

# ないなら作る MCP サーバー構築ハンズオン

MCP サーバーの基礎から実践レベルの知識まで

# 自己紹介

- ▶ azukiazusa
- ▶ <https://azukiazusa.dev>
- ▶ FE(フロントエンド|ファイアーエムブレム)が  
好き



# アジェンダ

- ▶ MCP サーバーの基礎知識について(10分)
- ▶ MCP サーバー構築ハンズオン(20分)
- ▶ MCP サーバーの実践的な知識について(15分)

# MCP サーバーの基礎知識

# MCP とは何か

## Model Context Protocol (MCP)

アプリケーションが LLM にコンテキストを渡す方法を標準化するための規格

- Claude を提供する Anthropic が開発・発表
- ツールのインターフェースを統一
- AI アプリケーション用の USB-C ポートのようなもの

# なぜ MCP が必要なのか?

→ LLM の機能を拡張するため

- LLM には知識カットオフがあり、最新の情報や組織内の情報を取得できない
  - Web 検索をしたり、社内ドキュメントを参照しその情報をコンテキストに渡す必要がある
  - ChatGPT の Plugins では外部のツールを呼び出す仕組みが提供された
    - Excel や PDF をアップロードして、組織内の情報を取得できるように
    - メールの送信やカレンダーの予定作成など、日常的なタスクを自動化できるように
- ChatGPT の Plugins は OpenAI 独自の仕組みであり、他の LLM では利用できない

# function calling

- ▶ 開発者がコードレベルで LLM に外部ツールを呼び出させるためには、function calling といった仕組みが使われてきた
  - ▶ 天気情報を取得するために天気情報 API を呼び出したり、Slack API を呼び出してメッセージを送信したりする関数を LLM が呼び出す
- ▶ LLM の SDK ごとに異なる実装が必要で、開発者にとっては負担が大きい

## function calling の例

```
const response = await openai.chat.completions.create({  
  model: "gpt-4",  
  tools: [  
    {  
      type: "function",  
      function: {  
        name: "get_weather",  
        description: "天気を取得",  
        parameters: {  
          type: "object",  
          properties: {  
            location: {  
              type: "string",  
            },  
          },  
        },  
      },  
    },  
  ],  
});
```

```
const response = await anthropic.messages.create({  
  model: "claude-3-5-sonnet",  
  tools: [  
    {  
      name: "get_weather",  
      description: "天気を取得",  
      input_schema: {  
        type: "object",  
        properties: {  
          location: {  
            type: "string",  
          },  
        },  
      },  
    },  
  ],  
});
```

# MCP が解決したこと

- ・ 1つの MCP サーバーを開発すれば、複数のクライアントから利用できる
- ・ 各プログラミング言語向けの SDK が提供されているため、効率よく MCP サーバーを開発し、パッケージマネージャーで配布できる
- ・ 企業が自社のデータを LLM に提供する手段として普及が進んだ
- ・ 現在では Anthropic が提供する Claude だけでなく、OpenAI の GPT や Google の Gemini など、主要な LLM が MCP をサポートし事実上の標準となっている

# MCP サーバーの例

- Slack MCP サーバー: Slack の情報を検索したり、メッセージを送信したりできる
- Playwright MCP サーバー: Web ブラウザを操作して情報を取得したり、操作を自動化したりできる
- Figma MCP サーバー: Figma のリソースを元にコードを生成
- Sentry MCP サーバー: Sentry のエラー情報を取得し、原因の特定や修正方法を提案

Registry で MCP サーバーを探してみよう <https://github.com/mcp>

# MCP の仕組み



# MCP のトランSPORT

## stdio(標準入出力)

- 標準入力/出力を使用した通信
- ローカル環境で動作するため、パッケージをインストールする必要がある

## Streamable HTTP

- HTTP を使用した通信
- ユーザーは MCP をローカルにインストールする必要がない

## SSE(非推奨)

- 互換性維持のために残されている
- Streamable HTTP を実装した場合、後方互換性のために SSE も実装する必要がある

# JSON-RPC 2.0

- ▶ JSON-RPC 2.0 (<https://www.jsonrpc.org/specification>)を使用して通信
- ▶ JSON-RPC とは、リモートプロシージャコール (RPC) を JSON フォーマットで実装するための軽量なプロトコル

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

# MCP の 3 つの機能

## リソース

- ▶ ユーザーや LLM がアクセスできるデータ
- ▶ 例: ドキュメント、画像、UI コンポーネント

## プロンプト

- ▶ 再利用可能なプロンプトテンプレート
- ▶ 組織内で効果的なプロンプトを共有

## ツール

- ▶ LLM が呼び出せる外部ツール
- この発表ではツールに焦点を当てる

# MCP サーバー構築デモ

TypeScript SDK を使用してサイコロツールを実装

# プロジェクトのセットアップ

```
git clone https://github.com/azukiazusa1/mcp-hanson.git
```

```
cd mcp-hanson  
npm install
```

# 使用するパッケージ

- `@modelcontextprotocol/sdk`: MCP サーバーを構築するための公式 SDK
- `zod`: ツールの入力と出力のスキーマを定義するためのライブラリ
- `@modelcontextprotocol/inspector`: MCP サーバーの動作確認を行うためのツール

