Experiment 1: Batch Record Updates with AI Assistance

Created by Paul Brower on 2026-01-24 12:00 PM

What This Document Is

This is a hands-on experiment comparing two approaches to AI-assisted batch operations:

1. **Direct Tool Calls** - The AI calls tools one at a time (the standard approach)
2. **Code Execution** - The AI writes code that runs all operations at once

We built a working implementation, measured real performance, and documented everything so you can reproduce the results yourself.

Bottom line: Code execution was **42,000x faster** for batch operations in our tests.

Quick navigation:

- Results Summary - The key numbers
- The Problem - Why this matters
- The Solution - How it works
- Detailed Metrics - Full measurements
- Implementation Steps - How we built it (Phases 1-6)
- Lessons Learned - What we discovered

TL;DR - Results

Metric	Direct Tool Calls	Code Execution	Improvement
5 record updates	42.7 seconds	<1 ms	**42,000x faster**
50 record updates	7 minutes	1 ms	**430,000x faster**
Tool calls needed	2 per record	1 total	**90-99% reduction**
Context tokens used	6,000	300	**95% reduction**

Why such a dramatic difference? Each direct tool call takes ~4.3 seconds (model thinking + API latency + tool execution). With code execution, the model thinks once, writes the code, and the sandbox runs all iterations instantly.

```
Direct:     model thinks → tool call → model thinks → tool call → ... (100 round trips)
Code exec:  model thinks → writes code → sandbox runs all 50 iterations → done
```

The Problem We're Solving

Batch operations are common in real work:

Domain	Task
CRM	"Mark all leads from last week's campaign as 'contacted'"

Domain	Task
DevOps	“Add a ‘health_check: passed’ note to each server”
Project Mgmt	“Tag these 50 tickets with the sprint number”
Data Migration	“For each customer in System A, update their record in System B”

When you ask an AI assistant to do this, it processes records one at a time:

```
find record 1 → update record 1 → find record 2 → update record 2 → ...
```

Each step is a full round-trip through the AI. At ~4 seconds per tool call, 50 records takes **7 minutes**. Scale to 500 records and you’re waiting **over an hour**.

The Solution: Code Execution

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

```
for (const id of recordIds) {
    const record = await memory.openNodes([id]);
    await memory.addObservations([{ entityName: id, contents: ['status: processed'] }]);
}
```

This runs in a sandbox - one tool call, all iterations execute locally.

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

Technical Background

This experiment implements the “Code Execution with MCP” pattern from Anthropic’s engineering blog.

How it works:

1. Create TypeScript wrappers for MCP tools (type-safe, clean API)
2. Build a sandboxed execution environment
3. Expose a single `execute_code` tool that runs JavaScript with access to the wrappers

Benefits beyond speed:

- Token efficiency (fewer tool definitions in context)
- Better scalability with many tools/integrations
- In-process data filtering (large datasets don’t pass through the model)

Reproducibility: All operations use the `code_mode_memory` MCP server with test data in `code-mode-memory.jsonl`. See the README for setup instructions.

Detailed Measurements

> `[Token Efficiency]` This section tracks quantitative measurements to validate (or invalidate) the claimed benefits of code execution with MCP.

Baseline Measurements (Before)

Captured during Phase 1.

Metric	Value	Notes
Tool Definitions Loaded	31	20 built-in + 2 IDE + 9 MCP memory
Estimated Tokens per Tool	150-300	Varies by complexity; memory tools on simpler end
Total Tool Definition Tokens	5,000-9,000	Conservative estimate for current setup
Context Window Usage (%)	2.5-4.5%	Based on 200K context window

Scaling projection (if adding common integrations):

Scenario	Tools	Est. Tokens	Context %
Current	31	6,000	3%
+Google Drive	46	10,000	5%
+Slack	58	13,000	6.5%
+GitHub	78	18,000	9%
+Salesforce	93	22,000	11%

Sample Task: Memory Query + Filter + Update

Task performed: Read graph → Create 3 test entities → Search for “active” → Add observations to entity → Delete test entities

