Member-only story

From Assistant to Autonomous Engineer: The 9-Month Technical Evolution of Claude Code

A technical deep-dive into Claude Code's transformation from terminal assistant to autonomous agent platform. Explore version-by-version improvements, plugin architecture, and what checkpoint systems mean for enterprise development workflows.

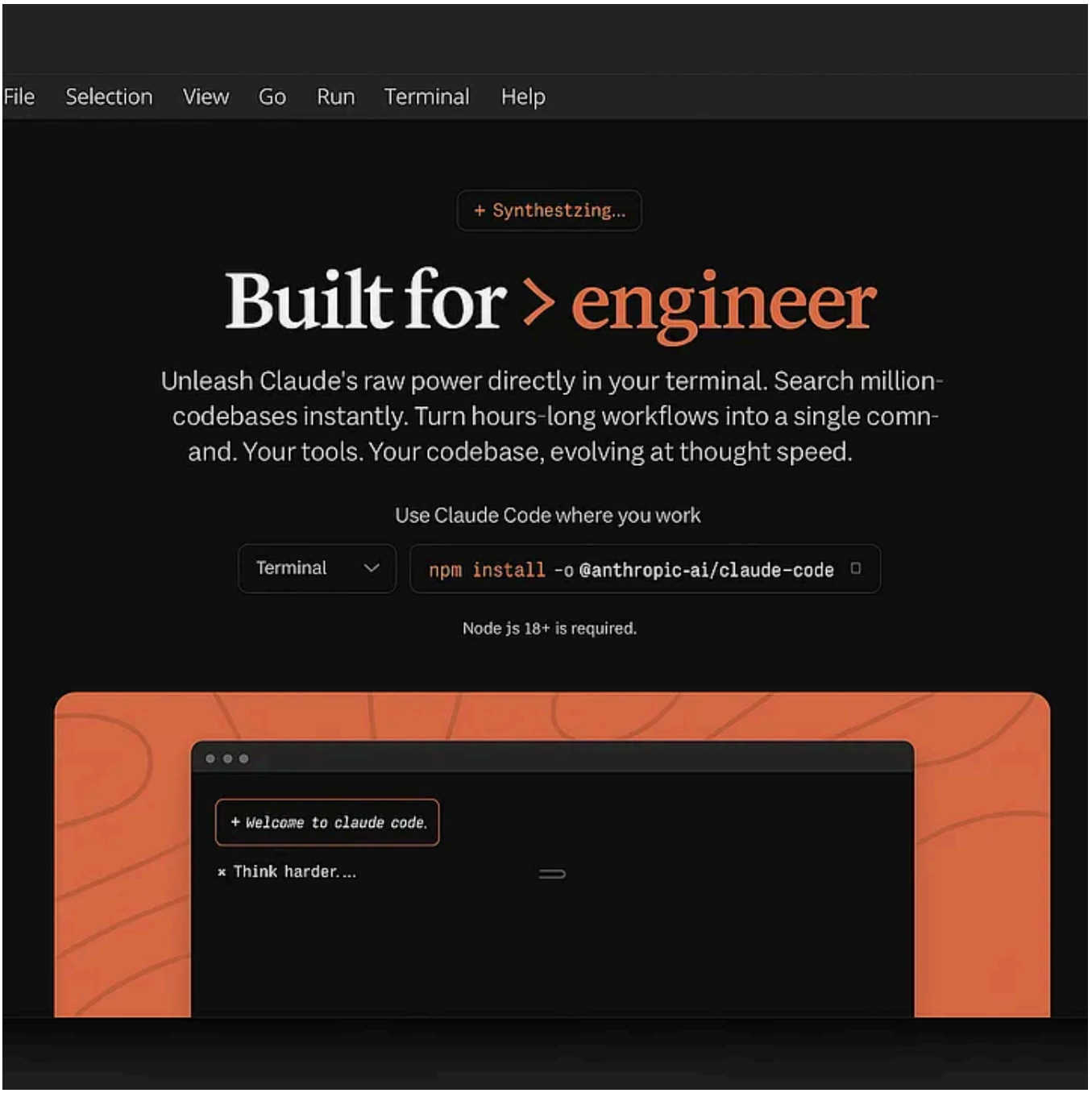
19 min read · 21 hours ago

Reza Rezvani Following

February, 2025. I'm staring at cascading failures across three microservices — authentication, session management, and our real-time notification system. Our AI coding assistant had suggested *"a simple refactor"* to reduce database round-trips. The code looked clean. Tests passed. The PR came back green.

Then production exploded.

The root cause wasn't in the suggested code itself. It was architectural blindness — our AI tool had zero understanding that our auth service maintains WebSocket connections with 2.3-second keep-alive intervals. Reducing the session check from synchronous to async created a race condition in the connection pool.



