LLM-Augmented Editing for the Workflow Construct

Ronald G. Thomas

2026-04-28

Figure 1: A small wooden writing desk with a closed laptop, an open notebook lying flat with a fountain pen resting across it mid-sentence, and a small handwritten notecard propped upright nearby, suggesting a running set of working notes alongside the machine.

An LLM-augmented editing workflow is one in which the human retains custody of the writing surface and the versioned source, and the assistant proposes diffs that the human reviews, applies, or rejects. The opposite arrangement (the assistant owns the buffer and the human supervises) produces different ergonomics and a different distribution of failure modes.

Introduction

The arrival of capable code-aware language models between 2022 and 2026 has produced a spectrum of integration patterns, each making a slightly different bet about who owns the buffer and how diffs are reviewed. At one end of the spectrum, the human types into a familiar editor and occasionally invokes a short consultation with the model, either through a CLI or through an in-buffer plugin. At the other end, the model generates a draft into a chat surface and the human pastes the result

into their editor by hand. In the middle sit graphical 'AI-first' editors that have forked an existing IDE (most notably Cursor, a fork of VS Code) and rebuilt the editing surface around the model's output stream.

Each integration is internally consistent. The choice between them is largely a choice about which artifacts the human treats as authoritative. A workflow that treats the versioned source under git as authoritative, with the assistant proposing diffs that the human reviews before committing, places the assistant in a deferential role. This lets every existing piece of construct machinery (modal editor, secret-scanning git wizard, container-bound R session, reproducible compendium) operate unchanged. A workflow that treats the assistant's chat surface as authoritative, with the human pasting outputs into a separate editor, fragments the construct. The chat transcript becomes a load-bearing artifact that lives outside git, outside the dotfiles repository, and outside the secret-scanner's reach.

We document here the first integration, Claude Code at the shell, as the recommended primary for the Workflow Construct. The shell-CLI integration is the cheapest to adopt (one binary, one configuration file per project), the most consistent with the construct's existing tools (it runs in a terminal alongside everything else), and the most defensible under git review (the diffs it proposes pass through `zzgit` and the staging-time secret scanner like any other diff). In-editor plugins (Neovim with `codecompanion.nvim` or `avante.nvim`) are documented as conditional adjacents for users whose buffer-level workflow benefits from inline diff display. Graphical alternatives (Cursor, Zed) and the historical browser-based prompting pattern (post 36, retained as a case study of one specific older integration) are noted but are not the recommended primary.

More formally, we document here the LLM-augmented editing extension family of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist. This post is one of the two posts in the construct's Tier D (Knowledge / LLM), alongside a planned companion post (not yet written) on Zotero, Obsidian, and Pandoc. The two posts together address the construct's research-side companion layers: the companion post covers the inputs (references, notes, bibliographic data); this post covers the editing interaction with those inputs and with the working source tree.

Motivations

The pain points that motivated documenting the shell-CLI integration as the construct's primary are specific:

- **Loss of context across editor and chat surfaces.** A workflow that pastes the assistant's output between a browser tab and an IDE loses fidelity at every paste: whitespace, indentation, partial selections, and the surrounding code structure all degrade. The assistant also loses access to the rest of the project (the other files, the git history, the test suite) because it sees only what the human has copied into the chat surface.
- **Authoritative artifacts split across systems.** A graphical AI editor that holds chat-style discussion inside the editor produces transcripts that are not in git. The transcripts contain the rationale for changes the human committed; their absence in git means the history loses the argument that produced each commit.
- **Vendor lock-in to a specific editor.** A workflow built around Cursor requires Cursor; a workflow built around the construct's existing modal editor (post 26) plus a shell CLI is portable to any editor and any remote machine that supports a terminal.

- **Compatibility with the construct's review and audit layers.** The `zzgit` secret-scanning workflow (post 49) inspects the staged diff for credentials. A diff produced by an in-buffer assistant is no different from a hand-typed diff at staging time; both pass through the same scanner. A diff applied by a graphical editor that bypasses the construct's git wizard produces an unscanned commit.
- **Adoption cost amortization.** A workflow that requires learning a new editor, a new shell, and a new set of ergonomic conventions front-loads the adoption cost. The shell-CLI integration adds a single new binary (`claude`) to a workflow whose other layers are unchanged; the marginal learning is small.

