12

MCP でのコード実行: より効率的なエージェントを作る

— Code execution with MCP: Building more efficient agents —

公開日	2025-11-04
原題	Code execution with MCP: Building more efficient agents
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/code-execution-with-mcp
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

MCP でのコード実行: より効率的なエージェントを作る

Model Context Protocol（MCP） は、AI エージェントを外部システムに接続するためのオープン標準です。従来は、エージェントとツール・データを繋ごうとすると、ペアごとにカスタム統合が必要でした。これがフラグメンテーションと作業の重複を生み、真に接続されたシステムをスケールさせることを難しくしていました。MCP は普遍的なプロトコルを提供します——開発者は自分のエージェントに MCP を一度実装すれば、統合のエコシステム全体が解き放たれます。

2024 年 11 月に MCP をローンチして以来、採用は急速に広がりました。コミュニティは数千の MCP サーバー を作り、主要プログラミング言語向けの SDK が提供され、業界はエージェントをツール・データに接続する事実上の標準として MCP を採用しています。

今日、開発者は日常的に数百から数千のツール、数十の MCP サーバーにアクセスできるエージェントを作ります。しかし、接続ツールの数が増えるにつれ、すべてのツール定義を前もってロードし、中間結果をコンテキストウィンドウ経由で渡すことはエージェントを遅くし、コストを増やします。

本稿では、MCP サーバーとより効率的に相互作用できるようにするためにコード実行がどう役立つか、より多くのツールを扱いつつトークン使用を減らす方法を探ります。

ツールからの過剰なトークン消費がエージェントを非効率にする

MCP の利用がスケールすると、エージェントのコストとレイテンシを増やす 2 つの典型的パターンがあります。

1. ツール定義がコンテキストウィンドウを圧迫する
2. 中間的なツール結果が追加のトークンを消費する

1. ツール定義がコンテキストウィンドウを圧迫する

ほとんどの MCP クライアントは、すべてのツール定義を直接コンテキストにロードし、直接ツール呼び出し構文でモデルに露出させます。ツール定義は次のようになっています。

```
gdrive.getDocument
     Description: Retrieves a document from Google Drive
     Parameters:
               documentId (required, string): The ID of the document to retrieve
               fields (optional, string): Specific fields to return
     Returns: Document object with title, body content, metadata, permissions, etc.
```

```
salesforce.updateRecord
    Description: Updates a record in Salesforce
    Parameters:
               objectType (required, string): Type of Salesforce object (Lead, Conta
ct,      Account, etc.)
               recordId (required, string): The ID of the record to update
               data (required, object): Fields to update with their new values
     Returns: Updated record object with confirmation
```

ツール説明はコンテキストウィンドウをより多く占有し、応答時間とコストを増やします。エージェントが数千のツールに接続されているケースでは、リクエストを読む前に数十万トークンを処理することになります。

2. 中間的なツール結果が追加のトークンを消費する

ほとんどの MCP クライアントは、モデルに MCP ツールを直接呼ばせることを許します。たとえばエージェントにこう頼むとしましょう:「Google Drive からミーティングの書き起こしをダウンロードして、Salesforce のリードに添付して」。