Claude Code Interface — Changelog & Evolution Roadmap (Feb. 2025 — Oct. 2025)

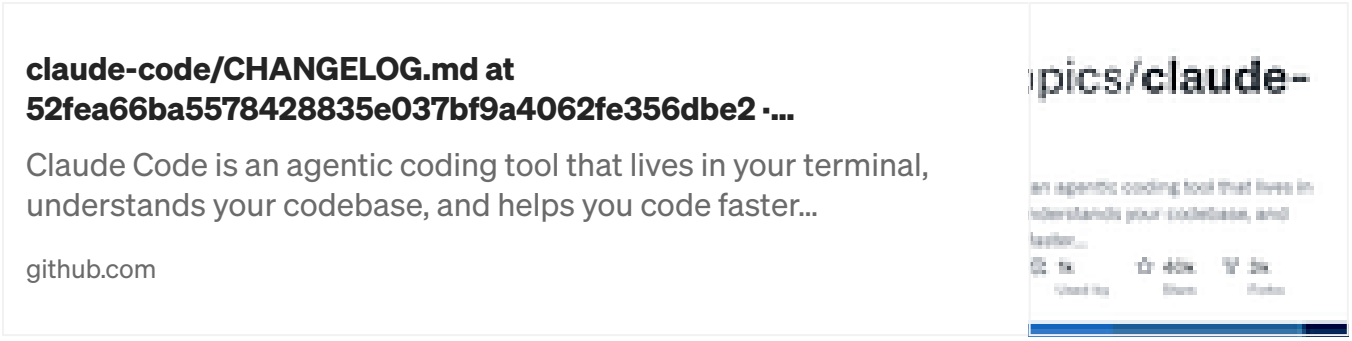
The AI couldn't see this because it lacked persistent context of our infrastructure patterns, no memory of previous architectural decisions, and no ability to reason across the boundaries of a single file edit.

That night, I realized the fundamental flaw in every AI coding assistant: they optimize for code generation speed while completely ignoring the context persistence problem.

Nine months and 27 versions later, Claude Code has systematically dismantled that limitation. Not through better autocomplete. Not through larger context windows.

Through rebuilding the entire architecture of how AI systems maintain state, reason across sessions, and integrate with development infrastructure.

This isn't a product review. It's a technical post-mortem of how Anthropic solved the problems that make every other AI coding tool feel like a chatbot with filesystem access.



. . .

Phase 1: The MCP Foundation (February-May 2025)

Before we can understand how Claude Code maintains context across weeks of development, we need to examine the protocol layer that makes it possible.

Version 0.2.x — The Protocol Nobody Noticed

February's preview release introduced something most developers dismissed as *"yet another integration format"* — the Model Context Protocol (MCP). I nearly made the same mistake. The documentation described MCP as *"an open standard for connecting LLMs to external data sources through JSON-RPC."* Standard enterprise architecture language. Nothing revolutionary.

Then I examined the actual implementation.

MCP isn't just an integration protocol — it's a complete state negotiation system. When Claude Code starts, it doesn't simply *"connect"* to data sources. It establishes stateful bidirectional sessions with multiple MCP servers, each maintaining independent conversation context through proper client-server architecture.

The Claude Code process acts as the MCP host, spawning client instances that communicate with servers via stdio, WebSocket, or HTTP Server-Sent Events transport layers.

Here's what this actually looks like in practice: I built an MCP server for our PostgreSQL schema exposing three primitives:

```
{
  "tools": ["query_schema", "validate_migration", "generate_types"],
  "resources": ["schema://tables", "schema://constraints", "schema://indexes"],
  "prompts": ["migration_template", "rollback_template"]
}
```

When Claude needs database context, it doesn't scan documentation files or guess table names. It invokes `query_schema` through the MCP client, receives structured JSON with table definitions, foreign key relationships, and index configurations, then uses that as typed context for generating migrations. The schema data never leaves my infrastructure—the MCP server runs locally, reading directly from `pg_catalog`.

This architecture solves the data sovereignty problem that makes most AI tools unsuitable for enterprise development. External APIs see prompts and responses. MCP servers see only structured queries and return only schema data, never code or business logic.

The `/permissions` command revealed even deeper sophistication. Instead of binary trust, Claude Code implements a capability matrix:

```
File Operations: Read [src/, tests/] | Write [src/] | Blocked [config/, .env]
Bash Execution: Allowed with confirmation | Network calls: Denied
MCP Servers: [postgres_schema: Active] [stripe_api: Inactive] [slack: Active]
```

Each capability can be granted, denied, or set to require confirmation at execution time. This isn't security theater — it's proper capability-based security implemented at the protocol level, enforced by the MCP host before any tool execution occurs.

