



# Hello, MCP World!

**Daniela Petruzalek**

Developer Relations Engineer @ Google Cloud

@danicat83



# Who am I?

**Developer Relations Engineer** at Google, originally from Brazil 🇧🇷, but living in the UK 🇬🇧 since 2019.

My background is in backend and data engineering. Have been doing Go development (on-and-off) since 2017.



Self-introduction

# Agenda

What is the Model Context Protocol

Architecture and Terminology

MCP servers for coding


Go support for MCPs

Write your own!

HOW STANDARDS PROLIFERATE:  
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)



This is maybe the first time in the industry where this joke doesn't apply, as everyone quickly converged to mcp, nevertheless I love it anyway and it is usually true!

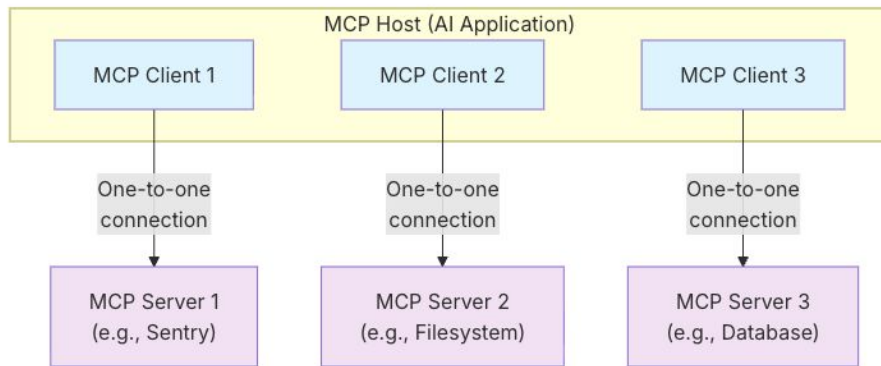


**MCP is an open protocol that standardizes how applications provide context to large language models (LLMs). (...) MCP enables you to build agents and complex workflows on top of LLMs and connects your models with the world.**

The official analogy Anthropic uses is the USB-C for AI Models, but I prefer to compare it to HTTP REST APIs

Talk about how MCP is the new thing - while we spent over the past 15, maybe 20 years making everything “API-first” to be able to connect systems and automate, we are going to spend probably the next 10 or so retrofitting everything to speak “AI Native” and the MCP is the secret sauce for that (at least until they come up with a new standard)

# MCP Architecture



<https://modelcontextprotocol.io/docs/learn/architecture>

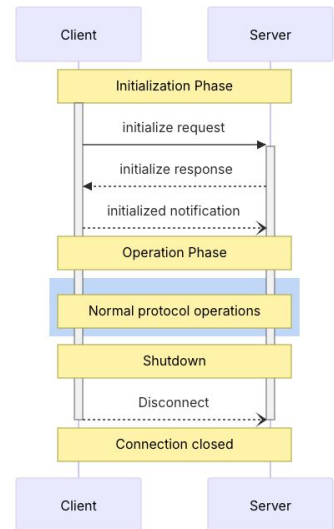
Host: your favourite IDE, coding agent, etc... the thing that spawns the clients

Client: talks to a server :)

Server: talks to a client :)

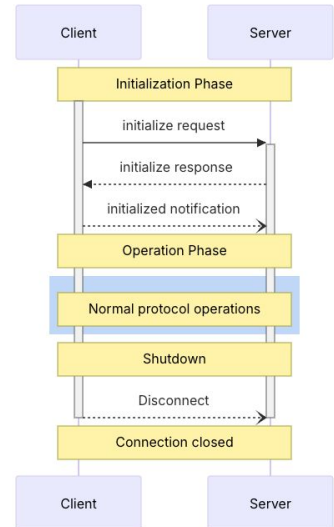
# MCP Layers

- **Data layer:** JSON-RPC based protocol for client-server comms
- **Transport layer:** communication channels between client and servers
  - Standard IO
  - Streamable HTTPS
  - HTTP+SSE: deprecated due to security concerns



# Initialization Flow

```
{"jsonrpc": "2.0", "id": 1, "method": "initialize",  
"params": {"protocolVersion": "2025-06-18"}}  
  
{"jsonrpc": "2.0", "method": "notifications/initialized",  
"params": {}}  
  
{"jsonrpc": "2.0", "id": 2, "method": "tools/list",  
"params": {}}
```



This is just an example of how JSON-RPC messages look like, in this case describing the initialization handshake between client and server.

