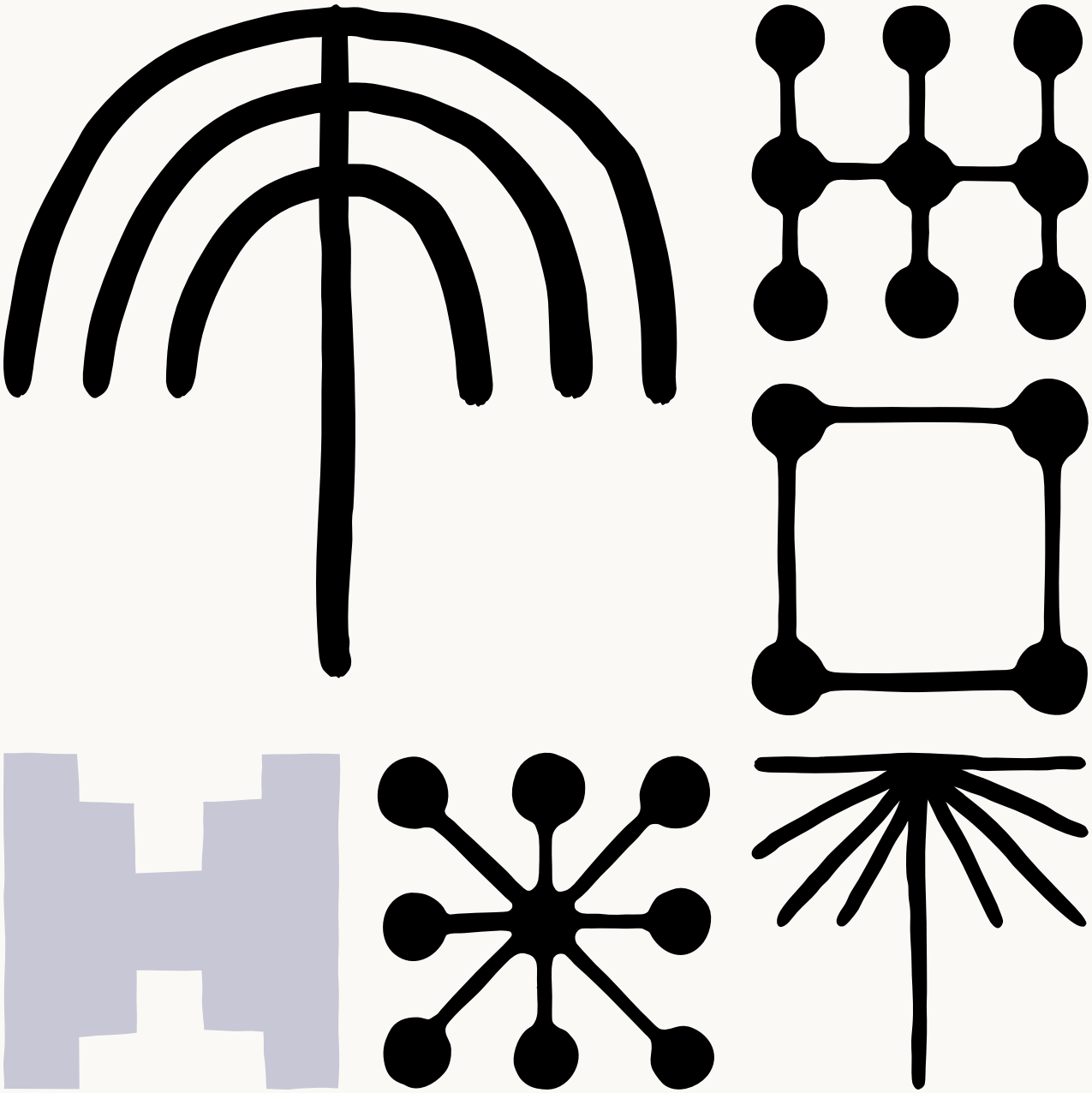
Engineering at Anthropic



Code execution with MCP: Building more efficient agents

Direct tool calls consume context for each definition and result. Agents scale better by writing code to call tools instead. Here's how it works with

MCP.

Published Nov 04, 2025

The Model Context Protocol (MCP) is an open standard for connecting AI agents to external systems. Connecting agents to tools and data traditionally requires a custom integration for each pairing, creating fragmentation and duplicated effort that makes it difficult to scale truly connected systems. MCP provides a universal protocol—developers implement MCP once in their agent and it unlocks an entire ecosystem of integrations.

Since launching MCP in November 2024, adoption has been rapid: the community has built thousands of MCP servers, SDKs are available for all major programming languages, and the industry has adopted MCP as the de-facto standard for connecting agents to tools and data.

Today developers routinely build agents with access to hundreds or thousands of tools across dozens of MCP servers. However, as the number of connected tools grows, loading all tool definitions upfront and passing intermediate results through the context window slows down agents and increases costs.