# サーバーの基本構造

src/server.ts

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

// MCP サーバーのインスタンスを作成
const server = new McpServer({
  name: "dice-server",
  version: "1.0.0",
});
```

# ツールの定義と実装

```
import { z } from "zod";

server.registerTool(
  "roll_dice", // ツールの一意な名前
  {
    title: "Roll Dice", // 人間が読めるツールの名前
    description: "ランダムな数字を生成するサイコロツール", // ツールの説明
    // ツールの入力スキーマ
    inputSchema: {
      sides: z.number().optional().describe("サイコロの面の数(デフォルト: 6)"),
    },
    // ツールの出力スキーマ
    outputSchema: {
      result: z.number(),
    },
    // LLM がツールを呼び出したときに実行される関数
    async ({ sides = 6 }) => {
      const result = Math.floor(Math.random() * sides) + 1;

      return {
        content: [{ type: "text", text: JSON.stringify({ result }) }],
        structuredContent: { result },
      };
    },
  );
}
```

## ツールのポイント

- ▶ `title` はツールの一覧に表示されるため、人間が理解しやすい名前にする
- ▶ `description` は LLM がツールを呼び出す判断に影響するため重要
  - ▶ LLM のシステムプロンプトに自動で追加される
- ▶ `inputSchema` と `outputSchema` でツールのインターフェイスを定義
  - ▶ JSON Schema に基づいている。TypeScript では zod を使用して定義可能
- ▶ `outputSchema` は必須ではないが、定義することでクライアントや LLM がツールを適切に処理しやすくなる
- ▶ ツールの戻り値の `structuredContent` は `outputSchema` に準拠している必要がある
- ▶ `outputSchema` をサポートしていないクライアントのために、`content` も返す

# サーバーの起動(stdio)

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

// stdio transport を使用してサーバーを起動

async function main() {
  const transport = new StdioServerTransport();
  await server.connect(transport);
  // stdio で通信するので、console.log は使わない
  console.error("MCP Server is running...");
}

main();
```

# サーバーをビルド

```
npm run build
```

→ `build/server.js` が生成される

# MCP Inspector で動作確認

```
npx @modelcontextprotocol/inspector node build/server.js
```

- ・ LLM は実装したツールを確実に呼び出してくれるとは限らない
- ・ MCP Inspector はブラウザ上の UI でツールを直接呼び出して動作確認できる

# MCP Inspector での接続操作



# roll\_dice ツールの動作確認

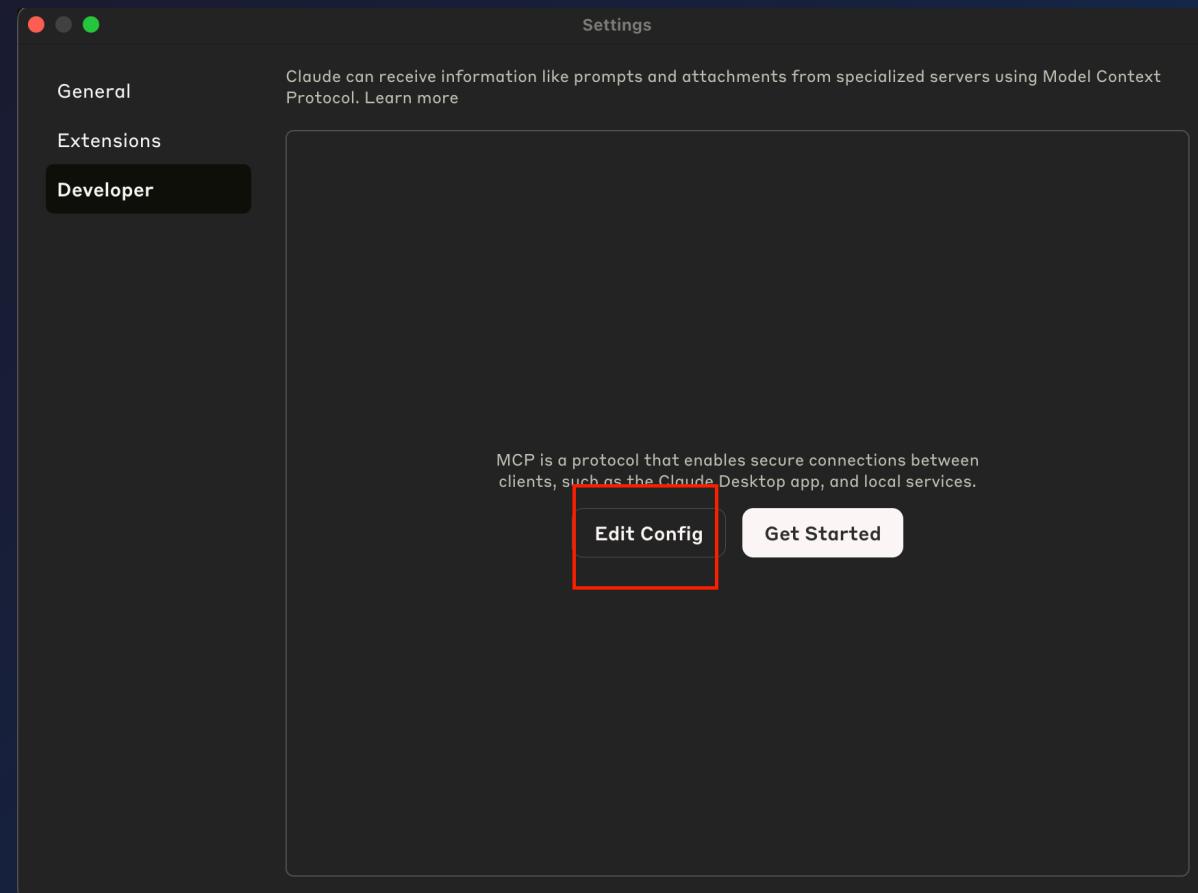
The screenshot shows the MCP Inspector interface with the following details:

- MCP Inspector v0.17.1** header.
- Transport Type**: STDIO.
- Command**: node.
- Arguments**: build/server.js, Environment Variables, Server Entry, Servers File, Authentication, Configuration.
- Buttons**: Restart, Disconnect (Connected).
- Tools Tab**: Shows the "Tools" section with a list of tools. The "roll\_dice" tool is highlighted with a red box and described as "ランダムな数字を生成するサイコロツール".
- roll\_dice Tool Details**:
  - Description: ランダムな数字を生成するサイコロツール.
  - sides**: サイコロの面の数(デフォルト: 6).
  - Output Schema:**

```
{  content: [    0: {      type: "text",      text: "{\"result\":1}"    }  ],  structuredContent: {    result: 1  }}
```
  - Buttons**: Run Tool (highlighted with a red box), Copy Input.
- Server Notifications**: No notifications yet.

# Claude Desktop での設定

Settings > Developer > Edit Config



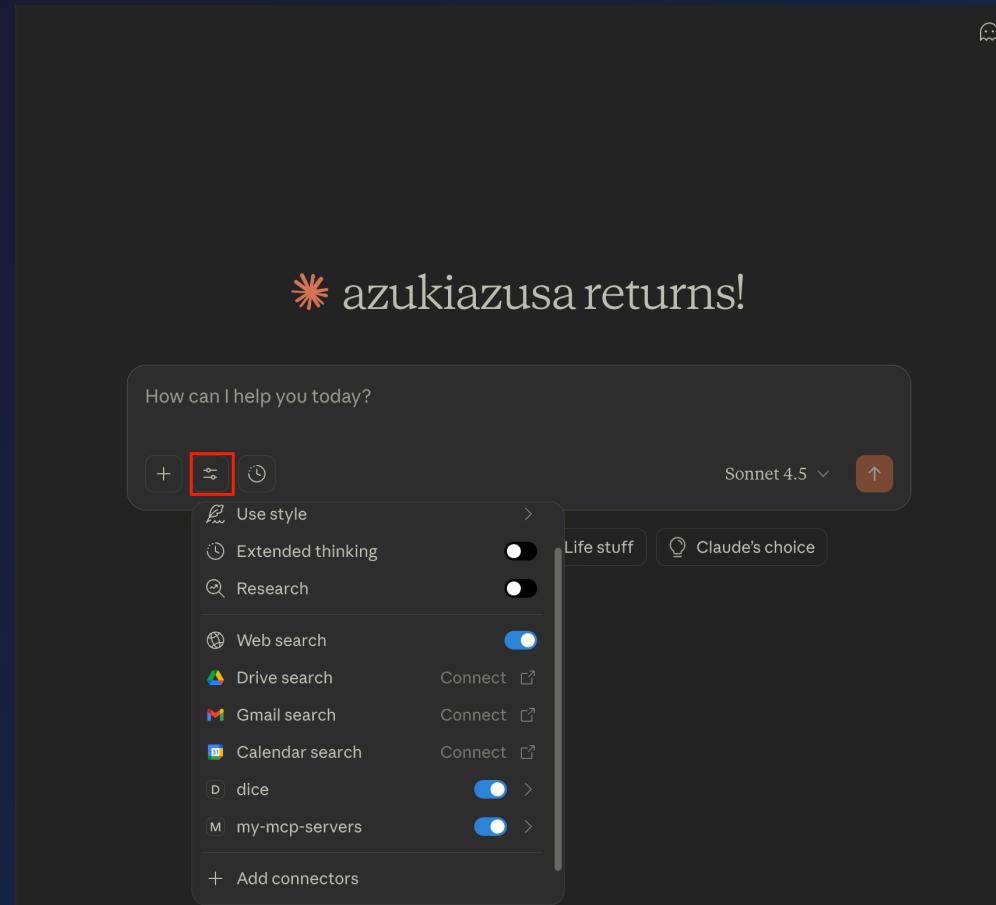
# JSON 設定例

claude\_desktop\_config.json ファイルに以下を追加

```
{  
  "mcpServers": {  
    "dice": {  
      "command": "node",  
      "args": ["/path/to/build/server.js"]  
    }  
  }  
}
```

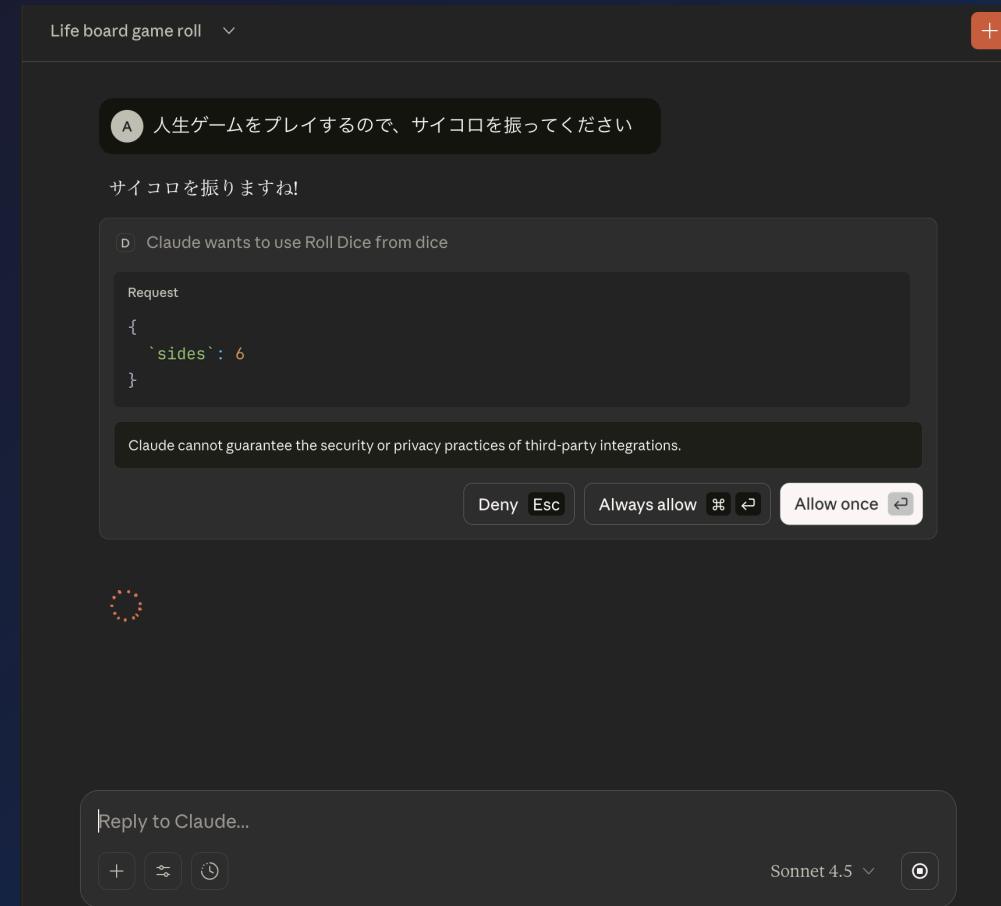
# MCP サーバーのツールが追加されたことを確認

Claude Desktop を再起動して、ツールの一覧に `dice` が表示されていることを確認



# Claude Desktop で実行

Roll Dice ツールの呼び出しの許可が求められる



# サイコロを振った結果を元に Claude が応答

Life board game roll ▾

A 人生ゲームをプレイするので、サイコロを振ってください

サイコロを振りますね!

D Roll Dice

1が出ました!