I often use plain requests like this during testing to ensure the implementation is sound. Helps while “vibe coding” an mcp server.



# Building Blocks

Building Block	Purpose	Who Controls It	Real-World Example
Tools	For AI actions	Model-controlled	Search flights, send messages, create calendar events
Resources	For context data	Application-controlled	Documents, calendars, emails, weather data
Prompts	For interaction templates	User-controlled	"Plan a vacation", "Summarize my meetings", "Draft an email"

Sometimes also referred to as “Primitives” or “Server Concepts”

# Tools

Tools enable AI models to perform actions through server-implemented functions

Tools demo: open the CLI and give an overview of the currently configured MCP, and make a few tool calls (e.g. code review)

● **godoc** - Ready (4 tools, 1 prompt)

Tools:

- code\_review
- godoc
- scalpel
- scribble

Prompts:

- import\_this



**GoDoctor**

<https://github.com/danicat/godoc>

Show Gemini CLI interface, use godoc to show documentation for net/http, then show documentation for modelcontextprotocol/go-sdk/mcp

main.go

```
func addTools(server *mcp.Server, apiKeyEnv string) {
    // Register the go-doc tool unconditionally.
    godoc.Register(server)
    scribble.Register(server)
    scalpel.Register(server)

    // Register the code_review tool only if an API key is available.
    codereview.Register(server, os.Getenv(apiKeyEnv))
}
```

godoc.go

```
// Register registers the go-doc tool with the server.
func Register(server *mcp.Server) {
    mcp.AddTool(server, &mcp.Tool{
        Name:        "godoc",
        Description: "Retrieves Go documentation for a specified package or symbol.",
        Handler:      getDocHandler,
    })
}

// GetDocParams defines the input parameters for the go-doc tool.
type GetDocParams struct {
    PackagePath string `json:"package_path"`
    SymbolName  string `json:"symbol_name,omitempty"`
}
```

godoc.go

```
func getDocHandler(ctx context.Context, s *mcp.ServerS
    if pkgPath == "" { ...
    }

    args := []string{"doc"}
    if symbolName == "" { ...
    } else { ...
    }

    cmd := exec.CommandContext(ctx, "go", args...)
    var out bytes.Buffer
    cmd.Stdout = &out
    cmd.Stderr = &out
    if err := cmd.Run(); err != nil { ...
    }

    docString := strings.TrimSpace(out.String())
    if docString == "" { ...
    }

    return &mcp.CallToolResult{
        Content: []mcp.Content{
            &mcp.TextContent{Text: docString},
        },
    }, nil
}
```

## Prompts

Prompts provide reusable templates. They allow MCP server authors to provide parameterized prompts for a domain, or showcase how to best use the MCP server.

Prompts demo: again a brief tour of the CLI, show a few slash commands and finish with `/reflect`

● **speedgrapher** - Ready (1 tool, 11 prompts)

Tools:

- fog

Prompts:

- context
- expand
- haiku
- interview
- localize
- outline
- publish
- readability
- reflect
- review
- voice



<https://github.com/danicat/speedgrapher>

```

func Haiku() *mcp.Prompt {
    return &mcp.Prompt{
        Name:      "haiku",
        Description: "Creates a haiku about a given topic, or infers the topic from the current conversation.",
        Arguments: []*mcp.PromptArgument{
            {
                Name:      "topic",
                Description: "The topic for the haiku. If not provided, the model will infer it from the conversation.",
                Required:   false,
            },
        },
    }
}

func HaikuHandler(ctx context.Context, session *mcp.ServerSession, params *mcp.GetPromptParams) (*mcp.GetPromptResult, error) {
    prompt := "Write a haiku about the main subject of our conversation."
    if topic, ok := params.Arguments["topic"]; ok && topic != "" {
        prompt = fmt.Sprintf("The user wants to have some fun and has requested a haiku about the following topic: %s", topic)
    }

    return &mcp.GetPromptResult{
        Messages: []*mcp.PromptMessage{
            {
                Role: "user",
                Content: &mcp.TextContent{
                    Text: prompt,
                },
            },
        },
    }, nil
}

```



# Resources

Resources expose data from files, APIs, databases, or any other source that an AI needs to understand context.

```
{
  "uriTemplate": "weather://forecast/{city}/{date}",
  "name": "weather-forecast",
  "title": "Weather Forecast",
  "description": "Get weather forecast for any city and date",
  "mimeType": "application/json"
}

{
  "uriTemplate": "travel://flights/{origin}/{destination}",
  "name": "flight-search",
  "title": "Flight Search",
  "description": "Search available flights between cities",
  "mimeType": "application/json"
}
```

Mostly designed for data retrieval but still haven't seen those used in the wild. Can't opionate much, but I can speculate that this can be game changer for AI coding when instead of working with plain files with code, we have full context aware resources that represent code units, fragments, etc. - maybe its time to think about a new internal representation of code.