Metric	Value	Notes
Tool Calls Made	5	read_graph, create_entities, search_nodes, add_observations, delete_entities
Model Decisions	5	Each tool call requires model to decide which tool + parameters
Data Passed Through Model	All	Every result returned to model for next decision

Tool call sequence:

```
1. mcp__code_mode_memory__read_graph        → returns full graph
2. mcp__code_mode_memory__create_entities    → creates 3 entities
3. mcp__code_mode_memory__search_nodes       → returns matching entities
4. mcp__code_mode_memory__add_observations   → updates entity
5. mcp__code_mode_memory__delete_entities    → cleans up
```

Observation: With direct tool calls, every intermediate result passes through the model’s context. In a code execution pattern, steps 2-5 could be a single code block that runs in a sandbox, with only the final result returned to the model.

Scaled Baseline: Iterative Operations

Purpose: Demonstrate how direct tool calls scale with iteration count.

Setup: Created 50 test records (Record_001 through Record_050) with attributes:

- `count: 0`
- `status: active|inactive`
- `category: A|B|C`

Test: “Find record by ID, increment count” × 5 iterations

Phase	Tool Calls	Details
Data setup	5	create_entities × 5 (batches of 10 records)
Iteration 1	2	open_nodes(Record_007) → add_observations(count: 1)
Iteration 2	2	open_nodes(Record_023) → add_observations(count: 1)
Iteration 3	2	open_nodes(Record_041) → add_observations(count: 1)
Iteration 4	2	open_nodes(Record_015) → add_observations(count: 1)
Iteration 5	2	open_nodes(Record_033) → add_observations(count: 1)
Total	**15**	

Timed Baseline Test:

Metric	Value
Start timestamp	1769369149226 ms
End timestamp	1769369191943 ms
Total elapsed	**42,717 ms (42.7 seconds)**
Iterations	5
Tool calls	10
Time per iteration	**8,543 ms (8.5 sec)**
Time per tool call	**4,272 ms (4.3 sec)**

Time includes: model reasoning, API latency, MCP tool execution

Scaling projection (direct tool calls):

Iterations	Tool Calls	Est. Time	Practical?
5	10	43 sec	Yes
50	100	7 min	Slow, expensive
100	200	14 min	Impractical
1000	2000	2.4 hours	Impossible

Key insight: With direct tool calls, iteration count directly multiplies tool calls AND time. With code execution, ANY number of iterations is a single `execute_code` call:

```
// This runs entirely in sandbox - 1 tool call regardless of iteration count
for (let i = 1; i <= 1000; i++) {
  const record = await memory.openNodes([`Record_${i.toString().padStart(3, '0')}
`]);
```

```
  const currentCount = parseInt(record.observations.find(o =>
o.startsWith('count:'))?.split(': ')[1] || '0');
  await memory.addObservations([{
    entityName: record.name,
    contents: [`count: ${currentCount + 1}`]
  }]);
}
```

After Measurements (Code Execution Pattern)

Captured after implementing the code execution sandbox.

Metric	Value	Notes
Tool Definitions Loaded	1	Only execute_code needed
Estimated Tokens per Tool	200-400	Depends on schema complexity
Total Tool Definition Tokens	300	Down from 6,000
Context Window Usage (%)	0.15%	Down from 3%

Same Task: Memory Query + Filter + Update (via Code Execution)

Metric	Value	Notes
Tool Calls Made	1	Single executeCode() call
Code Execution Time	0-1 ms	All iterations run in sandbox
Wall Clock Time (5 iter)	<1 sec	vs 42.7 sec direct
Wall Clock Time (50 iter)	<1 sec	vs 7 min direct

Comparison Summary

Metric	Before (Direct)	After (Code Exec)	Improvement
Tool tokens in context	6,000	300	**95% reduction**
Tool calls (5 iterations)	10	1	**90% reduction**
Wall clock time (5 iter)	42,717 ms	<1 ms	**42,000x faster**
Wall clock time (50 iter)	430,000 ms	1 ms	**430,000x faster**
Context window usage	3%	0.15%	**95% reduction**

Key Findings

1. **Model round-trips are the bottleneck** - Each direct tool call takes ~4.3 seconds (model reasoning + API latency + MCP execution). Code execution eliminates this for iterative operations.
2. **Scaling is dramatically different:**
   - Direct: O(n) - time grows linearly with iterations
   - Code exec: O(1) - constant time regardless of iteration count
3. **Token savings compound** - Fewer tool definitions in context means more room for actual work, and each tool call avoided saves the tokens for that call's input/output.