Version 1.0 GA (May 22, 2025) stabilized this foundation with Claude Sonnet 4 and Opus 4, adding production-grade error handling and the @-mention system.

The @-mention implementation matters: when I type `@DatabaseSchema`, Claude Code queries the **postgres_schema MCP server** for the relevant resource, retrieves it as structured data with proper MIME types, and injects it into conversation context with namespace isolation preventing token collision with other resources.

Key architectural insight: MCP transforms AI from *"text predictor with tools"* to *"infrastructure-aware agent with structured data access."* This distinction becomes critical as complexity scales.

. . .

Phase 2: Stateful Composition (June-August 2025)

With the protocol foundation stable, Anthropic's focus shifted to a harder problem: how do you make agent workflows composable without requiring developers to write glue code for every integration?

The SDK That Exposed The Agent Runtime

Claude Code SDK (June 16, 2025) wasn't marketed as a major release. The announcement focused on *"easier integrations"* and *"custom agent development."* What it actually provided was direct access to Claude Code's internal agent harness — the same infrastructure powering the terminal application, now available as programmable TypeScript and Python APIs.

The Agent SDK exposes three critical primitives revealing how Claude Code actually works:

1. Session Management with Persistent State

Sessions persist to `~/.claude/sessions/{session-id}/` as SQLite databases with incremental append-only logs. When Claude Code restarts, it doesn't reconstruct context from chat history—it deserializes the complete session state, including MCP server connections and their internal state.

```
import { ClaudeSession } from '@anthropic-ai/claude-agent-sdk';
const session = await ClaudeSession.restore('project-refactor-2025-06');
// Session includes:
// - Complete conversation history (not just last N messages)
// - Active MCP server connections with their session state
// - Tool execution logs with inputs/outputs
// - File system snapshots as git-style tree objects
```

This is stateful agent execution, not conversation replay. The difference becomes obvious when working on multi-day projects — Claude maintains architectural context without asking you to explain decisions made three sessions ago.

2. Tool Coordination with Dependency Resolution

The SDK handles tool execution ordering automatically through dependency graph construction. When Claude decides to execute `read_file`, `run_tests`, and `write_file`, the SDK builds a dependency graph and executes in reverse dependency order with proper error propagation.

Real implementation: I built a deployment agent orchestrating five tools across three MCP servers — building Docker images, pushing to ECR, updating Kubernetes manifests, rolling out deployments, and posting to Slack. The SDK handles coordination. My code defines tools and dependencies. The agent runtime executes them correctly, handling failures with automatic rollback to previous stable state.

3. Context Streaming Without Full Rehydration

Traditional AI tools reserialize entire conversation context on every turn. With conversations containing code snippets and error messages, that's megabytes of JSON parsing per interaction. Claude Code's SDK implements incremental context streaming — only deltas get transmitted, reducing network overhead and enabling sub-second response times even with large contexts.

The 47-Hour Marathon That Almost Made Me Quit Claude Code — Until Everything Changed on September...

I had to share it with you ... Maybe you had to go through the same. I need to tell you about Tuesday, which almost broke...

alirezarezvani.medium.com

. . .

These SDK primitives enabled the next breakthrough — making agent behaviors composable through first-class extension points.

Version 1.0.45+ — Hooks As Executable Workflow Steps

July 2025 brought hooks, and I initially dismissed them as "just callbacks." Then I examined the implementation. Hooks aren't event listeners. They're full MCP tools invoked at specific lifecycle points with complete conversation context and filesystem state.

My production PostToolUse hook implementation runs ESLint and Jest automatically:

```
# .claude/hooks/post_tool_use.py
from claude_hooks import Hook, ToolContext

class LintAndTest(Hook):
    async def execute(self, context: ToolContext):
        if context.tool_name in ['write_file', 'edit_file']:
            # Run ESLint on changed files
            lint_results = await self.run_tool('bash', {
                'command': f'eslint {context.modified_files} --format json'
            })

            if lint_results['errors']:
                # Violations become structured context for next turn
                return {
                    'status': 'failed',
                    'violations': lint_results['errors'],
                    'suggestion': 'Fix linting errors before proceeding'
                }

            # Run affected tests
            test_results = await self.run_tool('bash', {
```

```
                'command': f'npm test -- --findRelatedTests {context.modified_f
            })

            return {
                'status': 'success',
                'tests_passed': test_results['numPassedTests'],
                'coverage_delta': test_results['coverageChange']
            }
```

When Claude writes a file, my hook executes automatically, runs ESLint and Jest, and feeds structured results back into Claude's reasoning context. If tests fail, Claude sees the failure output and fixes the issue before I review any diff. **My test coverage increased from 73% to 91% — not because I wrote more tests, but because the hook makes test failures impossible to ignore.**