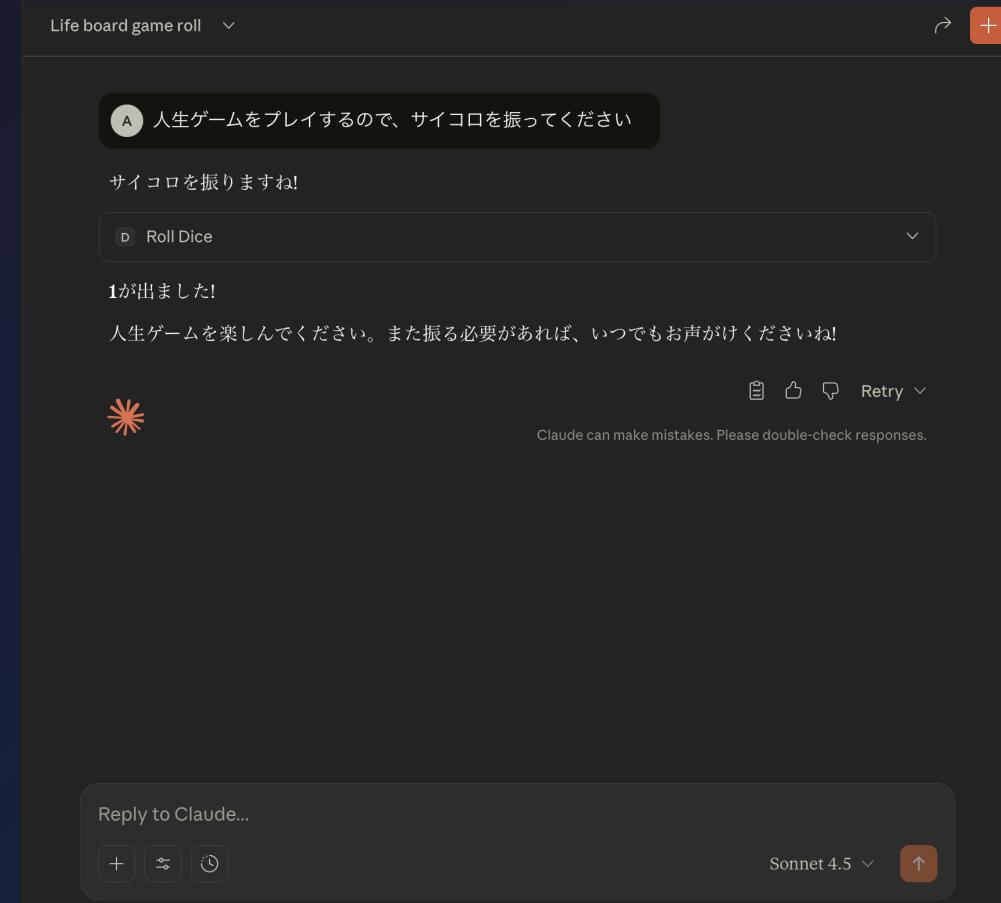
人生ゲームを楽しんでください。また振る必要があれば、いつでもお声がけくださいね!

Retry ▾

Claude can make mistakes. Please double-check responses.

Reply to Claude...

Sonnet 4.5 ▾



# MCP サーバーを提供するには

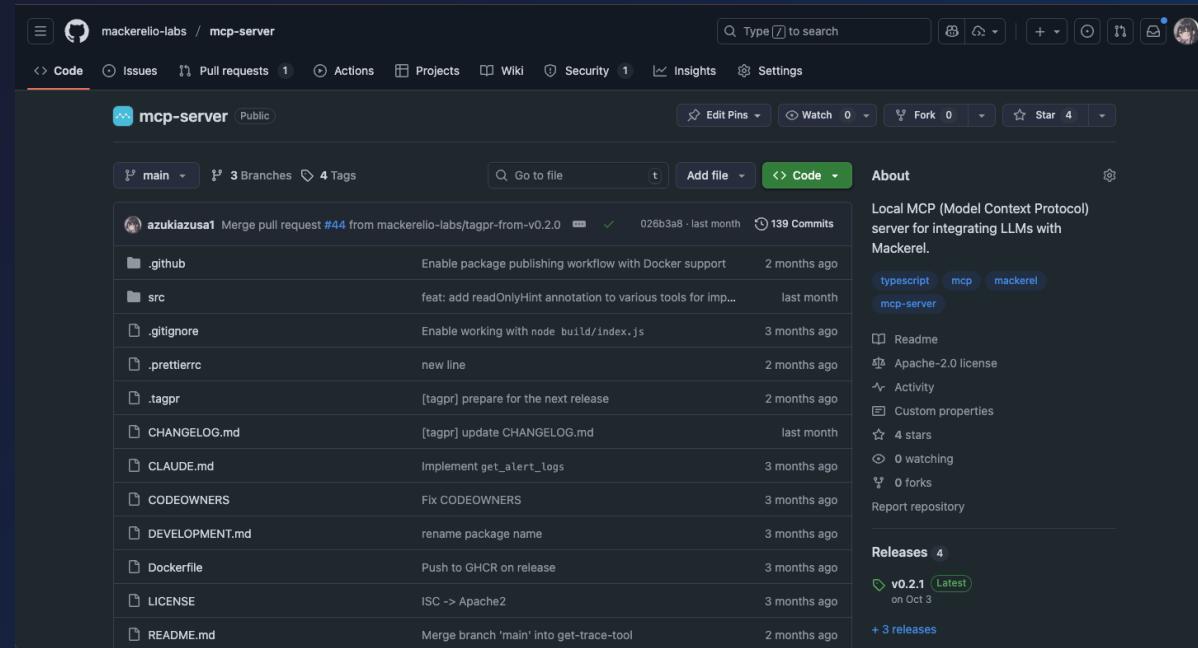
- stdio: プログラミング言語のパッケージマネージャー・もしくは Docker コンテナで配布するのが一般的
  - TypeScript: npm
  - Python: uv
- Streamable HTTP: HTTP サーバーとしてデプロイ

