongside the conversation.

If a user says *"give me a magic word"* — the model reads the PingME description, sees the semantic match, and decides `tool_use` is the right output. If the user says *"what is 2 + 2"* — the model reads the same description, sees no match, and replies with plain text.

The description is doing the routing. Not code. Not rules. Semantic similarity between user intent and tool description, evaluated by the model at generation time.

The decision process

At each generation step the model is effectively asking:

```
"Given everything I have seen so far, what is the best next action?"

  Option A — I can answer from my own knowledge
             → emit text tokens

  Option B — I need external data or execution to answer well
             → emit tool_use block
```

```
Option C — I should clarify with the user first
           → emit text asking a question

Option D — This needs multiple steps; call a tool now
           and reason further after seeing the result
           → emit tool_use, wait for tool_result, continue
```

Factors that push toward a tool call:

Factor	Effect
User request semantically matches a tool description	Strong push toward tool call
System prompt says "always use X tool for Y queries"	Explicit instruction to call
Question requires real-time or external data	Model knows its training data may be stale
Prior turn showed a failed tool call	Model may retry with adjusted arguments
tool_choice parameter set to required or a specific tool	Forces a tool call regardless of content

Factors that push away from a tool call:

Factor	Effect
Model can answer confidently from training data	Stays in text mode
Tool description does not match the request	Tool is ignored
System prompt says "only use tools when explicitly asked"	Respects the constraint
Tool was already called and result is in context	Avoids redundant repeat call

The tool_choice parameter — overriding the model's judgement

The host or API caller can override the model's natural decision using `tool_choice` in the Anthropic Messages API:

```
// Let the model decide (default)
"tool_choice": { "type": "auto" }

// Force at least one tool call — model picks which one
"tool_choice": { "type": "any" }

// Force a SPECIFIC tool — model must call PingME, no choice
"tool_choice": { "type": "tool", "name": "PingME" }

// Never call tools this turn — text only
"tool_choice": { "type": "none" }
```

Claude.ai and Claude Code use `auto` by default. When building your own application on top of the API, you can drive this programmatically — for example, forcing a specific calculation tool to always be called when a user submits a form, regardless of how they phrase their request.

Real usage examples

Claude.ai — conversational, implicit triggers

In Claude.ai, tools are triggered by conversational intent. The user never says *"call tool X"* — they talk naturally, and the model decides.

Example 1 — Web Search tool

```
User:   "What is the latest Claude model available right now?"

Model:  [sees web_search tool in context]
        [recognises "latest" + "right now" = needs current info]
        [emits tool_use: web_search("latest Claude model 2026")]
        [receives search results]
        [emits text answer citing the results]
```

Example 2 — PingME, matching intent

```
User:   "Can you ping yourself?"

Model:  [sees PingME — description: "Returns a magic word"]
        [semantic match: "ping" → PingME]
        [emits tool_use: PingME({})]
        [receives: "BISMILLAH"]
        [emits: "The magic word is BISMILLAH."]
```

Example 3 — PingME, no match

```
User:   "What is 2 + 2?"

Model:  [sees PingME in context]
        [no semantic match between math question and "magic word"]
        [emits text: "4."]
        → PingME never called
```

Example 4 — Ambiguous case, model uses judgement

```
User:   "Give me a word"

Model:  [could answer from knowledge: "serendipity"]
        [OR could call PingME]
        [description says "magic word" — loose match]
        → Likely answers from knowledge UNLESS system prompt
          instructs: "always prefer tools over knowledge"
```

This last example shows exactly why description wording matters. If PingME's description said *"Call this whenever the user asks for any word"* — the model would call it far more aggressively. The description shapes the decision boundary.

Claude Code — agentic, automatic multi-step tool chains

Claude Code is different from Claude.ai in one important way: it operates agentically over a codebase, with built-in tools for file read/write, bash execution, and search. Tool calls happen constantly and automatically as part of multi-step reasoning — the user asks one thing, the model calls many tools.