The Stop hook implementation generates commit messages from conversation transcripts:

```
# .claude/hooks/stop.py
class AutoCommit(Hook):
    async def execute(self, context: SessionContext):
        # Generate commit message from conversation transcript
        summary = await self.claude_api_call(
            prompt=f"Summarize this session as a conventional commit:\n\n{conte
            model="claude-haiku-4.5"  # Fast, cheap for summaries
        )

        await self.run_tool('bash', {
            'command': f'git add -A && git commit -m "{summary}"'
        })
```

Every time my session ends, this hook generates a commit message by sending the conversation transcript through Claude's API with a summarization prompt. The commit messages are better than what I write manually — comprehensive, following conventional commit format, and explaining the architectural reasoning behind changes.

Custom slash commands in `.claude/commands/` solved template proliferation. These aren't text macros—they're markdown files with YAML frontmatter defining

parameters, validation schemas, and required MCP servers:

```
---
name: api:endpoint
description: Scaffold a new API endpoint with tests and documentation
parameters:
  - name: route
    type: string
    pattern: ^/api/v\d+/[a-z-]+$
    required: true
  - name: method
    type: enum
    values: [GET, POST, PUT, DELETE, PATCH]
    required: true
requires:
  mcp_servers: [postgres_schema, api_docs]
---
Create a new API endpoint at {{route}} handling {{method}} requests.
Requirements:
- TypeScript types from @postgres_schema
- OpenAPI documentation in @api_docs
- Integration tests with >80% coverage
- Input validation using Zod
- Error handling with proper HTTP status codes
```

When I type `/api:endpoint --route=/api/v2/users --method=POST`, Claude Code parses parameters, validates against schema, ensures required MCP servers are active, then injects the template with substituted variables. The MCP servers provide schema and documentation context automatically. I'm not writing boilerplate—I'm declaring intent, and the system handles implementation.

Version 1.0.100+ (August 2025) added background tasks — bash commands continuing execution while Claude works on other tasks. The implementation uses Unix process groups and SIGCONT/SIGSTOP for lifecycle management. I run `Ctrl+b` `npm run dev` to start the development server, and Claude continues coding while monitoring server logs through a separate file descriptor.

. . .

Phase 3: The Platform Transformation (September-October 2025)

The checkpoint system made long-running agent work possible. The plugin system made it shareable. Together, they transformed Claude Code from sophisticated tool into complete platform.

Version 2.0.0 — Checkpoints As Temporal Graph Navigation

September 29, 2025 brought the architectural insight that makes Claude Code fundamentally different: conversation state as a directed acyclic graph (DAG) with branching, not as linear history.

The checkpoint implementation stores:

```
Checkpoint Node {
  id: uuid,
  parent_id: uuid | null,
  timestamp: unix_epoch,
  conversation_state: {
    messages: Message[],
    mcp_sessions: Map<ServerId, SessionState>,
    tool_history: ToolExecution[]
  },
  filesystem_state: GitTreeObject,  // Delta-compressed, not full copy
  metadata: {
    model_used: string,
    tokens_consumed: number,
    active_skills: string[]
  }
}
```

When I execute `/rewind`, I'm not doing **Ctrl+Z** on file edits. I'm traversing the DAG to a previous node and optionally creating a branch. **This enables a development pattern impossible with traditional tools: experimental branching during active development.**

Real scenario from production work: I was debugging a memory leak in our Node.js service.

Claude proposed three approaches:

1. Add explicit garbage collection hints
2. Refactor to use WeakMap for caching

3. Switch from event emitters to async generators

I created a checkpoint, tried approach one. The leak persisted. Instead of manually undoing changes, I ran `/rewind --branch=weakmap-approach`, which restored the pre-change state and created a new branch in the conversation DAG. I tried approach two. Still leaking. Another rewind and branch to approach three—which fixed it.

Now I have three conversation branches in my checkpoint DAG, each containing complete implementation history, test results, and reasoning. I exported the successful branch as documentation, showing not just the solution but the full decision tree that led there.

Claude Sonnet 4.5 integration enabled this to work across multi-day sessions. Previous models degraded after extensive conversations as attention mechanisms struggled with long contexts. Sonnet 4.5 implements periodic context compression — it summarizes old conversation segments into dense "summary nodes" in the DAG while maintaining references to original nodes for expansion when needed.

I've had Claude work on a microservices migration for six calendar days across multiple conversation sessions. The checkpoint DAG maintains complete architectural context across all sessions. When Claude resumes, it doesn't start from scratch — it deserializes the most recent checkpoint, loads active MCP connections, and continues exactly where it stopped.