# MCP サーバーの実践的な知識

- ・ 私が実際に本番レベルで MCP サーバーを開発してきた中で得た知見・失敗談を共有

# Mackerel MCP サーバーを開発

- ・ Mackerel は株式会社はてなが提供するクラウド型サーバー監視サービス
- ・ MCP サーバーを利用して AI と連携することで、アラート情報の取得から原因分析、詳細な対処手順の提案までを AI が支援



# MCP サーバー設計で大切なこと

Web API の設計知識は捨てる = アンラーニング

# MCP サーバー設計で実際に失敗したこと

1. API のラッパーとして提供してしまう
2. レスポンスのコンテキストサイズが大きすぎる
3. LLM がツールの呼び出し方を誤る

**1.API のラッパーとして提供してしまう**

# API のラッパーとして提供すると失敗する

- ▶ REST API はリソースベースの設計
  - ▶ 1つのリソースに対して GET、POST、PUT、DELETE などの操作ごとにエンドポイントが存在
- ▶ ツールはユーザーが達成したいタスクベースの設計
- ▶ プログラミングでは1つのタスクを達成するために複数の API を組み合わせることが一般的
- ▶ LLM は複数のツールを組み合わせてタスクを達成することが苦手

# 例: カレンダー API

## 従来の API 設計

- ▶ `GET /users` - ユーザー取得
- ▶ `GET /events` - イベント取得
- ▶ `POST /events` - イベント作成

ユースケースの例: 参加者とのミーティングをスケジュール

1. `GET /events` で特定の日付の空き時間を取得
2. `GET /users` で参加者のユーザー ID を取得
3. `POST /events` でミーティングを作成

→ この設計に従うと、`get_users`, `get_events`, `create_event` といったツールを作りたくなる

# 例: カレンダー API

## LLM 向けのツール設計

- `schedule_meeting` - ミーティングをスケジュール（ツールの実装の中で複数のAPIを呼び出す）
  - 1つのタスクを1つのツールで完結
- 一方でツールの詳細度を上げすぎてしまうと、汎用性が低くなってしまうのでいい塩梅を見つける必要がある

## 従来のプログラミングの例

```
app.get("/get_user", async (req, res) => {
  const user = await searchUser(req.query.name);
  res.json(user);
});

app.get("/get_events", async (req, res) => {
  const events = await getEvents(req.query.userId, req.query.date);
  res.json(events);
};

app.post("/create_event", async (req, res) => {
  const event = await createEvent(
    req.body.title,
    req.body.attendeeIds,
    req.body.timeSlot,
  );
  res.json(event);
});
```

## MCP サーバーの例

```
// MCP ツール: 1 つのツールでタスクを完結
server.registerTool(
  "schedule_meeting",
  {
    title: "Schedule Meeting",
    description: "指定したメンバーとのミーティングをスケジュール",
    inputSchema: {
      attendeeName: z.string().describe("参加者の名前"),
      title: z.string().describe("ミーティングのタイトル"),
      date: z.string().describe("日付(YYYY-MM-DD)"),
      duration: z.number().describe("所要時間(分)"),
    },
    outputSchema: { eventId: z.string(), startTime: z.string() },
  },
  async ({ attendeeName, title, date, duration }) => {
    // ツール内部で全ての処理を実行
    const user = await searchUser(attendeeName);
    const freeSlot = await findFreeSlot(user.id, date, duration);
    const event = await createEvent(title, [user.id], freeSlot);

    return {
      content: [{ type: "text", text: `ミーティングを作成しました` }],
      structuredContent: { eventId: event.id, startTime: freeSlot.start },
    };
  },
);
```

# ツール設計のポイント

## 提供するツールの数を絞る

- ・ 提供するツールの数が多くなると、LLM がどのツールを使うべきか迷ってしまう
- ・ 多くのユーザーは複数の MCP サーバーを同時に利用するので、思ったよりもツールの数が多くなりがち
- ・ `description` の内容は LLM のシステムプロンプトに自動で追加されるため、コンテキストの圧迫につながる
- ・ MCP サーバーのコンテキストの圧迫という課題に対して Claude Skills という機能も発表された