Objectives

This post sets out to deliver:

1. Installation and configuration of Claude Code at the shell, including the per-project `CLAUDE.md` convention that makes the assistant aware of the project's conventions, file layout, and persona.
2. A small set of complementary in-editor integrations (Neovim with `codecompanion.nvim` for users who prefer inline diffs; Cursor and Zed noted briefly as graphical alternatives).
3. A daily-workflow command catalog mapping common editing patterns (refactor, document, test, review, explain) to the corresponding Claude Code invocation.
4. A failure-mode catalog covering hallucinated APIs, stale context, secret exposure, cost runaway, and the model's tendency to over-engineer beyond the request.

One should be able to install and configure the shell-CLI integration in approximately 30 minutes, with the per-project `CLAUDE.md` files added incrementally as one works through projects.

Figure 2: Two hands, one human and one a simple articulated wooden artist's mannequin hand, both resting near the same pen on a desk, neither quite touching it.

What Is LLM-Augmented Editing?

LLM-augmented editing is the broad name for any editing workflow in which a language model contributes to the production of source text alongside a human author. The contribution can take five distinct forms, distinguished by where the model lives relative to the human's authoritative buffer:

1. **Shell-CLI assistant.** The model is invoked from a terminal command (`claude` for Anthropic's Claude Code; `aider` for an open-source alternative). It can read any file in the project, propose diffs, run tests, and commit, all under the human's per-step approval. The buffer remains the human's editor; the assistant consults via the shell.
2. **In-buffer plugin.** The model is invoked from within the editor itself (`codecompanion.nvim` or `avante.nvim` for Neovim; Copilot inline suggestions for VS Code). It produces diffs that appear directly in the buffer for accept-or-reject. The integration is tighter than shell-CLI but is editor-specific.
3. **Graphical AI editor.** A graphical fork of an existing IDE (Cursor for VS Code; Zed with its built- in AI integration). The editor itself is rebuilt around the model's output, with a chat panel as a first-class UI element. The integration is tighter still but the editor is no longer the same editor the human used before.
4. **Browser-based prompting.** The model is consulted through its provider's web UI (claude.ai, chatgpt.com). The human copies code from their editor into the chat, asks for a transformation, and pastes the result back. Post 36 in this repository is a worked case study of this pattern with a 2023-era ChatGPT for a Shiny app.

5. **Direct API integration.** The application code itself calls the model's API at runtime. This is not 'editing' per se but is sometimes confused with it; see the Anthropic SDK and OpenAI SDK documentation for the actual application integration pattern.

The recommended primary for the construct is form 1 (shell-CLI assistant). Form 2 (in-buffer plugin) is a conditional adjacent for users whose review-loop benefits from inline diffs. Forms 3 (graphical AI editor) and 4 (browser-based) are documented for completeness but are not the construct's recommended path. Form 5 is out of scope for this post.

The interaction style worth distinguishing alongside the form is whether the assistant edits with a diff loop or with an autonomous loop. In a diff loop, the human reviews each proposed change before it lands; in an autonomous loop, the assistant runs to completion across multiple steps without per-step approval. The construct's recommended default is the diff loop: every change to the versioned source passes through human review before being committed. Autonomous loops are useful for narrow, well-scoped tasks (running a long batch of refactors across files; sweeping a known fix across a corpus) but should be explicit, opt-in, and bounded in scope.

Prerequisites

The integration assumes:

- **Anthropic API key.** A Claude API key, stored in `pass` per AWS and pass: A Three-Tier Secrets Scheme and injected via the `with-secret` helper. Long-lived API keys exported in `.zshrc` are a specific anti-pattern this post avoids.
- **Existing construct setup.** Posts 01 (Shell), 24 (Dotfiles), 25 (Git), 26 (Editor), 49 (zzgit), 52 (construct framing), and 55 (secrets management) should be in place. The integration depends on each: the shell for the CLI, the dotfiles for the alias, git and zzgit for the diff-review and staging-scan loop, the editor for the buffer the human reviews diffs in, and post 52's construct framing for the per-project `CLAUDE.md` convention.
- **Per-project convention discipline.** The integration's largest single ergonomic input is the per-project `CLAUDE.md` file documenting the project's layout, conventions, and persona. A first-pass `CLAUDE.md` takes ~15 minutes to author per project and is amortized across every subsequent assistant invocation in that project.
- **Time investment.** Approximately 30 minutes for the initial Claude Code installation and shell wrapper. The per-project `CLAUDE.md` authoring is incremental and per-project. Optional in-editor plugin configuration adds a further 30-60 minutes per editor.