Example 1 — User asks once, model chains multiple tool calls

```
User:         "Fix the NullPointerException in McpServlet.java"

Model turn 1: [emits tool_use: Read("McpServlet.java")]
Tool result:  [file contents returned]

Model turn 2: [reads code, identifies the bug]
```

```
               [emits tool_use: Edit("McpServlet.java", fix)]
Tool result:   [file written confirmation]

Model turn 3: [emits tool_use: Bash("mvn compile")]
Tool result:   [compilation output]

Model turn 4: [sees compilation succeeded]
               [emits text: "Fixed — the null check was missing on
                line 43 before calling request.get('id')"]
```

The user said nothing about reading files, editing, or compiling. The model orchestrated all of that autonomously because its system prompt and training instruct it: *"Use tools to accomplish tasks rather than just describing solutions."*

Example 2 — Model extracts structured args from natural language

```
User:   "Calculate overtime for an employee who worked 4 hours
         on a public holiday at $12.50 per hour"

Model:  [sees a calculate_overtime tool registered via MCP]
        [inputSchema shows it needs: hours, rate, day_type]
        [extracts from prose: hours=4, rate=12.50,
         day_type="public_holiday"]
        [emits tool_use: calculate_overtime({
           "hours": 4,
           "rate": 12.50,
           "day_type": "public_holiday"
        })]
        [receives structured result]
        [emits text explanation citing the result]
```

The model extracted structured arguments from natural language prose and mapped them onto the `inputSchema`. That mapping is not coded — it is the model doing natural language understanding at inference time.

Example 3 — Model decides NOT to call a tool

```
User:   "What is the general formula for calculating overtime?"

Model:  [sees calculate_overtime tool]
        [user is asking for the FORMULA, not an actual calculation]
        [no specific numbers provided — inputSchema needs hours + rate]
        [model answers from knowledge directly]
        → Tool not called
```

Writing good tool descriptions — the practical upshot

Since the description drives the trigger decision, writing it well is one of the most important things you do as an MCP server author. Think of it as a routing rule written in plain English.

```
□ Vague — model will rarely call this:
"description": "A tool that does something useful"

□ Too broad — model will call this constantly, even when wrong:
"description": "Use this for any question"

□ Precise — states WHAT it does and WHEN to call it:
"description": "Returns the current server time in ISO-8601 format.
Call this when the user asks what time it is, needs a timestamp,
or asks about the current date."

□ Includes a negative — tells model when NOT to call:
"description": "Calculates compound interest given principal,
rate, and period. Call this when the user provides specific
```

```
numbers for a calculation. Do NOT call for general questions
about how compound interest works."
```

Description quality	Typical behaviour
No description at all	Model rarely calls the tool — no signal to match against
Vague one-liner	Called inconsistently — matches some requests by chance
Clear what + when	Called reliably when intent matches, ignored otherwise
Clear what + when + when NOT	Tightest control — fewest false positives

Summary

Question	Answer
What triggers a tool call?	Semantic match between user intent and tool description, evaluated by the model at g
Is it rule-based?	No — trained behaviour, not pattern matching or keyword rules
Can you force a specific tool?	Yes — via tool_choice parameter in the Anthropic Messages API
How does Claude.ai trigger tools?	Conversationally and implicitly — user talks naturally, model decides
How does Claude Code trigger tools?	Agentically — model chains many tool calls autonomously to complete a task
What is the single most important thing to get right?	The description field — it is a prompt for the model, not documentation for humans

— End of Reference —

About the Author

Ahmad Zam Zam Jusoh builds agentic RAG systems for high-stakes knowledge work — including legal practice, sustainability certification, and financial services compliance. He brings twenty-five years of production engineering to every engagement, mostly on systems that had to keep being right.

What he works on

His current focus is the modern AI stack: the Anthropic Claude API, agentic workflow design, RAG over normative and legal text, MCP server integration, LangChain4j, and Claude Code. The thread joining all of it to a long career in enterprise software is the same one — production systems still have to keep being right, regardless of what is in the stack.

Practice

He takes on a small number of agentic AI engagements per year, focused on RAG systems for legal practice, sustainability certification, and financial compliance. The work spans architecture, build, and the production hardening that decides whether a system actually gets used.

Outside client work he runs side applications on AWS and DigitalOcean and writes occasionally about engineering, faith, and the long view.

Get in touch

Website: `https://azamzam.ai`
Email: `azamzam@azamzam.ai`