VS Code native extension (beta) implements a dedicated language server protocol (LSP) bridge. Claude's file edits appear as LSP textDocument/didChange notifications, triggering standard IDE features: syntax highlighting, linting, and IntelliSense work on AI-generated code in real-time. The diff viewer uses Monaco's native diff algorithm, not custom rendering. This is proper IDE integration, not a webview wrapper.

Version 2.0.12 — The Plugin System Architecture

October 9, 2025 introduced plugins — the feature most analysis misses. Claude Code Plugins aren't configuration files with scripts attached. They're complete, versioned extension packages implementing four distinct interfaces in Claude Code's execution model:

1. **Slash Commands** — Parameterized prompt templates with validation
2. **Subagents** — Isolated Claude instances with scoped permissions
3. **MCP Servers** — External integrations with full protocol support
4. **Hooks** — Event handlers with tool execution capabilities

The plugin manifest defines these and their dependencies:

```
{
  "name": "backend-workflow",
  "version": "2.1.0",
  "commands": [
    {
      "name": "migration:generate",
      "file": "commands/migration.md",
      "requires": ["mcp:postgres_schema"]
    }
  ],
  "agents": [
    {
      "name": "api-architect",
      "system_prompt_file": "agents/api_architect.md",
      "allowed_tools": ["read_file", "write_file", "bash:limited"],
      "mcp_servers": ["postgres_schema", "api_docs"]
    }
  ],
  "mcp_servers": [
    {
      "name": "postgres_schema",
      "type": "python",
      "module": "servers.postgres_mcp",
      "config": {
        "connection_string": "${DATABASE_URL}"
      }
    }
  ],
  "hooks": [
```

```
      {
        "type": "PostToolUse",
        "handler": "hooks/test_on_write.py",
        "tools": ["write_file", "edit_file"]
      }
    ]
  }
```

When someone installs this plugin, Claude Code validates compatibility, creates isolated environments, registers components, initializes MCP servers, and activates hooks in the execution pipeline.

Real implementation: Our team's onboarding used to require cloning five repositories, installing dependencies, configuring 23 environment variables, setting up a local database, configuring IDE settings, and installing custom tools.

Estimated time: 1.5 days for experienced developers, three-plus days for new hires.

With plugins:

```
claude plugin marketplace add company/backend-dev-stack
claude plugin install backend-complete
```

This single command installs MCP servers for our database schema and internal tools, custom slash commands for common workflows, a subagent for architecture guidance, hooks for automated testing and commits, and all configuration.

New developer onboarding time: 18 minutes. Not through automation scripts — because plugins package the entire development context as versioned, distributable infrastructure.

The plugin marketplace federation deserves attention. Any GitHub repository with `.claude-plugin/marketplace.json` becomes a discoverable marketplace. No centralized infrastructure. No approval process. No platform fees. Just a JSON file in a repository. Anthropic doesn't control plugin distribution—they created the protocol and let the ecosystem self-organize.

• • •

Version 2.0.17+ — Autonomous Context Discovery

The **Explore subagent** implements codebase navigation through semantic search instead of keyword matching. When Claude needs to understand our authentication flow, it spawns the Explore subagent, which embeds the query, performs vector similarity search over embedded code chunks, retrieves relevant files, and returns them as context. This happens transparently during reasoning.

The **interactive question tool** inverted a fundamental assumption. Instead of Claude making its best guess about ambiguous requirements and potentially getting them wrong, it now formulates structured clarification requests:

```
{
  "type": "multiple_choice",
  "question": "For the user authentication endpoint, which session strategy?",
  "options": [
    {
      "id": "jwt",
      "label": "JWT tokens with Redis cache",
      "implications": "Stateless, scales horizontally, requires Redis"
    },
    {
      "id": "session",
      "label": "Server-side sessions with PostgreSQL",
      "implications": "Stateful, simpler, single-point-of-failure"
    }
  ],
  "context": "Current codebase uses both patterns in different services"
}
```

I select an option, and the answer becomes structured context without breaking Claude's reasoning flow. This eliminates the *"almost right"* problem — where AI generates plausible but architecturally wrong solutions because it guessed at ambiguous requirements.

Claude Skills (Version 2.0.20+) represent persistent learned context. A skill is a directory structure containing instructions, executable scripts, and static reference data. When Claude Code detects a TypeScript project, it automatically loads the TypeScript skill.

> The skill's instructions become part of Claude's system prompt. Scripts become available tools. Resources become accessible through @-mentions.

Critical detail: Skills persist across sessions and projects. Once loaded, Claude remembers patterns and applies them to future work without reloading. This is transfer learning through structured context, not model fine-tuning.

Production example: I created a skill for our API design patterns. With this skill active, every API endpoint Claude generates follows our patterns automatically. No need to repeat requirements. No need to correct violations. The skill encodes organizational knowledge that persists across all Claude sessions.