Installation

Claude Code (recommended primary)

Claude Code is distributed as an `npm` package and a Homebrew formula. The Homebrew installation is preferred on macOS for consistency with the construct's other tools.

```
# macOS
brew install claude

# Or via npm (cross-platform)
npm install -g @anthropic-ai/claude-code

# Confirm installation
claude --version
```

The CLI does not include the API key by default; it reads `ANTHROPIC_API_KEY` from the environment. Per the secrets discipline (post 55), the key is stored in `pass` and injected at invocation time, not exported into the shell environment.

The relevant `.zshrc` block (extending the helper from post 55):

```
# ============================================================
# Claude Code wrapper (post 60)
# ============================================================
# Inject the Anthropic API key from pass at invocation time
# rather than exporting it into the shell environment.
claude() {
  with-secret ANTHROPIC_API_KEY anthropic/api-key \
    command claude "$@"
}

# Optional: log every Claude session to a dated transcript
# for later audit (the existing claudelog alias from .zshrc
# is the canonical implementation).
```

The `command claude` invocation is required to bypass the function definition itself; without it, the function calls itself recursively.

Optional: Neovim in-buffer plugin

For users whose review-loop benefits from inline diffs (form 2 from the taxonomy above), `code-companion.nvim` is the recommended Neovim integration. It supports multiple model providers (including Anthropic) and respects the `ANTHROPIC_API_KEY` environment variable, so the same `pass` plus `with-secret` injection pattern works.

Installation via the construct's Lazy plugin manager (post 26):

```
-- ~/.config/nvim/lua/plugins/codecompanion.lua
return {
  'olimorris/codecompanion.nvim',
  dependencies = {
    'nvim-lua/plenary.nvim',
    'nvim-treesitter/nvim-treesitter',
```

```
  },
  config = function()
    require('codecompanion').setup({
      adapters = {
        anthropic = function()
          return require('codecompanion.adapters').extend(
            'anthropic', {
              env = {
                api_key = 'ANTHROPIC_API_KEY',
              },
            })
        end,
      },
      strategies = {
        chat = { adapter = 'anthropic' },
        inline = { adapter = 'anthropic' },
      },
    })
  end,
}
```

Invoke Neovim from a `with-secret` wrapper so the API key is available to the plugin:

```
# .zshrc addition for the Neovim case
nvim() {
  with-secret ANTHROPIC_API_KEY anthropic/api-key \
    command nvim "$@"
}
```

Optional: graphical alternatives (Cursor, Zed)

Cursor (a fork of VS Code, fork-maintained) and Zed (a ground-up implementation in Rust) are graphical AI editors that integrate model interactions directly into the editing surface. Installation is via the projects' standard distribution channels. Neither is the construct's recommended primary; both are noted for users whose preference for graphical editors is established and for whom the integration ergonomics outweigh the loss of editor portability across remote sessions.

Configuration

The per-project `CLAUDE.md` convention

The single most consequential configuration artifact in this integration is not the CLI's settings file but a per-project markdown file named `CLAUDE.md`, committed at the project's git root. The file documents the project's:

- **Conventions** (naming, formatting, idiom preferences)
- **Layout** (directory structure, where to find things)
- **Persona** (who the project's primary author is, what their working style is)
- **Don'ts** (specific anti-patterns to avoid; tools or approaches the project rejects)

The CLI reads `CLAUDE.md` automatically when invoked from inside the project's git root and uses its contents as persistent context for every subsequent prompt. The cost is ~15 minutes to author per project; the benefit is that the assistant's responses match the project's style without per-prompt re-explanation.

