Supercharging C++ Development with Claude Code

How AI-Powered Workflow Automation Transformed My Development Process

Martin Kjeldsen

January 12, 2026

Outline

What is Claude Code?

Claude Code: AI-Powered Development Assistant

Claude Code (CC) is Anthropic's official CLI tool

Key Features:

- ▶ Interactive terminal interface
- ◊ Full codebase awareness
- ♣ Autonomous task execution
- ◯ Intelligent code search
- ▷ Multi-file editing

Not Just a Chatbot
Claude Code can:

- Read your entire codebase
- Execute shell commands
- Edit multiple files
- Run git operations
- Debug and test code

Why Claude Code?

Traditional Development:

- Manual file navigation
- Repetitive boilerplate
- Context switching
- Documentation hunting
- Tool configuration hell

With Claude Code:

- ✓ Natural language tasks
- ✓ Automated scaffolding
- ✓ Maintains context
- ✓ Built-in knowledge
- ✓ One-command setup

From hours to minutes.

The Workflow Revolution

Before Claude Code: The Old Workflow

Starting a new C++ project meant:

1. Create directory structure manually
2. Copy-paste CMakeLists.txt from old project
3. Setup .gitignore, .clang-format, .clang-tidy
4. Configure LSP (clangd) settings
5. Setup debugger configurations
6. Initialize git repository
7. Write README boilerplate

Time Investment
30-60 minutes of tedious setup before writing any actual code

After Claude Code: One Command

Now starting a new project:

```
$ ~/dotfiles/scripts/bootstrap-cpp-project.sh my-renderer
[INFO] Creating C++ project: my-renderer
[INFO] Location: /home/user/dev/my-renderer
[INFO] Created directory structure
[INFO]   OK CLAUDE.md
[INFO]   OK .clangd
[INFO]   OK CMakeLists.txt
[INFO]   OK .nvim.lua (debug configs)
[INFO]   OK src/main.cpp
[INFO]   OK Git repository initialized

OK Project created successfully!
```

Time Investment
5 seconds and you're ready to code

CLAUDE.md: The Secret Weapon

Every project gets a CLAUDE.md file:

What it contains:

- Project overview
- Architecture notes
- Build commands
- Testing instructions
- Common tasks
- Recent changes

Why it matters:

- Claude reads it automatically
- Maintains project context
- Onboards new developers
- Living documentation
- Prevents knowledge loss

Key Insight
CLAUDE.md turns every conversation into a context-aware collaboration

Example: CLAUDE.md in Action

Without CLAUDE.md:

```
You: "Add a new renderer class"
Claude: "Which file should I add it to? What's the
         architecture? What rendering API are you using?"
```

With CLAUDE.md:

```
You: "Add a new renderer class"
Claude: *reads CLAUDE.md* "I see you're using Skia for
        rendering. I'll add it to src/renderer.cpp
        following your existing factory pattern."
```

Zero back-and-forth. Pure productivity.

It's Not Just C++

The same workflow works for TypeScript!

C++ Bootstrap:

- clang-format
- clang-tidy
- CMakeLists.txt
- .clangd LSP config
- codelldb debugger
- Git hooks

TypeScript Bootstrap:

- Prettier
- ESLint
- tsconfig.json
- ts_ls LSP config
- Node.js debugger
- Git hooks

Multi-Language Support
Same philosophy, different tools. One workflow for all your projects.

TypeScript Workflow in Action

Creating a React + TypeScript app:

```
$ ~/dotfiles/scripts/bootstrap-ts-project.sh \
    my-app --framework=react

[INFO] Creating TypeScript project: my-app
[INFO] Framework: react
[INFO]   OK tsconfig.json
[INFO]   OK .eslintrc.json
[INFO]   OK .prettierrc
[INFO]   OK package.json
[INFO]   OK CLAUDE.md
[INFO]   OK Git hooks installed

$ cd my-app && npm install && npm run dev
  VITE ready in 191 ms
```

Real Example: We Just Did This

10 minutes ago, we built a complete TypeScript workflow from scratch:

What we created:

- TypeScript stow package
- Bootstrap script (4 frameworks)
- Neovim LSP + DAP config
- Git hooks (pre-commit/push)
- Demo React app
- Full documentation

Time breakdown:

- tsconfig/ESLint/Prettier/Vitest: 2 min
- Neovim TypeScript support: 1 min
- Bootstrap script (4 frameworks): 3 min
- Git hooks (4 quality gates): 2 min
- Testing demo project: 2 min

Total: 10 Minutes
From "I don't have TypeScript support" to "Working React app with full quality gates"

TypeScript Framework Support

One script, four frameworks:

–framework=node Simple Node.js CLI apps with tsx
–framework=express REST APIs with Express + TypeScript
–framework=react React apps with Vite + HMR
–framework=next Next.js 14+ with App Router

All include:

- Strict TypeScript config (no implicit any)
- ESLint + Prettier + Vitest pre-configured
- Git hooks (4 quality gates: format, types, lint, tests)
- Neovim LSP + DAP debugging ready
- CLAUDE.md with framework-specific guidance

