mini-claude: A Complete Agentic AI Shell in ~750 Lines of Bash

What It Is

`mini-claude.sh` is a self-contained, fully functional agentic coding assistant that runs entirely as a Bash script. It connects to Anthropic's Claude API and gives the model three tools — `bash`, `read_file`, and `write_file` — enabling it to autonomously execute commands, inspect files, write code, and chain multiple actions together to complete complex tasks. The entire thing is a single file with two dependencies: `curl` and `jq`.

It is, in essence, a from-scratch reimplementation of the core agentic loop that powers tools like Claude Code, Cursor, and Aider — stripped down to its absolute fundamentals and written in the most transparent language possible.

How It Works

The Agentic Loop

The heart of mini-claude is a nested loop:

```
User types a message
  └→ Send conversation history + tool definitions to Claude API
       └→ Claude responds with text and/or tool_use requests
            ├→ If text only: print it, wait for next user message
            └→ If tool_use: execute the tool, feed result back to Claude
                 └→ Claude sees the result and responds again
                      └→ (repeat until Claude stops calling tools)
```

This is the same fundamental pattern used by every agentic coding tool. Claude decides what to do, does it through tool calls, sees the results, and decides what to do next. A single user message like "find all TODO comments and create a summary" might trigger dozens of tool calls — `bash` to run `grep`, `read_file` to inspect matches, `write_file` to create the summary — all without the user intervening.

The Three Tools

Tool	What Claude Sees	What Actually Happens
`bash`	"Run a bash command and return stdout/stderr"	Executes via `bash -c`, captures combined output
`read_file`	"Read a file and return its contents"	`cat` with error handling
`write_file`	"Write content to a file"	`printf` to file, creates parent directories

These three primitives are sufficient for Claude to perform virtually any coding task. It can install packages, run test suites, edit configuration files, compile code, query

databases, manage git repositories — anything expressible as shell commands and file operations.

Permission Model

By default, every `bash` command and `write_file` operation requires explicit user confirmation:

```
[tool] bash: rm -rf /tmp/test
Allow? [y/N]
```

The `--dangerously-skip-permissions` flag disables all confirmation prompts, allowing Claude to operate fully autonomously. The `read_file` tool never requires confirmation — reading is always safe.

Authentication

Mini-claude supports two authentication methods, tried in order:

1. **`$ANTHROPIC_API_KEY`** — a standard Anthropic API key, sent as `x-api-key`
2. **Claude Code OAuth** — reads the token from `~/.claude/.credentials.json` (the same credentials file used by Claude Code), checks expiry, and authenticates via Bearer token with the appropriate beta header

This means if you've already run `claude login`, mini-claude works immediately with no additional setup.

Session Storage: The Filesystem as Database

This is where mini-claude diverges most interestingly from conventional design. Rather than storing conversation history as a JSON blob or in a database, every message is a directory on the filesystem:

```
~/.mini-claude/sessions/20250222-120000/
  00000-user/
    text.md                    ← "Find all Python files with syntax errors"
  00001-assistant/
    text.md                    ← "I'll search for Python files and check..."
    tool_use.json            ← [{id: "toolu_...", name: "bash", input: {...}}]
  00002-user/
    tool_result.json           ← [{tool_use_id: "toolu_...", content: "..."}]
  00003-assistant/
    text.md                    ← "Found 3 files with issues. Let me fix..."
    tool_use.json
  00004-user/
    tool_result.json
  00005-assistant/
    text.md                    ← "All fixed. Here's what I changed..."
  .last_payload.json           ← the most recent API request (debug artifact)
  .last_response.json          ← the most recent API response (debug artifact)
```

```
summary.md              ← compaction summaries (if conversation was compacted)
```

Each message is a numbered directory (`NNNNN-role`) containing plain text files. User text goes in `text.md`. Assistant text goes in `text.md`, with any tool calls in `tool_use.json`. Tool results go in `tool_result.json`. The numbered prefixes ensure correct ordering.