# ツール設計のポイント

## タスクベースで設計

- ▶ ユーザーが何を達成したいのか、ユースケースを考えてツールを設計することが重要
- ▶ ユーザーが `GET /users` を呼び出したいのは何のためか?
  - ▶ `POST /events` でミーティングを作成するために参加者のユーザー ID を知りたいのが真の目的で、最終的なタスクはミーティングのスケジュール
- ▶ ツールの実装の中で複数の API を組み合わせてタスクを達成

2. レスポンスのコンテキストが大きすぎる

# レスポンスのコンテキストが大きすぎる問題

LLM にはツールレスポンスのコンテキスト長の制限がある

- Claude Code: デフォルトで 25,000 トークン
  - これを超えるとレスポンスを返してしまうとエラーとなり、やり取りが終了してしまう
  - コンテキストが大きいと性能が低下
  - LLM が無関係な情報に注意を奪われる → コンテキスト汚染

# 従来のプログラミングとの考え方の違い

- ・ 現代の富豪的プログラミングでは 1,000 件のリストをメモリに載せてフィルタリング・ソートしても問題ない
- ・ LLM ではコンテキスト制限があるため同じアプローチは不可
- ・ 限られたコンテキストサイズで必要な情報だけを提供する工夫が必要  
→ API の応答をそのまま返すのは避ける

# 解決策 1: ページネーションの導入

```
server.registerTool(
  "search_users",
  {
    inputSchema: {
      query: z.string().describe("検索クエリ"),
      limit: z.number().optional().default(10).describe("取得件数"),
      offset: z.number().optional().default(0).describe("オフセット"),
    },
  },
  async ({ query, limit = 10, offset = 0 }) => {
    const users = await db.users.search(query).limit(limit).offset(offset);
    // ...
    return {
      content: [{ type: "text", text: JSON.stringify(result) }],
      structuredContent: result,
    };
  },
);
```

## 解決策 2: 必要なフィールドだけ取得

```
server.registerTool(
  "get_user",
  {
    inputSchema: {
      userId: z.string().describe("ユーザー ID"),
      fields: z
        .array(z.enum(["name", "email", "avatar", "bio"]))
        .optional()
        .default(["name", "email"])
        .describe("取得するフィールド"),
    },
  },
  async ({ userId, fields = ["name", "email"] }) => {
    const user = await db.users.findById(userId).select(fields);

    return {
      content: [{ type: "text", text: JSON.stringify(user) }],
      structuredContent: user,
    };
  },
);
```

## 解決策 3: データの粒度を選択させる

```
server.registerTool(  
  "get_log_summary",  
  {  
    inputSchema: {  
      responseFormat: z  
        .enum(["detailed", "summary"])  
        .optional()  
        .default("summary"),  
    },  
    },  
    async ({ responseFormat = "summary" }) => {  
      const logs = await parseLogs(date);  
      const result = responseFormat === "summary" ? summarizeLogs(logs) : logs;  
      return {  
        content: [{ type: "text", text: JSON.stringify(result) }],  
        structuredContent: result,  
      };  
    },  
);
```

3. LLM が誤ったツール呼び出しを行う

### 3. LLM が正しいツールの呼び出し方を理解していない問題

- ▶ ホストメトリック取得ツールを実装
- ▶ 存在しないメトリック名を繰り返し呼び出して失敗し続ける

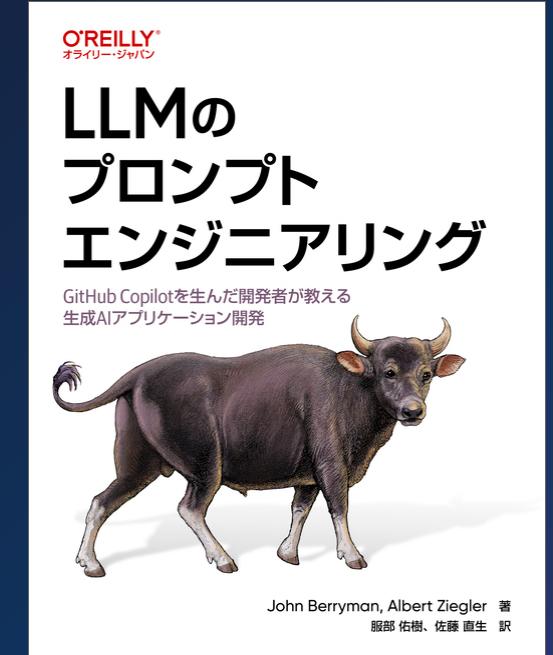
ホストメトリック: 監視対象のホストに対応する指標。例えば...