Concept Glossary

> **Annotation Key:**
>
> - `[MCP Core]` - Fundamental MCP protocol concepts
> - `[Token Efficiency]` - Related to context window / token usage optimization
> - `[Architecture]` - System design and structure patterns
> - `[Security]` - Sandboxing, isolation, data privacy
> - `[Practical]` - Directly applicable to your current Claude Code usage

Term	Definition	Relevance
MCP (Model Context Protocol)	A protocol that allows AI models to interact with external tools and data sources through a standardized interface	[MCP Core] The foundation everything else builds on
Context Window	The total amount of text (measured in tokens) an AI can “see” at once	[Token Efficiency] Limited resource; loading tools consumes part of it
Tool Definition	The schema describing what a tool does, its parameters, and return types	[Token Efficiency] Each tool definition consumes tokens; 100+ tools = significant overhead
Direct Tool Call	Traditional pattern: model sees all tool definitions, picks one, calls it directly	[Architecture] Current default in Claude Code
Code Execution Pattern	New pattern: model writes code that calls tools programmatically	[Architecture] What this experiment explores
Tool Wrapper	A TypeScript function that wraps an MCP tool call with type definitions	[Architecture] Makes tools callable from generated code
Sandboxed Execution	Running agent-generated code in an isolated environment with resource limits	[Security] Required for safe code execution
PII Tokenization	Replacing sensitive data with tokens before it reaches the model	[Security] Privacy protection technique

---

Implementation Steps

Phase 1: Understand the Current MCP Setup

Purpose: Establish baseline understanding of how MCP works in your current environment.

> **Metrics Note:** Capture all “Before” measurements in the Performance Metrics section during this phase.

Step 1.1: Review existing MCP configuration

- ☑ **Status:** Complete

- **Action:** Examine MCP settings and memory server configurations
- **Outcome:**

  Project MCP Configuration (`.mcp.json` in project root):

  - `code_mode_memory` - using `@modelcontextprotocol/server-memory`
  - `code_executor` - the code execution MCP server built in this experiment

  Configuration hierarchy:

  ```
  ~/.claude.json (global)          → defines globally available MCP servers
  .mcp.json (project root)         → defines project-specific servers
  ```

- **Annotations:** > `[MCP Core]` The `.mcp.json` file defines which MCP servers are available to Claude Code. Each server exposes tools that appear in the model's context. > > `[Architecture]` The layered config (global → project) allows shared servers while enabling project-specific additions.

Step 1.2: Examine how memory MCP servers work

- ☑ **Status:** Complete
- **Action:** Look at tool definitions exposed by memory servers
- **Outcome:**

  Tools per `@modelcontextprotocol/server-memory` instance (9 tools): | Tool | Purpose | |——|———| | `create_entities` | Create new entities in the knowledge graph | | `create_relations` | Create relationships between entities | | `add_observations` | Add observations to existing entities | | `delete_entities` | Remove entities from the graph | | `delete_observations` | Remove specific observations | | `delete_relations` | Remove relationships | | `read_graph` | Read the entire knowledge graph | | `search_nodes` | Search for nodes by query | | `open_nodes` | Retrieve specific nodes by name |

  This experiment uses:

  - `mcp__code_mode_memory__*` (9 tools) - project-specific memory for test data

  Observation: Each memory server instance adds 9 tool definitions to the context. The schemas include parameter types, descriptions, and return types - significant token overhead per tool.