This design has several consequences:

- **Sessions are human-readable.** You can `ls` a session directory and immediately see the conversation flow. You can `cat` any message. You can `grep` across all sessions.
- **Sessions are debuggable.** The `.last_payload.json` and `.last_response.json` files show exactly what was sent to and received from the API on the most recent turn.
- **Sessions survive crashes.** Each message is written to disk immediately. If the script dies mid-conversation, everything up to that point is preserved.
- **Sessions are editable.** You can delete a message directory, modify `text.md`, or even inject new messages by creating directories — then resume the session.
- **No large strings in memory.** When building the API payload, each message is assembled into a temporary JSON file, then all files are merged with a single `jq -s` call. The conversation never needs to exist as a single shell variable.

Session Resume

On startup, mini-claude offers to resume the most recent session:

```
Last session: 20250222-120000 (47 messages). Resume? [Y/n]
```

The `/sessions` command lists all saved sessions with message counts and compaction status, allowing you to switch between them.

Session Repair

When loading a session, mini-claude runs a repair pass that handles:

- **Orphaned tool_use messages** — an assistant message requesting tool calls but with no corresponding tool_result following it (e.g. from a crash during tool execution). These are deleted.
- **Mismatched tool IDs** — tool_result messages whose IDs don't match the preceding tool_use. Both messages are deleted.
- **Consecutive same-role messages** — two user or two assistant messages in a row (which the API rejects). These are merged.

Conversation Compaction

As conversations grow, they consume more tokens and eventually approach context window limits. Mini-claude handles this through compaction — an approach inspired by Claude Code's own context management.

How Compaction Works

1. **Trigger**: either manually via `/compact`, or automatically when total session file size exceeds 320,000 characters (~80k tokens)
2. **Split point**: the conversation is divided at a clean boundary — a user text message (not a tool result) near the point that preserves the last 10 messages verbatim
3. **Summarization**: the older portion is sent to Claude with a prompt asking for a factual summary capturing tasks, decisions, outcomes, file paths, commands, and current state
4. **Replacement**: the older messages are deleted from disk, the kept messages are renumbered, and a synthetic pair is inserted at the start:
   - `00000-user`: "[This conversation was compacted. Summary of prior context follows.]"
   - `00001-assistant`: the generated summary
5. **Audit trail**: the summary and token estimates are appended to `summary.md` in the session directory

The net effect is that the conversation shrinks dramatically while preserving the information Claude needs to continue working. A session can theoretically run indefinitely through repeated compactions.

Auto-Compaction

After every assistant response (when the agent loop completes), mini-claude checks total session size. If it exceeds the threshold, compaction fires automatically. This means long agentic runs self-manage their context without user intervention.

How It Differs from Claude Code

Aspect	mini-claude	Claude Code
Size	~750 lines, single bash file	~215 MB compiled Node.js application
Dependencies	`curl`, `jq`	Node.js runtime, bundled binaries (ripgrep, tree-sitter)
Tools	3 (`bash`, `read_file`, `write_file`)	15+ (including diff/patch editing, glob, grep, web fetch, memory/notebook, multi-file editing)
Permissions	Binary confirm/deny per tool call	Configurable allow-lists, regex patterns, auto-approval rules, per-project settings
Context management	File-size-based auto-compaction with API-generated summaries	Sophisticated token counting, intelligent compaction, codebase indexing

Aspect	mini-claude	Claude Code
Project awareness	None — Claude must explore via tools	Automatic codebase understanding, CLAUDE.md project files, git-aware context
Editing	Whole-file writes only	Surgical diff/patch editing, multi-file coordinated changes
Session storage	Filesystem directories with plain text files	Internal database, opaque format
Transparency	Every line is readable bash	Compiled, minified JavaScript bundle
Streaming	No — waits for complete response	Yes — tokens stream to terminal in real-time
MCP support	No	Yes — extensible via Model Context Protocol servers
System prompt	None (uses API defaults)	Extensive system prompt with tool documentation, coding guidelines, safety rules
Cost visibility	Estimates tokens from file sizes (~chars/4)	Precise token counting from API response metadata