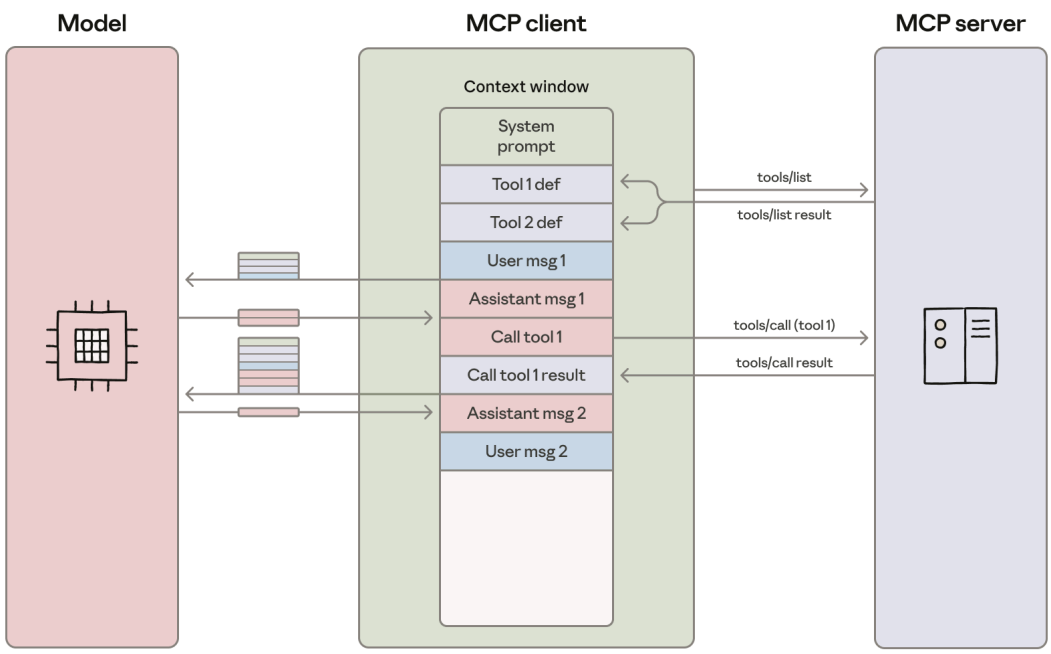
モデルはこのような呼び出しをします。

```
TOOL CALL: gdrive.getDocument(documentId: "abc123")
        → returns "Discussed Q4 goals...\n[full transcript text]"
           (loaded into model context)

TOOL CALL: salesforce.updateRecord(
    objectType: "SalesMeeting",
    recordId: "00Q5f000001abcXYZ",
      data: { "Notes": "Discussed Q4 goals...\n[full transcript text written out]" }
  )
  (model needs to write entire transcript into context again)
```

すべての中間結果はモデルを通過しなければなりません。この例では、通話の書き起こし全文が 2 回流れます。2 時間のセールスミーティングの場合、追加で 5 万トークン処理することになりかねません。さらに大きな文書ではコンテキストウィンドウ上限を超えてワークフローが壊れるかもしれません。

大きな文書や複雑なデータ構造を扱うとき、モデルはツール呼び出し間でデータをコピーする際にミスを犯しやすくなります。



MCP クライアントはツール定義をモデルのコンテキストウィンドウにロードし、各ツール呼び出しと結果が操作の合間にモデルを通過するメッセージループを調整する。

MCP でのコード実行でコンテキスト効率を改善する

エージェント向けのコード実行環境が一般的になるにつれ、MCP サーバーを直接のツール呼び出しではなくコード API として提示する、という解があります。エージェントはコードを書いて MCP サーバーと相互作用します。このアプローチは両方の課題に対処します——エージェントは必要なツールだけをロードでき、結果をモデルに返す前に実行環境でデータを処理できます。

これを行う方法はいくつかあります。1 つのアプローチは、接続された MCP サーバーから利用可能な全ツールのファイルツリーを生成することです。TypeScript を使った実装例:

```
servers
├── google-drive
│   ├── getDocument.ts
│   ├── ... (other tools)
│   └── index.ts
├── salesforce
│   ├── updateRecord.ts
│   ├── ... (other tools)
│   └── index.ts
└── ... (other servers)
```

各ツールはこんなファイルに対応します。

```
// ./servers/google-drive/getDocument.ts
import { callMCPTool } from "../../../client.js";

interface GetDocumentInput {
  documentId: string;
}

interface GetDocumentResponse {
  content: string;
}

/* Read a document from Google Drive */
export async function getDocument(input: GetDocumentInput): Promise<GetDocumentRespo
nse> {
  return callMCPTool<GetDocumentResponse>('google_drive__get_document', input);
}
```

先ほどの Google Drive から Salesforce の例は、次のコードになります。

```
// Read transcript from Google Docs and add to Salesforce prospect
import * as gdrive from './servers/google-drive';
import * as salesforce from './servers/salesforce';

const transcript = (await gdrive.getDocument({ documentId: 'abc123' })).content;
await salesforce.updateRecord({
  objectType: 'SalesMeeting',
  recordId: '00Q5f000001abcXYZ',
  data: { Notes: transcript }
});
```