A representative `CLAUDE.md` for an applied-biostatistics project might be:

```
# CLAUDE.md

## Project Overview
This is a clinical-trial analysis project using zzcollab.
The primary deliverable is a Quarto-rendered methods paper
for a peer-reviewed journal.

## Conventions
- Native R pipe `|>` only; do not use `%>%`.
- snake_case for variables and functions.
- testthat for unit tests; integration tests under
  `tests/integration/`.
- Two-space indentation. No trailing whitespace.
- Avoid em dashes; use commas, semicolons, or colons.

## Layout
- `analysis/scripts/` for the analysis pipeline.
- `analysis/report/index.qmd` for the rendered paper.
- `R/` for reusable functions.
- `data/raw/` (read-only) and `data/derived/` for
  pipeline outputs.

## Persona
The primary author is a biostatistician working on
randomised trials. They value technical precision over
narrative flair and prefer scholarly third-person voice
over first-person.

## Don'ts
- Do not introduce non-CRAN package dependencies without
  discussion.
- Do not modify `data/raw/`; generate to `data/derived/`.
- Do not refactor working code that has tests passing
  without explicit request.
```

The file is intentionally short. Longer is not better; specific is better than general. A `CLAUDE.md` that says 'be careful' is useless; a `CLAUDE.md` that says 'do not modify `data/raw/`' is operative.

Global user instructions

In addition to per-project `CLAUDE.md`, the CLI reads a global `~/.claude/CLAUDE.md` if present. The construct's canonical home for global preferences (R style, citation conventions, scholarly voice, no first-person, no em dashes) is this file.

Verification

```
# CLI installed and reachable
claude --version

# API key reachable (via pass)
pass anthropic/api-key | head -c 12 ; echo
# Should print the first 12 chars of the key, then newline.

# Functional smoke test: a one-shot prompt
echo 'Reply with the single word OK and nothing else.' | claude
# Should print OK.

# Per-project context: cd into a project with a CLAUDE.md
# and confirm the assistant uses it.
cd ~/prj/some-project
claude  # Interactive mode; should reference project conventions
```

A green pass on all four indicates the integration is operational.

Daily Workflow

Pattern	Invocation (representative)	Notes
One-shot consultation	`echo 'question' | claude`	For simple lookups; no project context needed
Project-context session	`claude` (from project root)	Reads CLAUDE.md; interactive
Apply a refactor	`claude -p 'refactor R/utils.R: split into separate files for IO and transforms'`	Proposes a diff; human approves before applying
Document a function	`claude -p 'add roxygen2 docstring to R/fit_models.R::fit_logistic'`	Targeted; small diff

Pattern	Invocation (representative)	Notes
Generate tests	`claude -p 'add testthat tests for R/fit_models.R::fit_logistic, edge cases included'`	Adds to tests/testthat/
Review a diff	`git diff | claude -p 'review this diff for style and correctness'`	Read-only; no file changes
Explain a function	`claude -p 'explain R/fit_models.R::fit_logistic'`	Read-only
Out-of-editor research	`claude -p 'summarise the assumptions of mixed-effects logistic regression'`	No project files needed

Note the consistent pattern: the human owns the buffer and the staged commit. The assistant proposes diffs that are reviewed before applying and pass through `zzgit`'s secret-scanning gate before they reach the remote.

Figure 3: An open notebook with a handwritten numbered list of conventions on the left page, and a blank page on the right waiting to be filled.

Things to Watch Out For

Six gotchas have surfaced repeatedly in real use of this integration. Each is small in isolation; in aggregation they are the most common reasons users either over-rely on the assistant (treat-

ing its output as authoritative) or under-rely (refusing to use the integration after a single poor experience).

- **The model occasionally invents APIs that do not exist.** Particularly in less-common R packages, the model may produce calls to functions that look plausible but are not in the package's namespace. The diff-loop workflow catches these at review time; the autonomous loop does not. Run the proposed code in the project's Docker container before committing; treat 'compiles cleanly' as the floor for trust, not the ceiling.
- **Stale context is silent.** The model's training cutoff is fixed; the package being used may have moved on. A request that depends on a feature added after the cutoff will produce code that uses the pre-cutoff API. Cross-check against the package's current documentation rather than against the model's recollection of it.
- **Secrets in `CLAUDE.md` or in pasted prompts are sent to the API.** A `CLAUDE.md` file that contains a database password, an API key, or a private URL will be transmitted with every invocation. Audit `CLAUDE.md` before committing it; treat it the same way as any other publicly-readable file in the repository (which it is, once committed).
- **Cost runaway in autonomous loops.** A poorly-bounded autonomous task can issue dozens or hundreds of API calls before reaching a stopping condition. The CLI exposes session-cost tracking; check it before kicking off long autonomous runs and set explicit budget ceilings. The construct's default is the diff loop precisely to keep cost predictable.
- **The model has an over-engineering bias.** Asked to refactor, it will often introduce abstractions, layers, and configuration options that the project does not need. Push back explicitly: 'do not introduce abstractions for hypothetical future requirements; three similar lines is better than a premature abstraction.' This kind of guidance belongs in the project's `CLAUDE.md` if it is a persistent preference.
- **Inline plugin diffs can land without git review.** In-buffer plugins (`codecompanion.nvim`, `avante.nvim`, Copilot) apply changes directly to the buffer; the human reviews in the editor but the change is not staged through `zzgit`. The construct's secret- scanning gate (post 49) catches secrets at staging time regardless, but the project's review discipline depends on the human staging deliberately rather than running a blind `git add -A` after a plugin-driven session.