What Claude Code Has That mini-claude Doesn't

- **Diff-based editing**: Claude Code can make surgical changes to specific lines in a file without rewriting the whole thing. Mini-claude must read, modify in memory, and write back the entire file.
- **Project context**: Claude Code automatically understands your project structure, reads `CLAUDE.md` files for project-specific instructions, and builds codebase indexes. Mini-claude starts with zero context — Claude has to explore using tools.
- **Streaming**: Claude Code streams tokens to the terminal as they're generated. Mini-claude waits for the complete API response before displaying anything.
- **Sophisticated permission management**: Claude Code lets you define fine-grained rules about what's allowed — specific commands, file path patterns, tools that can auto-approve. Mini-claude has a single binary switch.
- **MCP integration**: Claude Code can connect to external tool servers via the Model Context Protocol. Mini-claude's tool set is fixed.
- **Memory**: Claude Code has a persistent memory/notebook tool for storing context across sessions. Mini-claude's compaction summaries serve a similar purpose but less flexibly.

What mini-claude Has That Claude Code Doesn't

- **Total transparency**: you can read every line, understand every decision, modify any behaviour. There is no compiled code, no bundled runtime, no abstraction layers.
- **Inspectable sessions**: conversations are plain files you can browse, grep, edit, and version-control. You can see exactly what was sent to the API on every turn.
- **Zero install**: copy one file, make it executable, done. Works on any system with bash, curl, and jq.
- **Hackability**: want to add a new tool? It's 15 lines — add JSON to the `TOOLS` string, add a `case` branch to `execute_tool`. Want to change the compaction strategy? Edit `compact_conversation`. Want to add streaming? Replace the `curl` call.
- **Portability**: runs on any POSIX-like system. No Node.js, no npm, no native binaries.

Why It's Interesting

As a Teaching Tool

Mini-claude is perhaps the clearest possible demonstration of how agentic AI coding tools work. The entire system — authentication, conversation management, tool execution, the agent loop, context compaction — is visible in a single file. There's no framework, no abstraction, no indirection. The API call is a `curl` command. Tool execution is a `case` statement. The agent loop is a `while true`. Anyone who can read bash can understand exactly how an AI agent works.

As an Engineering Artefact

The filesystem-as-database approach is an interesting design choice. In most applications, using the filesystem for structured data storage would be considered naive. But for a conversational agent where messages are naturally sequential, append-mostly, and individually meaningful, it works remarkably well. The numbered-directory scheme gives you ordering, the file-per-field scheme gives you type separation, and standard Unix tools give you querying. The fact that sessions survive crashes, are human-editable, and are trivially debuggable are real practical benefits that come free from the storage model.

The build_payload function — which reconstructs the API request from dozens of small files via temporary per-message JSON fragments merged with `jq -s` — is an unusual pattern that avoids the $O(n^2)$ cost of growing a JSON string in a shell variable. It's arguably over-engineered for a bash script, but it demonstrates that the filesystem-first approach scales to long conversations without pathological performance.

As a Practical Tool

Despite its simplicity, mini-claude is genuinely useful for real work. Claude with `bash`, `read_file`, and `write_file` can accomplish the vast majority of coding tasks. It can navigate codebases, run tests, fix bugs, refactor code, manage infrastructure, write

documentation. The three-tool set isn't a limitation in practice — it's what Claude Code's more specialised tools decompose into anyway. `diff/patch` editing is just `read_file` + `write_file` with fewer tokens. `grep` is just `bash` running grep. The specialised tools in Claude Code are optimisations, not capabilities.

The session persistence and compaction mean you can use it for extended multi-session projects. Start working on something, close your terminal, come back days later and resume exactly where you left off. If the conversation gets too long, it automatically summarises and continues.

As a Baseline

Mini-claude establishes a useful lower bound: this is the minimum viable agentic coding tool. Everything above this — streaming, diff editing, project awareness, MCP, permission management — is an enhancement, not a requirement. If you're building or evaluating agentic tools, mini-claude shows you what the irreducible core looks like: an API call in a loop, three tools, and a place to store the conversation.

~750 lines. Two dependencies. One file. Everything else is Claude.