In this blog we'll explore how code execution can enable agents to interact with MCP servers more efficiently, handling more tools while using fewer tokens.

Excessive token consumption from tools makes agents less efficient

As MCP usage scales, there are two common patterns that can increase agent cost and latency:

1. Tool definitions overload the context window;
2. Intermediate tool results consume additional tokens.

1. Tool definitions overload the context window

Most MCP clients load all tool definitions upfront directly into context, exposing them to the model using a direct tool-calling syntax. These tool definitions might look like:

```
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
```

Copy

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

Copy

Tool descriptions occupy more context window space, increasing response time and costs. In cases where agents are connected to thousands of tools, they'll need to process hundreds of thousands of tokens before reading a request.

2. Intermediate tool results consume additional tokens

Most MCP clients allow models to directly call MCP tools. For example, you might ask your agent: "Download my meeting transcript from Google Drive and attach it to the Salesforce lead."

The model will make calls like:

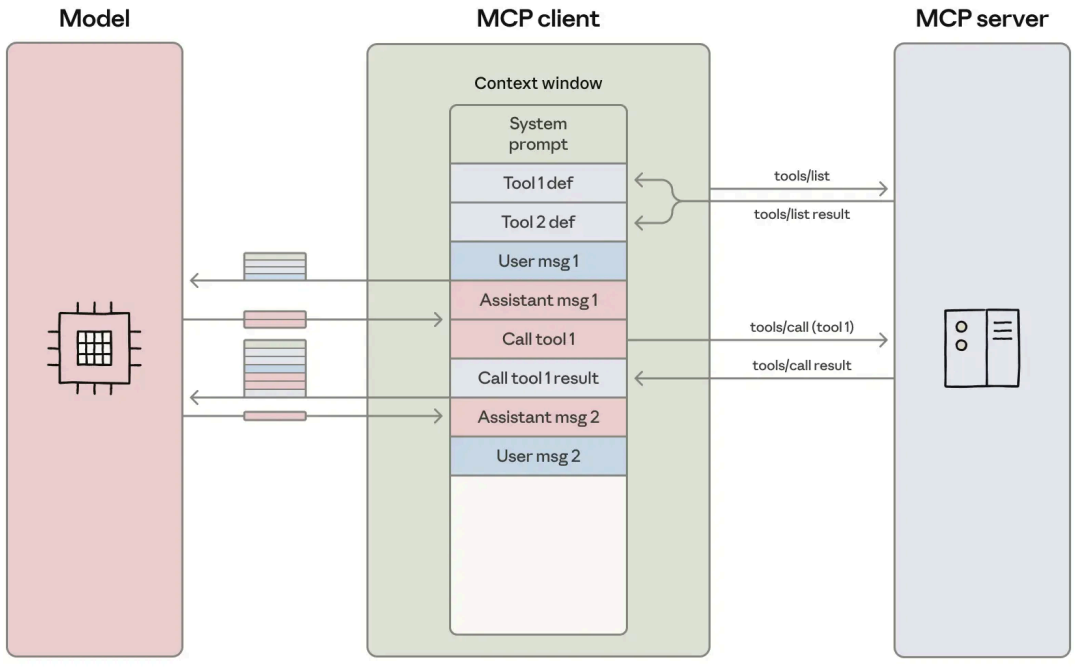
```
TOOL CALL: gdrive.getDocument(documentId: "abc123")
        → returns "Discussed Q4 goals...\n[full transcript text]"
           (loaded into model context)

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)
```

Copy

Every intermediate result must pass through the model. In this example, the full call transcript flows through twice. For a 2-hour sales meeting, that could mean processing an additional 50,000 tokens. Even larger documents may exceed context window limits, breaking the workflow.

With large documents or complex data structures, models may be more likely to make mistakes when copying data between tool calls.



The MCP client loads tool definitions into the model's context window and orchestrates a message loop where each tool call and result passes through the model between operations.

Code execution with MCP improves context efficiency

With code execution environments becoming more common for agents, a solution is to present MCP servers as code APIs rather than direct tool calls. The agent can then write code to interact with MCP servers. This approach addresses both challenges: agents can load only the tools they need and process data in the execution environment before passing results back to the model.

There are a number of ways to do this. One approach is to generate a file tree of all available tools from connected MCP servers. Here's an implementation using TypeScript:

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

Copy

Then each tool corresponds to a file, something like:

```
// ./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<GetDocumentResponse> {
  return callMCPTool<GetDocumentResponse>('google_drive__get_document',
input);
```

Copy

Our Google Drive to Salesforce example above becomes the code:

```
// 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 }
});
```

Copy

The agent discovers tools by exploring the filesystem: listing the `./servers/` directory to find available servers (like `google-drive` and `salesforce`), then reading the specific tool files it needs (like `getDocument.ts` and `updateRecord.ts`) to understand each tool's interface. This lets the agent load only the definitions it needs for the current task. This reduces the token usage from 150,000 tokens to 2,000 tokens—a time and cost saving of 98.7%.