エージェントはファイルシステムを探索してツールを発見します——`./servers/` ディレクトリをリストして利用可能なサーバー（`google-drive`、`salesforce` など）を見つけ、必要な特定ツールファイル（`getDocument.ts`、`updateRecord.ts`）を読んで各ツールのインターフェースを理解します。これにより、エージェントは現在のタスクに必要な定義だけをロードできます。トークン使用量は 15 万トークンから 2,000 トークンに減り、**時間とコストが 98.7% 削減** されます。

Cloudflare も同様の発見を公表しており、MCP でのコード実行を「Code Mode」と呼んでいます。中核的な洞察は同じです——LLM はコードを書くのが得意で、開発者はこの強みを活かして、MCP サーバーとより効率的に相互作用するエージェントを作るべきだ、ということです。

MCP でのコード実行のメリット

MCP でのコード実行は、ツールをオンデマンドでロードし、データがモデルに届く前にフィルタリングし、複雑なロジックを 1 ステップで実行することで、エージェントがコンテキストをより効率的に使えるようにします。このアプローチはセキュリティと状態管理の面でもメリットがあります。

漸進的開示（Progressive disclosure）

モデルはファイルシステムのナビゲートが得意です。ツールをファイルシステム上のコードとして提示することで、モデルは全てを前もってではなく、必要に応じてツール定義を読めます。

あるいは、関連定義を見つけるための `search_tools` ツールをサーバーに追加できます。たとえば仮想的な Salesforce サーバーで作業するとき、エージェントは「salesforce」を検索し、現在のタスクに必要なツールだけをロードします。`search_tools` ツールに詳細レベルのパラメータ（名前のみ、名前と説明、あるいはスキーマ付き完全定義）を含めておくと、エージェントがコンテキストを節約しつつ効率的にツールを見つけるのに役立ちます。

コンテキスト効率の良いツール結果

大きなデータセットを扱うとき、エージェントは結果を返す前にコードでフィルタ・変換できます。1 万行のスプレッドシートの例で考えてみましょう。

```
// コード実行なし - 全行がコンテキストを通る
TOOL CALL: gdrive.getSheet(sheetId: 'abc123')
        → returns 10,000 rows in context to filter manually

// コード実行あり - 実行環境でフィルタ
const allRows = await gdrive.getSheet({ sheetId: 'abc123' });
const pendingOrders = allRows.filter(row =>
  row["Status"] === 'pending'
);
console.log(`Found ${pendingOrders.length} pending orders`);
console.log(pendingOrders.slice(0, 5)); // Only log first 5 for review
```

エージェントは 10,000 行ではなく 5 行だけを見ます。同様のパターンは、集計、複数データソースの結合、特定フィールドの抽出にも使えます——どれもコンテキストウィンドウを膨張させません。

より強力でコンテキスト効率の良い制御フロー

ループ、条件分岐、エラーハンドリングを、個々のツール呼び出しを連鎖させるのではなく馴染みのあるコードパターンで扱えます。たとえば Slack にデプロイ通知が必要なとき、エージェントはこう書けます。

```
let found = false;
while (!found) {
  const messages = await slack.getChannelHistory({ channel: 'C123456' });
  found = messages.some(m => m.text.includes('deployment complete'));
  if (!found) await new Promise(r => setTimeout(r, 5000));
}
console.log('Deployment notification received');
```

このアプローチは、エージェントループ中に MCP ツール呼び出しと sleep コマンドを交互に繰り返すよりも効率的です。

加えて、実行される条件分岐木を書き出せることは「最初のトークンまでの時間」レイテンシの節約にもなります——if 文を評価するのにモデルを待つ必要はなく、エージェントはコード実行環境にこれをさせられるからです。

プライバシーを保つ操作

エージェントが MCP をコード実行で使うとき、中間結果はデフォルトで実行環境に留まります。そのためエージェントは、明示的にログや返却したものだけを見ます。つまりモデルに共有したくないデータは、モデルのコンテキストに入ることなくワークフローを通過できます。

より敏感なワークロードには、エージェントハーネスがセンシティブデータを自動的にトークナイズできます。たとえば、スプレッドシートから顧客連絡先詳細を Salesforce にインポートする必要があるとします。エージェントはこう書きます。

```
const sheet = await gdrive.getSheet({ sheetId: 'abc123' });
for (const row of sheet.rows) {
  await salesforce.updateRecord({
    objectType: 'Lead',
    recordId: row.salesforceId,
    data: {
      Email: row.email,
      Phone: row.phone,
      Name: row.name
    }
  });
}
console.log(`Updated ${sheet.rows.length} leads`);
```

