Code Execution with MCP

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

Dramatically speed up AI-assisted batch operations by having the model write code instead of making individual tool calls.

This repository explores the Code Execution with MCP pattern from Anthropic's engineering blog, with working examples you can run yourself.

The Problem

When you ask an AI assistant to process records one at a time, it's painfully slow:

```
User: "Update the status on these 50 records"

AI: [calls tool to find record 1]
AI: [calls tool to update record 1]
AI: [calls tool to find record 2]
AI: [calls tool to update record 2]
... repeat 48 more times ...
```

Each tool call requires a full round-trip: model reasoning → API call → tool execution → response parsing → model reasoning again. In our testing, **each tool call takes ~4 seconds**. For 50 records with 2 calls each, that's **~7 minutes**.

The Solution

Instead of the AI calling tools one-by-one, have it write code that does the batch operation:

```
// One tool call that runs this code:
for (const id of recordIds) {
  const record = await memory.openNodes([id]);
  await memory.addObservations([{ entityName: id, contents: ['status:
processed'] }]);
}
```

Result: 50 records in 1 millisecond instead of 7 minutes.

Why This Matters

Benefit	Description
Speed	Eliminate model round-trips for iterative operations
Token Efficiency	Load fewer tool definitions into context (95% reduction)
Reduced Context Rot	Less back-and-forth means cleaner conversation history
Natural Composition	Loops, conditionals, and data transformations are natural in code
Scalability	Handle hundreds of tools without bloating context

When to Use Each Approach

Use Direct Tool Calls For	Use Code Execution For
Exploration and discovery	Batch operations (“for each X, do Y”)
One-off queries	Complex filtering and transformation
Learning the data structure	Multi-step workflows
Simple operations	Operations on many records

Experiments

Experiment 1: Batch Record Updates

Tests the performance difference between direct tool calls and code execution for iterative memory operations.

Results Summary

Metric	Direct Tool Calls	Code Execution	Improvement
5 iterations	42.7 seconds	<1 ms	42,000x faster
50 iterations	7 minutes	1 ms	430,000x faster
Tool calls per batch	2 per iteration	1 total	90-99% reduction
Tool tokens in context	6,000	300	95% reduction

Key Finding

Model round-trips are the bottleneck, not tool execution. Each direct tool call takes ~4.3 seconds (model reasoning + API latency + MCP execution). Code execution eliminates this for iterative operations.

Running the Experiments Yourself

Prerequisites

- Node.js 18+
- Claude Code CLI
- npm

Setup

1. **Clone the repository**

```
git clone https://github.com/yourusername/code-mode.git
cd code-mode
```

2. **Install dependencies**

```
npm install
```

3. **Build the project**

```
npm run build
```

4. **Configure MCP servers**

   Create or update `.mcp.json` in the project root:

```
{
  "mcpServers": {
    "code_mode_memory": {
      "type": "stdio",
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-memory"],
      "env": {
        "MEMORY_FILE_PATH": "/path/to/code-mode/code-mode-memory.jsonl"
      }
    },
    "code_executor": {
      "type": "stdio",
      "command": "node",
      "args": ["dist/mcp-server/index.js"],
      "cwd": "/path/to/code-mode"
    }
  }
}
```

   Replace `/path/to/code-mode` with your actual project path.

5. **Restart Claude Code** to pick up the new MCP configuration

Running Tests

Test the memory wrapper library:

```
npm test
```

Test the code executor:

```
npm run test:executor
```

Using the Code Executor in Claude Code

Once configured, you can use the `execute_code` tool in Claude Code:

```
// Example: Count active records
const graph = await memory.readGraph();
const activeCount = graph.entities.filter(entity =>
  entity.observations.some(obs => obs === "status: active")
).length;
console.log(`Active records: ${activeCount}`);
```

The code executes in a sandboxed environment with access to memory operations but restricted system access.

Project Structure

```
code-mode/
├── servers/memory/       # TypeScript wrappers for memory MCP tools
```

```
│   ├── types.ts          # Type definitions
│   ├── client.ts         # MCP client interface
│   ├── operations.ts     # Tool wrapper functions
│   └── index.ts          # Public exports
├── executor/             # Code execution sandbox
│   ├── sandbox.ts        # VM-based sandboxed execution
│   └── index.ts          # Executor entry point
├── mcp-server/           # MCP server exposing execute_code tool
│   └── index.ts          # Server implementation
├── test/                 # Test files
├── code-mode-memory.jsonl # Test data (50 sample records)
├── Experiment1.md        # Detailed experiment documentation
└── mcp-config.example.json # Example MCP configuration
```

How It Works

1. Tool Wrappers

Instead of exposing raw MCP tools, we create typed TypeScript wrappers:

```
// Direct MCP call (verbose, untyped)
mcp__code_mode_memory__read_graph()

// Wrapper (clean, typed)
await memory.readGraph()
```

2. Sandboxed Execution

Agent-generated code runs in a Node.js VM sandbox with:

- Access to memory tool wrappers
- Console output capture
- Timeout protection
- No filesystem or network access

3. Single Tool Call

The `execute_code` MCP tool accepts JavaScript code and runs it in the sandbox:

```
Model → execute_code("...loop over 50 records...") → Sandbox executes all
iterations → Single result
```

vs. direct tool calls:

```
Model → tool call → result → Model → tool call → result → ... (100 round trips)
```

Resources

- Anthropic Engineering: Code Execution with MCP
- Model Context Protocol Documentation
- MCP Server Examples

License

MIT