Cloudflare published similar findings, referring to code execution with MCP as “Code Mode." The core insight is the same: LLMs are adept at writing code and developers should take advantage of this strength to build agents that interact with MCP servers more efficiently.

Benefits of code execution with MCP

Code execution with MCP enables agents to use context more efficiently by loading tools on demand, filtering data before it reaches the model, and executing complex logic in a single step. There are also security and state management benefits to using this approach.

Progressive disclosure

Models are great at navigating filesystems. Presenting tools as code on a filesystem allows models to read tool definitions on-demand, rather than reading them all up-front.

Alternatively, a `search_tools` tool can be added to the server to find relevant definitions. For example, when working with the hypothetical Salesforce server used above, the agent searches for "salesforce" and loads only those tools that it needs for the current task. Including a detail level parameter in the `search_tools` tool that allows the agent to select the level of detail required (such as name only, name and description, or the full definition with schemas) also helps the agent conserve context and find tools efficiently.

Context efficient tool results

When working with large datasets, agents can filter and transform results in code before returning them. Consider fetching a 10,000-row spreadsheet:

```
// Without code execution - all rows flow through context
TOOL CALL: gdrive.getSheet(sheetId: 'abc123')
        → returns 10,000 rows in context to filter manually

// With code execution - filter in the execution environment
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
```

Copy

The agent sees five rows instead of 10,000. Similar patterns work for aggregations, joins across multiple data sources, or extracting specific fields—all without bloating the context window.

More powerful and context-efficient control flow

Loops, conditionals, and error handling can be done with familiar code patterns rather than chaining individual tool calls. For example, if you need a deployment notification in Slack, the agent can write:

```
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');
```

Copy

This approach is more efficient than alternating between MCP tool calls and sleep commands through the agent loop.

Additionally, being able to write out a conditional tree that gets executed also saves on “time to first token” latency: rather than having to wait for a model to evaluate an if-statement, the agent can let the code execution environment do this.

Privacy-preserving operations

When agents use code execution with MCP, intermediate results stay in the execution environment by default. This way, the agent only sees what you explicitly log or return, meaning data you don’t wish to share with the model can flow through your workflow without ever entering the model's context.

For even more sensitive workloads, the agent harness can tokenize sensitive data automatically. For example, imagine you need to import customer contact details from a spreadsheet into Salesforce. The agent writes:

```
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`);
```

Copy

The MCP client intercepts the data and tokenizes PII before it reaches the model:

```
// What the agent would see, if it logged the sheet.rows:
[
  { salesforceId: '00Q...', email: '[EMAIL_1]', phone: '[PHONE_1]', name:
'[NAME_1]' },
  { salesforceId: '00Q...', email: '[EMAIL_2]', phone: '[PHONE_2]', name:
'[NAME_2]' },
  ...
]
```

Copy

Then, when the data is shared in another MCP tool call, it is untokenized via a lookup in the MCP client. The real email addresses, phone numbers, and names flow from Google Sheets to Salesforce, but never through the model. This prevents the agent from accidentally logging or processing sensitive data. You can also use this to define deterministic security rules, choosing where data can flow to and from.

State persistence and skills

Code execution with filesystem access allows agents to maintain state across operations. Agents can write intermediate results to files, enabling them to resume work and track progress:

```
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);

// Later execution picks up where it left off
const saved = await fs.readFile('./workspace/leads.csv', 'utf-8');
```

Copy

Agents can also persist their own code as reusable functions. Once an agent develops working code for a task, it can save that implementation for future use:

```
// 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');
```

Copy

This ties in closely to the concept of Skills, folders of reusable instructions, scripts, and resources for models to improve performance on specialized tasks. Adding a SKILL.md file to these saved functions creates a structured skill that models can reference and use. Over time, this allows your agent to build a toolbox of higher-level capabilities, evolving the scaffolding that it needs to work most effectively.

Note that code execution introduces its own complexity. Running agent-generated code requires a secure execution environment with appropriate sandboxing, resource limits, and monitoring. These infrastructure requirements add operational overhead and security considerations that direct tool calls avoid. The benefits of code execution—reduced token costs, lower latency, and improved tool composition—should be weighed against these implementation costs.

Summary

MCP provides a foundational protocol for agents to connect to many tools and systems. However, once too many servers are connected, tool definitions and results can consume excessive tokens, reducing agent efficiency.

Although many of the problems here feel novel—context management, tool composition, state persistence—they have known solutions from software engineering. Code execution applies these established patterns to agents, letting them use familiar programming constructs to interact with MCP servers more efficiently. If you implement this approach, we encourage you to share your findings with the MCP community.

Acknowledgments

This article was written by Adam Jones and Conor Kelly. Thanks to Jeremy Fox, Jerome Swannack, Stuart Ritchie, Molly Vorwerck, Matt Samuels, and Maggie Vo for feedback on drafts of this post.