MCP クライアントはデータを傍受し、PII がモデルに届く前にトークナイズします。

```
// エージェントが sheet.rows をログしたら見えるもの:
[
  { salesforceId: '00Q...', email: '[EMAIL_1]', phone: '[PHONE_1]', name: '[NAME_1]'
},
  { salesforceId: '00Q...', email: '[EMAIL_2]', phone: '[PHONE_2]', name: '[NAME_2]'
},
  ...
]
```

その後データが別の MCP ツール呼び出しで共有されるときは、MCP クライアントでのルックアップによりトークン化が解除されます。実際のメールアドレス、電話番号、名前は Google Sheets から Salesforce へ流れますが、モデルを通ることはありません。これにより、エージェントがセンシティブデータを誤ってログしたり処理したりすることを防げます。また、データがどこから／どこへ流れられるかを定義する決定論的なセキュリティルールを設けることにも使えます。

状態の永続化と skills

ファイルシステムアクセス付きのコード実行により、エージェントは操作にわたる状態を維持できます。エージェントは中間結果をファイルに書き、作業を再開して進捗を追跡できます。

```
const leads = await salesforce.query({
  query: 'SELECT Id, Email FROM Lead LIMIT 1000'
});
const csvData = leads.map(l => `${l.Id},${l.Email}`).join('\n');
await fs.writeFile('./workspace/leads.csv', csvData);

// 後続の実行は中断したところから続ける
const saved = await fs.readFile('./workspace/leads.csv', 'utf-8');
```

エージェントは自身のコードを再利用可能な関数として永続化することもできます。あるタスクに対して動くコードを開発したら、将来のために実装を保存できます。

```
// In ./skills/save-sheet-as-csv.ts
import * as gdrive from './servers/google-drive';
export async function saveSheetAsCsv(sheetId: string) {
  const data = await gdrive.getSheet({ sheetId });
  const csv = data.map(row => row.join(',')).join('\n');
  await fs.writeFile(`./workspace/sheet-${sheetId}.csv`, csv);
  return `./workspace/sheet-${sheetId}.csv`;
}

// Later, in any agent execution:
import { saveSheetAsCsv } from './skills/save-sheet-as-csv';
const csvPath = await saveSheetAsCsv('abc123');
```

これは Skills の概念と密接に結びついています。Skills は、モデルが特殊タスクで性能を改善するための再利用可能な指示、スクリプト、リソースのフォルダです。保存された関数に `SKILL.md` ファイルを追加すれば、モデルが参照・利用できる構造化された skill になります。時間をかければ、エージェントは高レベル能力のツールボックスを築き上げ、最も効果的に働くのに必要な足場を進化させられます。

コード実行には独自の複雑さが伴います。エージェントが生成したコードを実行するには、適切なサンドボックス化、リソース制限、監視を備えた安全な実行環境が必要です。これらのインフラ要件は、直接ツール呼び出しには無い運用オーバーヘッドとセキュリティ上の考慮を追加します。コード実行のメリット——トークンコストの削減、レイテンシの低下、ツール合成の改善——は、これらの実装コストと天秤にかけるべきです。

まとめ

MCP は、エージェントが多くのツールやシステムに接続するための基礎的プロトコルを提供します。しかし、接続されるサーバー数が多すぎると、ツール定義と結果が過剰なトークンを消費し、エージェントの効率を下げかねません。

ここでの問題の多く——コンテキスト管理、ツール合成、状態の永続化——は新しく感じられますが、ソフトウェアエンジニアリングに既知の解があります。コード実行はこれらの確立されたパターンをエージェントに適用し、馴染みのあるプログラミング構造で MCP サーバーとより効率的に相互作用させるものです。このアプローチを実装したら、ぜひ MCP コミュニティ に発見を共有してください。

謝辞

本稿は Adam Jones と Conor Kelly が執筆。草稿にフィードバックをくれた Jeremy Fox、Jerome Swannack、Stuart Ritchie、Molly Vorwerck、Matt Samuels、Maggie Vo に感謝します。