. . .

The Uncomfortable Truth: Most Developers Are Using Claude Code Wrong

Understanding what Claude Code can do is pointless if you're using it the same way you use Copilot. Here's the pattern separating novices from experts.

Most developers interact with Claude Code the same way they use ChatGPT: type natural language requests in the terminal, wait for responses, review generated code, repeat. This is the lowest-leverage use case of the entire platform.

Here's the reality: if you're primarily using conversational prompts, you're using expensive autocomplete with filesystem access. You're missing the entire architectural value.

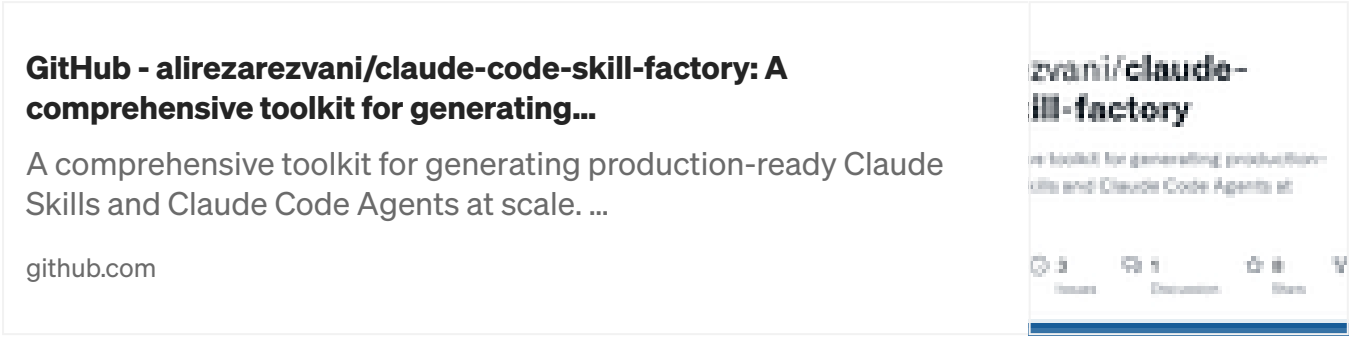
The developers extracting real leverage understand that Claude Code is **programmable infrastructure, not a conversational interface.** They're building on

four layers most users never touch:

Layer 1: MCP Servers as Data Adapters

Stop copy-pasting context. Write MCP servers that expose your infrastructure as structured resources — database schemas, API documentation, cloud infrastructure, internal tools. When these servers are active, Claude has real-time access to your actual infrastructure state. It doesn't guess table names — it queries your schema. It doesn't invent API endpoints — it reads your OpenAPI spec.

This is the difference between a coding assistant and an infrastructure-aware agent.



Layer 2: Skills as Organizational Memory

Your team has accumulated knowledge about architecture patterns, code style, testing strategies, and deployment procedures. That knowledge lives in Slack conversations, Notion documents, and senior developers' heads.

Skills encode that knowledge as executable context. Create skills for your specific technology stack, your design patterns, your quality standards, your operational procedures.

Once you have five to ten active skills, Claude Code becomes an extension of your team's collective technical memory.

Layer 3: Hooks as Workflow Automation

Stop manually running linters, tests, and formatters. Stop manually committing code. Stop manually updating documentation. Hooks automate every repetitive step in your development workflow.

My production hooks run *ESLint and Prettier* on every file write, execute affected tests automatically, update API documentation when routes change, generate

conventional commits from conversation transcripts, and post deployment summaries to Slack when sessions end.

I haven't manually run a linter in three months. My commit messages are better than before. My test coverage is higher. None of this required changing my development process — I just configured hooks once.

Layer 4: Plugins as Team Standards

The final leverage point: package your entire development context as distributable plugins. New team members install your plugin and get pre-configured MCP servers, team-specific f, automated workflows through hooks, and organizational knowledge through skills.

This is how you scale development practices without documentation debt or tribal knowledge.

The pattern that separates experts from everyone else: they spend more time configuring Claude Code's infrastructure than writing conversational prompts. They build MCP servers, write skills, configure hooks, and package plugins. Then Claude Code becomes genuinely autonomous, working within the constraints and patterns they've defined.

If you're not using at least three of these layers, you're using expensive autocomplete.

Claude Code 2.0.27:

Claude Code 2.0.27: Why and How this Update Actually Matters (And Why It Doesn't Replace Your Terminal) Web-based...

alirezarezvani.medium.com

. . .

The Architecture That Actually Matters

Let's examine the technical decisions enabling Claude Code's capabilities — and why they represent a fundamentally different approach to AI tooling.