- **Annotations:** > `[MCP Core]` The `@modelcontextprotocol/server-memory` package exposes tools like `create_entities`, `search_nodes`, `read_graph`, etc. Each of these has a schema that gets loaded into context. > > `[Token Efficiency]` Every tool definition has overhead. Even a simple tool might be 100-200 tokens of schema. With multiple servers, this adds up. > > `[Practical]` With just one memory server you have 9 MCP tools. Add more servers (Slack, Google Drive, Salesforce) and you quickly reach 50-100+ tools, consuming significant context.

Step 1.3: Document current token usage patterns

- ☑ **Status:** Complete
- **Action:** Note how many tools are loaded, identify when it becomes unwieldy
- **Outcome:**

  Tool Count (typical Claude Code session): | Category | Tools | Count | |———-|——-|——-| | Claude Code Built-in | Bash, Glob, Grep, Read, Edit, Write, NotebookEdit, WebFetch, WebSearch, Task, TaskOutput, AskUserQuestion, Skill, EnterPlanMode, ExitPlanMode, TaskCreate, TaskGet, TaskUpdate, TaskList, TaskStop | 20 | | IDE MCP | getDiagnostics, executeCode | 2 | | Memory MCP (project) | create_entities, create_relations, add_observations, delete_entities, delete_observations, delete_relations, read_graph, search_nodes, open_nodes | 9 | | **Total** | | **31** |

Scaling Consideration: If you added common integrations:

- Google Drive MCP: +10-15 tools
- Slack MCP: +8-12 tools
- GitHub MCP: +15-20 tools
- Salesforce MCP: +10-15 tools
- → Could easily reach **80-100+ tools**

- **Annotations:** > `[Token Efficiency]` This establishes the "before" measurement. You'll compare against this after implementing code execution. > > `[Practical]` Signs of tool overload: slow responses, context limit warnings, model confusion about which tool to use. > > `[Token Efficiency]` At ~150-300 tokens per tool definition, 100 tools = 15,000-30,000 tokens just for tool schemas. That's significant context overhead before any actual work begins.

Phase 2: Create a Simple Tool Wrapper Library

Purpose: Build the infrastructure that makes tools callable from generated code.

Step 2.1: Create servers/ directory structure

- ☑ **Status:** Complete
- **Action:** `mkdir -p servers/memory`
- **Outcome:**

  Created directory structure:

```
servers/
└── memory/
    ├── types.ts      # Type definitions
    ├── client.ts     # MCP client interface + mock
    ├── operations.ts # Tool wrapper functions
    └── index.ts      # Public exports
```

  Also initialized npm project with TypeScript:

  - `package.json` - ES module config, build scripts
  - `tsconfig.json` - TypeScript configuration

- **Annotations:** > `[Architecture]` The filesystem becomes a "discovery mechanism." Instead of loading all tools, the agent navigates directories to find what it needs - similar to how developers explore a codebase. > > `[Token Efficiency]` Only the tools the agent actually needs get loaded into context.

Step 2.2: Create TypeScript wrappers for memory server tools

- ☑ **Status:** Complete
- **Action:** Write typed wrapper functions for each memory tool
- **Outcome:**

  Created wrappers for all 9 memory server tools:

Wrapper Function	MCP Tool
readGraph()	read_graph
createEntities()	create_entities
createRelations()	create_relations

Wrapper Function	MCP Tool
addObservations()	add_observations
deleteEntities()	delete_entities
deleteObservations()	delete_observations
deleteRelations()	delete_relations
searchNodes()	search_nodes
openNodes()	open_nodes

Also created:

- `MCPClientInterface` - Abstract interface for MCP clients
- `MockMCPClient` - In-memory mock for testing without MCP server
- Full TypeScript types for all inputs/outputs

- **Annotations:** > `[Architecture]` Wrappers provide: (1) Type safety for inputs/outputs, (2) Clean API for generated code, (3) Abstraction over raw MCP calls. > > `[Practical]` Example: `await memory.createEntities([...])` is cleaner than a raw MCP tool call with full schema. > > `[Architecture]` The MockMCPClient allows testing the code execution pattern without needing an actual MCP server running.

