Code Execution with MCP: Concepts & Experimentation Guide

Created by Paul Brower on 2026-01-24 11:59 AM

Based on Anthropic Engineering: Code Execution with MCP

Overview

Traditional AI tool usage loads all tool definitions upfront into the context window, consuming significant tokens. **Code execution with MCP** takes a different approach: instead of exposing tool definitions directly to the model, it presents MCP integrations as programmatic APIs that agents invoke through generated code.

Key benefit: Token usage can drop from ~150,000 tokens to ~2,000 tokens (98.7% reduction) by loading only necessary tool definitions on demand.

Core Concepts

1. File-Based Tool Discovery

Tools are organized in a filesystem structure that agents navigate like a codebase:

```
servers/
├── google-drive/
│   ├── getDocument.ts
│   └── searchFiles.ts
├── salesforce/
│   ├── updateRecord.ts
│   └── query.ts
└── slack/
    ├── postMessage.ts
    └── getChannelHistory.ts
```

Each tool is a TypeScript file with typed interfaces. Agents explore this structure to find relevant tools rather than having everything preloaded.

2. Tool Wrapper Pattern

Individual tools wrap MCP calls with type definitions:

```typescript
export async function getDocument(
  input: GetDocumentInput
): Promise<GetDocumentResponse> {
  return callMCPTool<GetDocumentResponse>(
    'google_drive__get_document',
    input
  );
}
```

3. Agent-Generated Code

Instead of direct tool calls, agents write executable code that composes multiple tools:

```
// Fetch a document from Google Drive
const transcript = (await gdrive.getDocument({
  documentId: 'abc123'
})).content;

// Update a Salesforce record with the content
await salesforce.updateRecord({
  objectType: 'SalesMeeting',
  recordId: '00Q5f000001abcXYZ',
  data: { Notes: transcript }
});
```

4. In-Process Data Filtering

Large datasets are filtered within the execution environment, not passed through the model:

```
const allRows = await gdrive.getSheet({ sheetId: 'abc123' });
const pendingOrders = allRows.filter(row =>
  row["Status"] === 'pending'
);
console.log(`Found ${pendingOrders.length} pending orders`);
```

5. Control Flow in Code

Complex operations use standard programming constructs:

```
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));
}
```

6. State Persistence

Agents maintain state across executions using filesystem operations:

```
// Save intermediate results
const leads = await salesforce.query({
  query: 'SELECT Id, Email FROM Lead LIMIT 1000'
});
await fs.writeFile('./workspace/leads.csv', csvData);

// Later execution can resume from saved state
const saved = await fs.readFile('./workspace/leads.csv');
```

Security Considerations

Sandboxed Execution

Agent-generated code must run in a secure environment with:

- Process isolation
- Resource limits (CPU, memory, network)
- Monitoring and logging

Data Privacy via Tokenization
The MCP client can automatically tokenize PII before it reaches the model:

```
// Agent sees tokenized data
{ email: '[EMAIL_1]', phone: '[PHONE_1]', name: '[NAME_1]' }

// Real data flows between services untokenized
```

Deterministic Security Rules
Define explicit data flow policies specifying which systems can access particular data types.

Benefits Summary

Benefit	Description
Token Efficiency	Load only necessary tool definitions on demand
Scalability	Handle hundreds of tools across dozens of MCP servers
Reduced Latency	Loops and conditionals execute in code, not via repeated model calls
Complex Data Ops	Aggregations, joins, filtering happen in-process
Composability	Multiple MCP servers compose naturally through code

---

Experimentation Steps

Phase 1: Understand the Current MCP Setup
- ☐ **Step 1.1:** Review your existing MCP configuration

```
cat ~/.claude/settings.json
ls -la ~/.claude/memory/
```

- ☐ **Step 1.2:** Examine how your memory MCP servers work
  - Look at the tool definitions they expose
  - Understand the current direct tool-call pattern
- ☐ **Step 1.3:** Document current token usage patterns
  - Note how many tools are loaded in typical sessions
  - Identify scenarios where tool count becomes unwieldy