- ▶ `cpu.user.percentage`: CPU 使用率
- ▶ `memory.used`: メモリ使用率
- ▶ `lambda.count.invocations`: AWS Lambda の呼び出し回数

# 解決策 1: **description** のプロンプトエンジニアリング

- ・ ツールの説明は LLM のコンテキストに含まれるので、プロンプトエンジニアリングの知識が活用できる
- ・ 特殊なクエリ形式、ニッチな用語の定義があれば明示的に記述する
- ・ 反対に SQL のように広く知られた形式は説明しない方が良い
- ・ ユーザーがどのような場面でツールを使うべきかを明示する
- ・ ツールの使用例を Few-shot で示す

<https://www.oreilly.co.jp/books/9784814401130/>



# ツールの `description` の例

```
{  
  name: "get_host_metric",  
  description: `ホストのメトリックを取得します。`
```

サポートされているメトリック:

- `cpu.user.percentage`: CPU 使用率
- `memory.used`: メモリ使用率
- `disk.used.percentage`: ディスク使用率

以下のユーザーの質問に答えるためにこのツールを使用してください:

- ホスト A はスケールアップが必要かどうか調査してください
- ホスト B のパフォーマンスをグラフで表示してください
- ホスト C のディスク使用率が高いか確認してください

<example>

- ホスト A の CPU 使用率を取得: `get_host_metric("host A", "cpu.user.percentage")`
- ホスト B のメモリ使用率を取得: `get_host_metric("host B", "memory.used")`

</example>

```
`  
,  
// ...  
`
```

## 解決策 2: エラー応答を詳細にする

### 悪い例

```
{  
  "code": 404,  
  "message": "Not Found"  
}
```

- ▶ LLMにとって意味のない応答で、この結果を元に LLMがどのように問題を解決すればよいか分からぬ
- ▶ AIエージェントの「行動・フィードバック・改善」フィードバックループが回せぬ

## エラー応答をプロンプトエンジニアリングする

- ・なぜツールの呼び出しが失敗したのか、どのように問題を解決できるのかをマークダウン形式で返す
  - ・エラーコードやスタックトレースを返すのではなく、具体的かつ実用的な情報を提供
- ・必ずしも JSON 形式で返す必要はない
  - ・ツールの実装はついプログラミング的に処理しやすい構造化されたデータで考えてしまいがちだが、LLM にとっては対話形式の方が理解しやすいことを忘れてはならない

# 良いエラー応答の例

```
server.registerTool("get_host_metric", async ({ host, metric }) => {
  try {
    // ...
  } catch (error) {
    const errorMessage = `# メトリクスが見つかりません
```

指定されたメトリクス名が無効であるか、このホストで利用できないため、リクエストが失敗しました。

## 考えられる原因

- メトリクス名にタイプミスまたは誤った形式が含まれています
- このホストではメトリクスが利用できない可能性があります
- メトリクスの収集が有効になっていない可能性があります

## メトリクスの問題を解決する方法

- メトリクス名のタイプミスを確認してください
- このホストタイプでメトリクスが利用可能か確認してください`;

```
return {
  content: [{ type: "text", text: errorMessage }],
  isError: true,
};
}
});
```

# その他の Tips

# ID より人間が読める名前を使う

## 問題点

LLM は難解な識別子の処理が苦手

- ▶ UUID: `e5b6d8c2-7dad-438e-8b05-92c5cce00246`
- ▶ 英数字 ID: `abc123xyz`

# 推奨されるアプローチ

## 良い例

```
search_user_by_name("Alice");
get_product_by_name("iPhone 15 Pro");
```

## 悪い例

```
get_user_by_id("usr_1234567890");
get_product_by_id("prod_abc123");
```

# ツールを実際に試してフィードバックを収集する

- ・ 従来のプログラミングと異なり、LLM のツール呼び出しは決定的ではない
- ・ 実際に LLM を通じてツールを試してみると発見できない問題が多い
- ・ ツールで達成したいユースケースを通じてフィードバックを収集し、ツールを改善していく

# まとめ

- MCP はツールのインターフェースを標準化するプロトコル
- Claude だけでなく主要な LLM が MCP をサポートし事実上の標準に
- TypeScript SDK でツールの開発を行った
- ツールの `description` と入力スキーマの設計が重要
- Web API のラッパーではなく、タスクベースでツールを設計
- コンテキストサイズを意識し、ページネーションや要約を活用
- LLM に優しい `description` とエラー応答で誤呼び出しを防ぐ

# 参考資料

- ▶ Model Context Protocol公式ドキュメント  
<https://modelcontextprotocol.io/>
- ▶ やさしい MCP 入門  
<https://www.shuwasytem.co.jp/book/9784798075730.html>
- ▶ TypeScript SDK  
<https://github.com/modelcontextprotocol/typescript-sdk>
- ▶ Writing Tools for Agents  
<https://www.anthropic.com/engineering/writing-tools-for-agents>
- ▶ The second wave of MCP: Building for LLMs, not developers  
<https://vercel.com/blog/the-second-wave-of-mcp-building-for-langs-not-developers>