Step 2.3: Create index file exporting available tools

- ☑ **Status:** Complete
- **Action:** Create `servers/memory/index.ts` with exports
- **Outcome:**

  Created index that exports:

  - All type definitions
  - Client utilities (`setMCPClient` , `MockMCPClient`)
  - All 9 operation functions

  Usage in generated code:

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

  setMCPClient(new MockMCPClient());

  const graph = await memory.readGraph();
  await memory.createEntities([...]);
  ```

  Test results: `npm run build && npm test` passes:

  ```
  ✓ Mock MCP client initialized
  ✓ Created 4 entities
  ✓ Graph has 4 entities, 0 relations
  ✓ Found 4 matching entities
  ✓ Updated 5 entities in 0ms (With direct tool calls, this would be 10 model round-trips)
  ✓ Cleaned up. Graph now has 0 entities
  === All tests passed ===
  ```

- **Annotations:** > `[Architecture]` Standard TypeScript module pattern. Allows `import * as memory from './servers/memory'` for clean namespace access. > > `[Practical]` The test demonstrates the key benefit: 5 iterations of find+update completed in a single code block (0ms), vs 10 model round-trips with direct tool calls.

Phase 3: Build a Code Execution Environment

Purpose: Create a safe place to run agent-generated code.

Step 3.1: Set up sandboxed TypeScript execution

- ☑ **Status:** Complete
- **Action:** Choose and implement sandboxing approach (Docker, Node sandbox, etc.)
- **Outcome:**

  Chose: Node.js `vm` module - lighter weight, good for demonstrating the pattern.

  Created `executor/sandbox.ts`:

  - `executeCode(code, options)` - Runs JS code in VM context
  - Timeout protection (default 30s)
  - Captures console output
  - Returns `{ success, output, error, elapsedMs }`

  Sandbox provides access to:

  - All memory tool wrappers (`memory.readGraph()`, etc.)
  - Console (`console.log`, etc.)
  - Core JS globals (Promise, JSON, Array, etc.)

  Does NOT provide:

  - File system access
  - Network access
  - Process/child_process
  - require/import

- **Annotations:** > `[Security]` Critical step. Agent-generated code could do anything - file system access, network calls, infinite loops. Sandbox provides: process isolation, resource limits (CPU/memory/time), restricted filesystem access. > > `[Architecture]` Options: Docker containers (strongest isolation), VM2/isolated-vm (Node.js sandboxes), Deno with permissions. > > `[Practical]` Node's `vm` module is NOT fully secure - it's sufficient for this experiment but production would need stronger isolation.

Step 3.2: Create REPL for executing tool wrapper code

- ☑ **Status:** Complete
- **Action:** Build executor that runs code with access to tool wrappers
- **Outcome:**

  Created `executor/index.ts` - exports `executeCode()` function.

  Usage:

  ```
  import { executeCode } from './executor';

  const result = await executeCode(`
    const graph = await memory.readGraph();
    console.log('Found', graph.entities.length, 'entities');
  ```

```
`);
// result.output: ["Found 50 entities"]
// result.elapsedMs: 1
```

- **Annotations:** > `[Architecture]` This is the "runtime" where generated code executes. It needs access to your tool wrappers but restricted system access.

Step 3.3: Test basic code execution

- ☑ **Status:** Complete
- **Action:** Run simple test cases through the executor
- **Outcome:**

  Test results (`npm run test:executor`):

Test	Direct Tool Calls	Code Execution	Speedup
5 iterations	42,717 ms	**0 ms**	∞
50 iterations	430,000 ms	**1 ms**	430,000x

  Complete test output:

  ```
  ✓ Loaded 50 test records into mock client
  ✓ Simple execution: Entity count: 50 (1ms)
  ✓ 5 iterations find+update: 0ms (vs ~43 sec direct)
  ✓ 50 iterations: 1ms (vs ~7 min direct)
  === All tests passed ===
  ```

  Key insight: The model round-trip time (~4 sec per tool call) is completely eliminated for iterative operations. All iterations execute in the sandbox; model only invokes `executeCode` once.

Phase 4: Implement Progressive Tool Discovery

Purpose: Enable on-demand tool loading instead of upfront loading.

Step 4.1: Create tool search/discovery interface

- ☐ **Status:** Not started
- **Action:** Build function to search servers/ directory for tools matching a query
- **Outcome:** *To be filled*
- **Annotations:** > `[Token Efficiency]` This is the key innovation. Instead of "here are 50 tools," it's "search for tools related to 'memory'" → only returns relevant ones. > > `[Architecture]` Discovery can use: filename matching, JSDoc comments, tool metadata files.

Step 4.2: Test on-demand vs preloaded discovery

- ☐ **Status:** Not started
- **Action:** Compare the two approaches in practice
- **Outcome:** *To be filled*

Step 4.3: Measure token savings

- ☐ **Status:** Not started
- **Action:** Quantify actual token reduction in controlled test
- **Outcome:** *To be filled*
- **Annotations:** > `[Token Efficiency]` This is your "after" measurement. Compare against Phase 1.3 baseline.

Phase 5: Claude Code Integration Exploration

Purpose: Understand how this pattern could work with Claude Code specifically.

> **Metrics Note:** Capture all "After" measurements in the Performance Metrics section during this phase, using the same sample task from Phase 1.

Step 5.1: Explore Claude Code's MCP integration

- ☑ **Status:** Complete
- **Action:** Review .mcp.json format, understand tool exposure mechanism
- **Outcome:**

  Claude Code MCP configuration uses `.mcp.json` in project root:

  ```
  {
    "mcpServers": {
      "server_name": {
        "type": "stdio",
        "command": "node",
        "args": ["path/to/server.js"]
      }
    }
  }
  ```

  Each server's tools appear as `mcp__servername__toolname` in Claude's context.
- **Annotations:** > `[MCP Core]` Claude Code reads .mcp.json to know which MCP servers to connect to. Each server's tools become available to the model.

Step 5.2: Prototype "code execution" MCP server

- ☑ **Status:** Complete
- **Action:** Create MCP server with single `execute_code` tool
- **Outcome:**

  Created `mcp-server/index.ts`:

  Tools exposed: | Tool | Description | |—–|———-| | `execute_code` | Run JavaScript with access to memory operations | | `get_record_count` | Debug helper to check record count |

  Usage:

  ```
  {
    "mcpServers": {
      "code_executor": {
        "type": "stdio",
        "command": "node",
        "args": ["dist/mcp-server/index.js"],
        "cwd": "/path/to/code-mode"
      }
    }
  }
  ```

  Tool comparison: | Approach | Tools in Context | |———-|—————-| | Direct memory server | 9 tools | | Code executor server | 2 tools (execute_code + debug helper) |

- **Annotations:** > `[Architecture]` Meta-level: instead of exposing 50 tools, expose 1 tool (`execute_code`) that can call all 50 through generated code. > > `[Token Efficiency]` One tool definition in context vs fifty. The 49 tools become accessible via code, not context.

Step 5.3: Test with real workflow

- ☑ **Status:** Complete
- **Action:** Example: Query memory → filter results → update nodes. Compare approaches.
- **Outcome:**

  Test performed: Count records with `status: active` using code executor

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

  Results: | Metric | Value | |——-|——-| | Tool calls | 1 (`execute_code`) | | Execution time | 2 ms | | Records processed | 50 | | Active records found | 34 |

  Comparison: With direct tool calls, this would require:

  - 1 `read_graph` call + model processing to filter
  - OR 50 individual `open_nodes` calls to check each record

  The code execution pattern handled the entire query+filter operation in a single tool call.

Phase 6: Advanced Patterns

Purpose: Explore sophisticated applications of the pattern.

Step 6.1: Implement state persistence

- ☐ **Status:** Not started
- **Action:** Create workspace/ directory, test saving/resuming operations
- **Outcome:** *To be filled*
- **Annotations:** > `[Architecture]` Code execution enables multi-step workflows that span multiple model interactions. Save intermediate results to filesystem, resume later. > > `[Practical]` Example: Query 1000 records, save to CSV, later execution filters and updates specific ones.

Step 6.2: Build reusable "skills"

- ☐ **Status:** Not started
- **Action:** Save working code snippets as reusable functions
- **Outcome:** *To be filled*
- **Annotations:** > `[Architecture]` Skills = tested, working code that accomplishes a specific task. Library grows over time. > > `[Practical]` Instead of re-generating code each time, import a skill: `import { backupAllEntities } from './skills/memory'`

Step 6.3: Explore PII tokenization

- ☐ **Status:** Not started
- **Action:** Implement data sanitization layer
- **Outcome:** *To be filled*
- **Annotations:** > `[Security]` Real data flows between services; model only sees tokenized versions (`[EMAIL_1]`, `[NAME_1]`). Protects privacy while maintaining usefulness.

Key Questions Being Investigated

Question	Status	Metric Reference	Finding
How much context can you save with on-demand tool loading?	**Answered**	Tool Definition Tokens (Before vs After)	95% reduction (6,000 → 300 tokens)
Does code execution add latency for simple operations?	**Answered**	Wall Clock Time comparison	No - execution is 1-2ms; model reasoning is the bottleneck
At what number of tools does code execution become clearly beneficial?	**Partially answered**	Context Window Usage (%)	Beneficial at any scale; dramatic for iterative operations
What sandboxing approach works best?	**Answered**	Code Execution Time, qualitative	Node.js vm module sufficient for experiments; production needs stronger isolation
How does this pattern feel compared to direct tool invocation?	**Answered**	Qualitative (DX notes below)	Natural - feels like writing normal code
What multi-tool workflows become easier?	**Answered**	API Calls, Total Tokens	Batch operations: query+filter+update in one call vs N calls

Developer Experience Notes

Qualitative observations about using each approach.

Direct Tool Calls (Current)

- Familiar pattern - works like any other Claude Code tool
- Each operation visible in conversation (good for understanding, verbose for bulk ops)

- Model decides each step - flexible but slow for repetitive tasks
- Natural for exploratory work, ad-hoc queries
- Tool selection overhead when many tools available

Code Execution Pattern (New)

- Feels like writing normal JavaScript - low friction
- Single tool call for complex operations - cleaner conversation
- Code is explicit - easy to verify what will happen before execution
- Fast feedback loop (2ms execution)
- Requires knowing the API (memory.readGraph, etc.) - slight learning curve
- Batch operations feel natural: “write code that does X to all records matching Y”
- Debugging via console.log output works well

Lessons Learned

Technical Lessons

- **Model round-trips dominate latency** - Tool execution is fast (ms); model reasoning + API latency is slow (~4 sec per call). Code execution eliminates this for iterative work.
- **Node.js `vm` module works for prototyping** - Sufficient isolation for experiments, but not production-ready security. Real deployments need Docker or similar.
- **MCP server creation is straightforward** - The `@modelcontextprotocol/sdk` makes it easy to expose custom tools to Claude Code.

Conceptual Insights

- **The pattern inverts the tool paradigm** - Instead of “model picks from N tools,” it’s “model writes code using a programmable API.” This scales better.
- **Code is more expressive than tool sequences** - Loops, conditionals, variable reuse - all natural in code, awkward in sequential tool calls.
- **O(1) vs O(n) scaling** - Direct tool calls scale linearly with iterations. Code execution is constant time regardless of iteration count.

Practical Takeaways

- **Use code execution for batch operations** - Any “for each X, do Y” task benefits dramatically.
- **Use direct tools for exploration** - Ad-hoc queries, understanding data structure, one-off operations.
- **The approaches complement each other** - Not either/or. Use direct tools to understand the data, then code execution to operate on it at scale.

Resources

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