Phase 2: Create a Simple Tool Wrapper Library
- ☐ **Step 2.1:** Create a `servers/` directory structure in this project

```
mkdir -p servers/memory
```

- ☐ **Step 2.2:** Create TypeScript wrapper for memory server tools

```
// servers/memory/createEntities.ts
interface Entity {
```

```
  name: string;
  entityType: string;
  observations: string[];
}

export async function createEntities(entities: Entity[]): Promise<void> {
  // Wrapper that would call mcp__memory__create_entities
}
```

- ☐ **Step 2.3:** Create an index file that exports available tools

```
// servers/memory/index.ts
export * from './createEntities';
export * from './searchNodes';
export * from './readGraph';
```

Phase 3: Build a Code Execution Environment

- ☐ **Step 3.1:** Set up a sandboxed TypeScript execution environment
  - Consider using Docker or a Node.js sandbox library
  - Implement resource limits
- ☐ **Step 3.2:** Create a simple REPL that can execute tool wrapper code

```
// executor.ts
import * as memory from './servers/memory';

async function executeAgentCode(code: string) {
  // Sandbox and execute the code with access to tool wrappers
}
```

- ☐ **Step 3.3:** Test basic code execution with your wrappers

Phase 4: Implement Progressive Tool Discovery

- ☐ **Step 4.1:** Create a tool search/discovery interface

```
// discovery.ts
export async function searchTools(query: string): Promise<ToolDefinition[]> {
  // Search through servers/ directory for matching tools
}
```

- ☐ **Step 4.2:** Test discovering tools on-demand vs preloading all definitions
- ☐ **Step 4.3:** Measure token savings in a controlled experiment

Phase 5: Claude Code Integration Exploration

- ☐ **Step 5.1:** Explore how Claude Code's existing MCP integration works
  - Review `.mcp.json` configuration format
  - Understand how tools are exposed to the model
- ☐ **Step 5.2:** Prototype a "code execution" MCP server
  - Single tool: `execute_code` that runs TypeScript in sandbox
  - Access to tool wrapper library within sandbox
- ☐ **Step 5.3:** Test the pattern with a real workflow
  - Example: Query memory, filter results, update specific nodes

- Compare token usage vs direct tool calls

Phase 6: Advanced Patterns

- ☐ **Step 6.1:** Implement state persistence
  - Create a `workspace/` directory for intermediate results
  - Test resuming operations from saved state
- ☐ **Step 6.2:** Build reusable "skills"
  - Save working code snippets as functions
  - Create a skill library that grows over time
- ☐ **Step 6.3:** Explore PII tokenization
  - Implement a data sanitization layer
  - Test that sensitive data doesn't reach the model

Questions to Answer Through Experimentation

1. **Token Efficiency:** How much context can you save with on-demand tool loading?
2. **Latency Trade-offs:** Does code execution add latency vs direct tool calls for simple operations?
3. **Complexity Threshold:** At what number of tools does code execution become clearly beneficial?
4. **Security Model:** What sandboxing approach works best for your use case?
5. **Developer Experience:** How does this pattern feel compared to direct tool invocation in Claude Code?
6. **Composability:** What multi-tool workflows become easier with code execution?

Relevant to Your Claude Code Usage

Based on your current setup with memory MCP servers, here are specific opportunities:

1. **Memory Operations:** Complex queries across your knowledge graph could be written as code rather than multiple tool calls
2. **Batch Operations:** Creating/updating many entities could be a single code execution vs many individual calls
3. **Conditional Logic:** "If entity exists, update it; otherwise create it" is natural in code
4. **Data Transformation:** Filter and transform query results in-process before storing

---

Resources

- Model Context Protocol Documentation
- Anthropic Engineering Blog
- MCP Server Examples