Uninstall / Rollback

```
# Remove the CLI
brew uninstall claude            # macOS
# or npm uninstall -g @anthropic-ai/claude-code

# Remove the .zshrc wrapper functions for `claude` and
# `nvim` if added.

# Remove per-project CLAUDE.md files (optional; they are
# committed to git, so removal is a deliberate commit, not
# a `rm`).
```

```
# Optional: revoke the Anthropic API key in the Anthropic
# console and remove the entry from pass.
pass rm anthropic/api-key

# Optional: remove the Neovim plugin
# Edit ~/.config/nvim/lua/plugins/codecompanion.lua to remove
# the spec, then run :Lazy clean inside Neovim.
```

The integration is non-destructive at every layer. None of the above operations affect the project's source code, the git history, or the secrets store beyond the specific entry removed.

Figure 4: A single chess knight resting on a nearly-empty board next to a closed rulebook, suggesting one deliberate, constrained move rather than free play.

Lessons Learned

Working through this integration on the qblog and on three applied analysis projects surfaced lessons grouped into three buckets.

Conceptual

- **The diff loop is the construct's natural integration.** Every existing piece of construct machinery (modal editor, secret-scanning git wizard, container-bound R session, reproducible compendium) operates unchanged when the assistant proposes diffs that pass through the same staging-and-review path as a hand-typed change. The autonomous loop is occasionally useful but is not the default.

- **`CLAUDE.md` is the load-bearing artifact.** A model with rich per-project context produces output that is in the project's voice and respects the project's conventions. A model without that context produces output that is in the model's default voice and ignores conventions silently. The 15-minute upfront cost of a good `CLAUDE.md` is amortized across every subsequent invocation in that project.
- **The assistant is a collaborator, not an oracle.** A workflow that treats the assistant's output as authoritative produces commits the human cannot defend in code review. A workflow that treats the assistant's output as a draft to review produces commits the human can defend. The latter is the only sustainable mode for research code that may be cited in a methods paper.
- **The shell CLI's portability is a feature, not a consolation.** A shell-CLI integration works identically on the laptop, on the EC2 instance over SSH, and inside the construct's Docker container. A graphical AI editor's integration works only where the editor runs. For a workflow that deliberately moves between local, remote, and containerized contexts, portability is load-bearing.

Technical

- **Per-process API-key injection beats environment export.** The `with-secret ANTHROPIC_API_KEY` pattern from post 55 applies unchanged. A diagnostic script that runs `env` does not see the key; a CI runner that inherits the shell environment does not see the key; shell history does not see the key.
- **Project-level `CLAUDE.md` and global `~/.claude/CLAUDE.md` compose.** The global file documents persistent personal preferences (style, citation, voice); the project file documents project-specific conventions. Both are read; both shape the assistant's responses; neither overrides the other.
- **The CLI's session-cost metering is sufficient for diff-loop budgeting.** Each invocation reports its token use and approximate cost. A daily-cost ceiling set by the operator (informally, by attention to the metering) is sufficient for most use; explicit budget-enforcement via `--max-cost` flags is available for autonomous loops.
- **`zzgit` plus the diff loop catches assistant- introduced secret leaks at staging time.** The staging-time `gitleaks` scan is content-based, not origin-based, so a leak from an assistant-proposed diff is caught the same way as a leak from a hand-typed diff. No special integration is needed.