Automated Code Quality

Git Hooks: Quality Gates on Autopilot

Automatic enforcement for all languages:

C++ Projects:

- Pre-commit: clang-format
- Pre-push: clang-tidy
- Fix: `git cf`
- ASCII art on violations

TypeScript Projects:

- Pre-commit: Prettier
- Pre-push: tsc + ESLint + Vitest
- Fix: `npm run format`
- Type errors & test failures blocked

How It Works:

- Bootstrap script installs hooks
- Runs on every commit/push
- Fast (staged files only)
- Catches issues before review
- Bypass: `--no-verify`

Philosophy
Make the right thing automatic

Pre-commit Hook in Action

Formatting violation detected:

```
Checking C++ code formatting...

src/renderer.cpp needs formatting:

  *** CODE FORMATTING VIOLATIONS DETECTED! ***

How to fix:
  git cf                    # Format staged files
  git commit                # Try again

To bypass (not recommended):
  git commit --no-verify
```

No more "I'll fix formatting later" — it's enforced automatically.

Cross-Platform Dotfiles

One Dotfiles Repo, Four Platforms

Unified configuration across:

★ **Arch Linux** Full Hyprland desktop + all dev tools
★ **WSL** CLI tools only (no GUI)
◯ **macOS** Homebrew + Alacritty terminal
☐ **Windows** Scoop + Windows Terminal

Shared configs:

- Git (hooks, aliases)
- Neovim (full IDE)
- Clang tools
- Starship prompt

Platform-specific:

- Package managers
- Bash configs
- Terminal emulators
- Desktop environments

Platform Installation

Each platform has a one-command install:

```
# Arch Linux
./install_arch.sh

# WSL (Ubuntu/Debian)
./install_wsl.sh

# macOS
./install_darwin.sh
```

```
# Windows (PowerShell)
.\install_windows.ps1
```

Built with Claude Code

All four install scripts were created collaboratively with CC in one session

Unreal Engine Development

Neovim + Unreal Engine = Possible

Configured for UE C++ coding standards:

What works:

- ✓ LSP (clangd)
- ✓ Debugging (DAP)
- ✓ Formatting (Allman braces)
- ✓ Static analysis
- ✓ Git hooks

UE Standards:

- PascalCase naming
- Type prefixes (U/A/F/E/T)
- Boolean prefix (b)
- Tab indentation
- Always-braces policy

Hybrid Workflow
Neovim for C++ code, Unreal Editor for Blueprints/Assets (7/10 feasibility)

UE Clang-Tidy Configuration

Enforces Unreal Engine coding standards:

```
# .clang-tidy configured for UE

CheckOptions:
  # PascalCase for everything (not snake_case)
  - key: readability-identifier-naming.FunctionCase
    value: CamelCase
  - key: readability-identifier-naming.VariableCase
    value: CamelCase

  # No private member suffix (UE doesn't use trailing _)
  - key: readability-identifier-naming.PrivateMemberSuffix
    value: ''

  # Always-braces policy
  - key: readability-braces-around-statements.ShortStatementLines
    value: 0
```

Pre-push hook catches naming violations before code review.

Real-World Impact

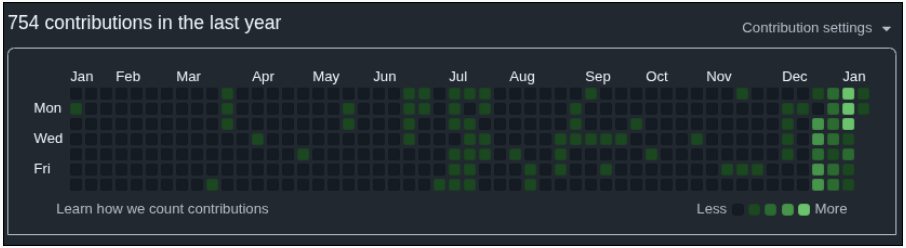
Productivity Gains: By The Numbers

Task	Before CC	With CC
New C++ project setup	30-60 min	5 sec
New TS/React project	20-40 min	5 sec
Configure LSP/DAP	20-30 min	0 min (automatic)
Setup git hooks	15-20 min	0 min (automatic)
TSConfig + ESLint + Prettier + Vitest	15-25 min	0 min (automatic)
Cross-platform config	Hours/days	1 session
Format code	Manual	Automatic (pre-commit)
Type/lint checking	Manual	Automatic (pre-push)
Find project docs	N/A	Instant (CLAUDE.md)

Time Saved

2+ hours per project in setup and configuration alone

GitHub Activity: The Proof



754 Contributions in the Last Year

The activity surge shows consistent productivity with Claude Code — more commits, better workflow, cleaner code.

Workflow Improvements

What changed with Claude Code:

1. **Context Switching** — Eliminated via CLAUDE.md
2. **Boilerplate Code** — Automated via templates
3. **Configuration Hell** — One-command install scripts
4. **Code Quality** — Enforced via git hooks
5. **Documentation Rot** — Prevented via living CLAUDE.md
6. **Platform Fragmentation** — Unified dotfiles

Focus on solving problems, not fighting tools.