From Stateless Requests to Stateful Sessions

Traditional AI coding tools treat each interaction as independent. Conversation context exists only as JSON payload sent with each API request. When you restart your IDE, context disappears. When you switch projects, context resets. When context windows fill, old context gets truncated.

Claude Code implements persistent session state as SQLite databases with append-only transaction logs. Sessions include complete conversation history, active MCP server connections with their internal state, tool execution history, filesystem snapshots as git-style tree objects with delta compression, active skills and their loaded resources, and checkpoint DAG with branching support.

This architectural decision enables agent work spanning days or weeks without context degradation. When Claude Code restarts, it doesn't reconstruct context from chat history — it deserializes complete session state, including MCP server connections and their session data.

From Request-Response to Continuous Context Streaming

Most AI tools reserialize entire conversation context on every turn. Claude Code implements incremental context streaming. The SDK sends only deltas — new messages, MCP resource updates, tool results, and modified files. The Claude API doesn't receive the entire conversation on each turn. It receives structured delta updates that get merged into running context state.

This enables sub-second response times even with large conversations, and it scales to genuinely long-running agent work.

From Prompt Injection Defense to Capability-Based Security

The AI security discourse obsesses over prompt injection — tricking models into executing unintended actions through carefully crafted inputs. This threat model assumes agents have undifferentiated access to tools.

Claude Code implements capability-based security at the protocol level. The MCP host maintains a permission matrix for every tool and MCP server. Even if a prompt injection succeeded in getting Claude to attempt unauthorized actions, the MCP host would deny tool execution before it reaches the filesystem or network.

This is defense in depth: prompt-level protection plus protocol-level enforcement plus OS-level sandboxing.

From Vendor Lock-in to Protocol Federation

Most AI coding tools are platforms: centralized marketplaces, proprietary plugins, vendor-controlled distribution. Claude Code implements protocol federation.

The MCP protocol is open and standardized. Anyone can write an MCP server. Any IDE can become an MCP host. Plugin marketplaces are just JSON files in repositories — no centralized infrastructure required.

This architectural choice makes Claude Code valuable even if Anthropic stops developing it. The MCP protocol, plugin format, and session state structure are documented and open. The ecosystem can continue developing without Anthropic's permission or participation.

This is how platforms should scale: through protocol standardization, not platform lock-in.

. . .

Your Implementation Roadmap

Understanding the architecture is pointless without a concrete plan for implementation. Here's the technical path from novice to expert use.

Week 1: Understand the Infrastructure Layer

Don't start by asking Claude to write code. Start by understanding how Claude Code actually works:

1. **Examine session persistence**: Navigate to `~/.claude/sessions/` and open a session database with SQLite. Study the schema—messages, tool_executions, mcp_connections, checkpoints tables.
2. **Study MCP protocol**: Read the specification at docs.claude.com/mcp. Understand JSON-RPC message format, client-server architecture, and the three primitives.
3. **Test checkpoint DAG**: Create a checkpoint, make changes, rewind. Make different changes, create a branch. Export both branches. Visualize how the DAG stores conversation alternatives.
4. **Review plugin manifest schema**: Look at existing plugins' `plugin.json` files. Understand how commands, agents, MCP servers, and hooks get declared.

Week 2: Build Your First MCP Server

Choose a simple data source and implement a basic MCP server. Start with your internal API schema, database schema, or documentation system. Connect to Claude Code via `.mcp.json` configuration.

Test it by using @-mentions to verify Claude can retrieve your resources and invoke your tools.

Week 3: Implement Automated Workflows

Create hooks that automate your development workflow. Start with a PostToolUse hook running tests on file modifications. Add a Stop hook for automated git commits.

The key is making the workflow invisible — hooks should execute automatically without manual invocation.

Week 4: Package as Team Plugin

Create a complete plugin for your team combining custom slash commands, specialized subagents, MCP servers for internal tools, and hooks automating common workflows.

Define in `plugin.json`, publish to a GitHub repository, distribute to team via the plugin marketplace.

Verify new team members can install your plugin and get a complete, configured development environment in minutes.

. . .

The Actual Conclusion

The race condition that cost me my Valentine's Day taught me something nine months of Claude Code evolution has now proven at scale: **The problem with AI coding tools isn't the models — it's the architecture.**

Better autocomplete doesn't solve context persistence. Larger context windows don't solve state management. Faster inference doesn't solve the trust problem. You can't prompt-engineer your way around architectural limitations.

Claude Code's technical evolution demonstrates a different approach: build sophisticated infrastructure that makes existing models genuinely useful. The MCP protocol solved the integration problem. The checkpoint DAG solved the exploration problem. The plugin system solved the distribution problem. Skills and hooks solved the customization problem.