# Client Concepts

**Sampling:** allows servers to request language model completions through the client

**Roots:** are a mechanism for clients to communicate filesystem access boundaries to servers.

**Elicitation:** Elicitation enables servers to request specific information from users during interactions

Sampling enables servers to perform AI-dependent tasks without directly integrating with or paying for AI models. Instead, servers can request that the client—which already has AI model access—handle these tasks on their behalf. – sampling looks very promising from a security and billing perspective, as server writers don't need to hold the keys to the kingdom for expensive models

Roots are a mechanism for clients to communicate filesystem access boundaries to servers. They consist of file URIs that indicate directories where servers can operate, helping servers understand the scope of available files and folders.

Elicitation provides a structured way for servers to gather necessary information on demand. Instead of requiring all information up front or failing when data is missing, servers can pause their operations to request specific inputs from users.

**Let's try something practical!**  
(what could go wrong?)



# A Quick Demo

Your task is to create a Model Context Protocol (MCP) server to expose a "hello world" tool. For the MCP implementation, you should use the official Go SDK for MCP and use the stdio transport.

Read these references to gather information about the technology and project structure before writing any code:

- <https://raw.githubusercontent.com/modelcontextprotocol/go-sdk/refs/heads/main/README.md>
- <https://go.dev/doc/modules/layout>

To test the server, use shell commands like these:

```
(
  echo
  '{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protocolVersion":"2025-06-18"}}'
;
  echo '{"jsonrpc":"2.0","method":"notifications/initialized","params":{}}';
  echo '{"jsonrpc":"2.0","id":2,"method":"tools/list","params":{}}';
) | ./bin/godoctor
```

Here is an example of a prompt with all these elements. Let's try to run it and see some of those tools in action. (run the prompt, watch for a bit and continue with the talk, might take a while ~3 min)

# A Peek at the Future

Go SDK for MCP:

<https://github.com/modelcontextprotocol/go-sdk>

MCP Support for gopls:

<https://tip.golang.org/gopls/features/mcp>

Here are some highlights we can expect over the next few months from the Go community, the first two being two things the Go team is actively working on

(give a tour of gopls)

The Go SDK for MCP is a partnership between Anthropic and Google and is the sdk I've just used on my example.

## Useful MCP Servers

Playwright: navigates web pages, takes screenshots, ...

<https://github.com/microsoft/playwright-mcp>

Context7: retrieves documentation from a crowdsourced repo

<https://context7.com/>

Don't forget to go back to the demo / prompt you left running two slides ago!

## What About Building Your Own?

[How to Build a Coding Assistant  
with Gemini CLI, MCP and Go |  
Google Codelabs](#)



Here is the link to a code lab where you can build your own mcp server from scratch and play with all these concepts.

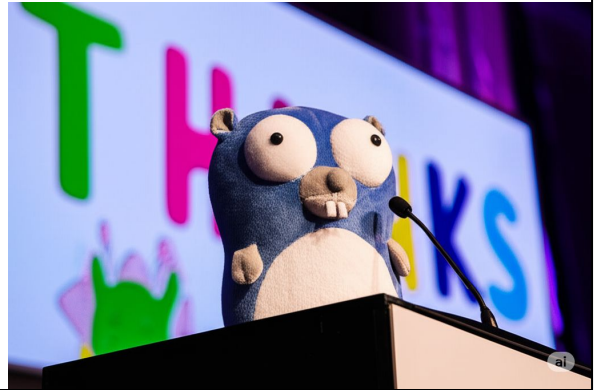
# Thank you!

[danicat.dev](https://danicat.dev)

[linkedin.com/in/petruzalek](https://linkedin.com/in/petruzalek)

@danicat83

Google Cloud



If you would like to know more, reach out to me on any of my socials! Thank you and have a great day!