Best Practices

Best Practices for Using Claude Code

1. **Maintain CLAUDE.md** — Update it as the project evolves
2. **Generate compile_commands.json** — Required for LSP
3. **Use project-local configs** — .clangd, .nvim.lua, etc.
4. **Trust the hooks** — Don't bypass pre-commit/pre-push
5. **Template everything** — Bootstrap scripts for consistency
6. **Document conventions** — Put them in CLAUDE.md
7. **Version your dotfiles** — Git repo with install scripts

Golden Rule
Invest time in automation once, reap benefits forever

What NOT to Use Claude Code For

CC is powerful, but not magic:

- × **Complex debugging** — Use a proper debugger (gdb/lldb)
- × **Graphical design** — Use specialized tools (Unreal Editor, Blender, etc.)
- × **Performance profiling** — Use profilers (perf, Valgrind, etc.)
- × **Real-time interaction** — CC has latency, not instant
- × **Binary/compiled analysis** — CC works with source code

Use Case
Claude Code excels at **automation, scaffolding, and refactoring** — not runtime operations

Live Demo

Live Demo: The Four Quality Gates

Let's write some bad code and watch the gates catch it!

What we'll break:

- Double quotes (Prettier)
- Missing spaces (Prettier)
- Implicit `any` types (TypeScript)
- Missing return types (ESLint)
- Failing tests (Vitest)

What will catch it:

- Gate 1: Prettier
- Gate 2: TypeScript
- Gate 3: ESLint
- Gate 4: Vitest

The Goal
Show that you literally cannot push broken code

Gate 1: Prettier Catches Formatting

Writing code with bad formatting:

```
// Bad formatting - inconsistent quotes and spacing
import React, { useState } from "react";

const Counter = () => {
  const [count,setCount]=useState(0);  // No spaces!
  ...
};
```

Run formatter:

```
$ npm run format
src/Counter.tsx 17ms
```

Result:

```
import React, { useState } from 'react';  // Single quotes!

const Counter = () => {
  const [count, setCount] = useState(0);  // Proper spacing!
  ...
};
```

Gate 2: TypeScript Catches Type Errors

Type checker finds implicit any:

```
$ npm run type-check

src/Counter.tsx(11,24): error TS7006:
  Parameter 'e' implicitly has an 'any' type.

const handleClick = (e) => {  // <-- No type!
  setCount(count + 1);
};
```

Fix by adding explicit types:

```
const handleClick = (): void => {  // <-- Explicit return type
  setCount(count + 1);
};
```

Strict mode prevents implicit any — no shortcuts allowed!

Gate 3: ESLint Enforces Best Practices

ESLint catches missing return types:

```
$ npm run lint

src/Counter.tsx
  8:20  warning  Missing return type on function
        @typescript-eslint/explicit-function-return-type

const Counter = () => {  // <-- No return type!
  ...
};
```

Fix by adding JSX.Element return type:

```
const Counter = (): JSX.Element => {  // <-- Explicit!
  ...
};
```

Three gates down, one more to go!

Gate 4: Vitest Catches Logic Bugs

Tests verify runtime behavior:

```
$ npm run test:run

src/Counter.test.tsx
  OK renders with initial count of 0
  OK displays the increment button
  OK increments count when button is clicked
  OK increments multiple times correctly

Test Files  1 passed (1)
     Tests  4 passed (4)
  Duration  1.25s
```

What Vitest catches that others can't:

- Incorrect logic (button doesn't increment)
- Wrong initial values (starts at 10 instead of 0)
- Runtime errors (undefined function calls)
- Component behavior bugs

Live Demo: Bootstrap a Project

Let's create a new C++ project from scratch:

1. Run bootstrap script
2. Show generated files (CLAUDE.md, CMakeLists.txt, .nvim.lua)
3. Build the project
4. Open in Neovim
5. Show LSP features (go-to-definition, autocomplete)
6. Show debug configurations (F5 to debug)
7. Make a change and commit (pre-commit hook triggers)

From zero to fully-configured IDE in under 60 seconds.

Live Demo: Claude Code in Action

Typical Claude Code session:

1. Ask CC to add a new feature
2. CC reads CLAUDE.md for context
3. CC explores codebase (grep, read files)
4. CC writes code across multiple files
5. CC runs tests to verify
6. CC commits with descriptive message

Key Observation
You describe *what* you want, Claude Code figures out *how*

Conclusion

Key Takeaways

1. **CLAUDE.md is essential** — Project context changes everything
2. **Automate setup** — Bootstrap scripts save hours
3. **Git hooks enforce quality** — Pre-commit + pre-push = clean code
4. **Cross-platform dotfiles** — Write once, run everywhere
5. **Invest in tooling** — Good tools compound over time

Claude Code isn't just a tool — it's a workflow multiplier.

Resources

Learn More:

- • GitHub: `https://github.com/anthropics/claude-code`
- ≡ Documentation: `https://claude.com/claude-code`
- ♢ My Dotfiles: `https://github.com/zrrbite/dotfiles`

Questions?

This presentation was created *with* Claude Code, naturally.