None of these are model improvements. They're all architectural decisions.

The uncomfortable reality is that most developers will never exploit these capabilities. They'll use Claude Code the same way they use Copilot — as a conversational interface that writes code.

They'll miss the MCP servers providing real-time infrastructure context. They'll skip the skills encoding organizational knowledge. They'll ignore the hooks automating workflows. They'll never package their setup as distributable plugins.

And they'll wonder why their *"AI pair programmer"* still feels like a chatbot that occasionally guesses right.

The actual value of Claude Code isn't in the conversational interface. It's in the programmable infrastructure beneath it.

The checkpoint system gets headlines because it's easy to demo. The plugin marketplace gets attention because it looks like an app store. But the real technical achievement — the one that will define the next decade of autonomous development — is the MCP protocol, the session persistence architecture, and the capability-based security model.

These aren't features. They're foundational infrastructure for a new category of development tools: AI agents that integrate with your actual infrastructure, maintain context across weeks of development, and operate within well-defined security boundaries.

The question isn't whether you should learn Claude Code. It's whether you understand the underlying architecture well enough to build on it.

Start with the MCP protocol. Build a server that exposes your infrastructure as structured resources. Write a skill that encodes your team's patterns. Create hooks that automate your workflow. Package it as a plugin and distribute to your team.

Then you'll understand what *"autonomous development"* actually means. Not AI that writes faster. AI that understands deeper.

The conversation model is the demo. The protocol layer is the product.

. . .

I'm tracking novel MCP server implementations and plugin architectures. If you've built something interesting — particularly MCP servers for uncommon data sources or plugins that solve team-specific problems — I want to analyze the implementation and share the patterns. Message me with your architecture details, or document your approach in the comments below.

. . .

About This Analysis

This technical deep dive is part of an ongoing series examining the architecture of autonomous AI development tools. For more analysis of AI coding assistants, agent architectures, and development infrastructure, follow me on Medium.

About the Author

Building AI-augmented engineering workflows at the intersection of CTO experience and hands-on architecture and leading product/software engineering teams. Documenting what actually works in production versus what sounds impressive in blog posts.

Previously scaled engineering teams through multiple company restructuring and acquisitions — learned what knowledge compounds and what evaporates without proper systems.

Connect: LinkedIn | **Read more:** Medium Reza Rezvani | **Explore:** GitHub

Claude Code | Software Development | Developer Tools | Model Context Protocol | Ai Autonomous Agents

Following

Written by Reza Rezvani

911 followers · 71 following

As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

No responses yet

Bgerby

What are your thoughts?

More from Reza Rezvani

Reza Rezvani

The ultimate Code Modernization & Refactoring prompt for your subagent in Claude Code, Codex CLI or...

Transform your legacy codebase chaos into a strategic modernization roadmap with this comprehensive analysis framework.

Oct 4 130 2

In nginity by Reza Rezvani

The Flutter Architecture That Saved Our Team 6 Months of Rework

Sep 14 393 14

In nginity by Reza Rezvani

I Let Claude Sonnet 4.5

IMAGINE this: It's 6 a.m., the kind of quiet dawn where the world's still wrapped in that soft, hazy light filtering through your blinds...

Sep 29 101 1

Reza Rezvani

Mastering Claude Code: A 7-Step Guide to Building AI-Powered Projects with Context Engineering

From Chaos to Code: How I Reduced Development Time by 70% Using Claude Code's Hidden Power

Sep 9 175 4

See all from Reza Rezvani

Recommended from Medium

Agen.cy

20+ Genius Ways Power Users Are Using Claude Code Right Now

Here are 10+ ways power users are using Claude Code🧵

4d ago 14

In nginity by Reza Rezvani

Claude AI and Claude Code Skills: Teaching AI to Think Like Your Best Engineer

Oct 19 46 1

In Realworld AI Use Cases by Chris Dunlop

The complete guide to Claude Code’s newest feature “skills”

Claude Code released a new feature called Skills and spent hours testing them so you don’t have to. Here’s why they are helpful

Oct 21 63 4

Joe Njenga

I Tested Claude Code + Claude 4.5 (3 Tests for $1—And It Made Me Question How I Code)

I took Claude Code + Claude Sonnet 4.5 back-to-back through real coding tasks, for a few dimes ($0.35, $0.48, $0.17), and I discovered the

3d ago 18

In Dare To Be Better by Max Petrusenko

Claude Skills: The $3 Automation Secret That's Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 376 5

In Coding Nexus by Code Coup

OpenSpec: A Spec-Driven Workflow for AI Coding Assistants (No API Keys Needed)

I've been using AI coding tools for a while—Claude, Cursor, even Copilot. They're fast, sometimes magical... but they also love to...

Oct 18 65

See more recommendations