Gotcha-shaped

- **`claude` invoked from outside the project root sees no `CLAUDE.md`.** Move into the project's git root before invoking the CLI. The CLI emits a small banner noting which `CLAUDE.md` it loaded, useful for confirming the right file is in scope.
- **`CLAUDE.md` updates that reduce a previous behavior do not retroactively reverse the previous behavior.** If a long-running session has been operating without an instruction, adding the instruction to `CLAUDE.md` affects subsequent invocations. End the current session and start a fresh one to ensure the new instruction is in scope.
- **Some R idioms are model-disfavoured.** The native R pipe `|>` is treated as a recent feature by older training corpora; explicit instruction in `CLAUDE.md` ('use the native R pipe `|>`

only; do not use `%>%`) mitigates. Similar for `data.table` versus `dplyr`, `base::ifelse` versus `dplyr::if_else`, and other preference-bearing idioms.
- **The CLI does not currently support per-tool budget ceilings out of the box.** A user who runs autonomous loops should set up an external budget ceiling (a daily cron that polls the Anthropic billing API and alerts above a threshold) until the CLI exposes per-session budget enforcement directly.

Limitations

The integration as documented has the following honest limitations:

- **It depends on a hosted API.** Claude Code requires network access and an active Anthropic API account. Air-gapped or institutionally network-restricted environments cannot use this integration. Local-model alternatives (Ollama with `aider`) are noted in the alternatives section but are not the recommended primary.
- **The integration's quality is upper-bounded by the model's quality.** A request the model cannot answer produces output the human will spend more time correcting than they would have spent writing from scratch. Internalize the cases where the assistant helps (boilerplate, refactors, documentation, test scaffolding) and the cases where it does not (novel statistical method derivations, domain-specific code that is rare in the training corpus).
- **Per-project `CLAUDE.md` files are version-controlled artifacts and inherit version-control's coordination costs.** A team-shared `CLAUDE.md` requires the same branch-merge discipline as any other shared file. A preference change in one branch can conflict with a different change in another.
- **The post does not address fine-tuning, RAG (retrieval augmented generation), or other application-level integrations.** Those are application-engineering concerns, not editing concerns. See the Anthropic SDK documentation for the application path.
- **The post is Anthropic-Claude-specific in its primary recommendation.** The `aider` open-source CLI supports multiple model providers (OpenAI, Anthropic, local via Ollama) and is a reasonable provider-portable alternative; it is mentioned in the alternatives section but is not the canonical primary in this revision.

Alternatives

`claude` (Claude Code) is the recommended primary, but three alternatives deserve mention.

`aider`: an open-source, multi-provider CLI

`aider` is an open-source Python CLI that supports Anthropic, OpenAI, local models via Ollama, and a number of other providers. The interaction model is similar (diff loop, project-context awareness, git-staging integration). The case for `aider` over `claude` is provider portability (useful when budget or air-gap concerns require a different model) and source-code transparency. The case against is a slightly less polished UX and a less coherent project-context convention than the `CLAUDE.md` file.

In-buffer plugins beyond `codecompanion.nvim`

`avante.nvim` is a Neovim plugin modelled on Cursor's in-editor UX, with a chat panel and inline diffs. The case for `avante.nvim` over `codecompanion.nvim` is the slightly more polished diff display; the case against is a less mature plugin ecosystem at the time of writing. For VS Code users who do not want to switch editors, GitHub Copilot's inline suggestions (subscription-based) and the Anthropic-published Claude extension are the two established options.

Browser-based prompting

The browser-based pattern, in which the human pastes code into a chat surface and pastes the result back into the editor, is documented in Prototyping a Shiny App with ChatGPT as a worked case study with a 2023-era ChatGPT for a Shiny app. The post is retained as a historical reference for one specific older integration pattern; it is not the construct's recommended primary in 2026, because the shell-CLI integration achieves the same outcome with less context loss and tighter integration into the construct's other layers.

Opportunities for Improvement

Several extensions are plausible for subsequent revisions:

1. **Local-model option via `ollama` plus `aider`.** Documents the air-gapped path: a local Llama or Qwen model running on the laptop or a local server, with `aider` as the CLI. Reduces the API-cost concern to zero and addresses the institutional-network restriction.
2. **A construct-aware `CLAUDE.md` template generator.** A `~/bin/claudemd-init.sh` script that creates a first-pass `CLAUDE.md` for a new project by inspecting the project's structure, Dockerfile, and `DESCRIPTION` file. Reduces the per-project authoring cost from 15 minutes to 5.
3. **Per-team `CLAUDE.md` patterns.** A small set of `CLAUDE.md` templates for common project archetypes (clinical-trial analysis, simulation study, methods paper, R package). Authored once, reused across projects.
4. **Integration with `zzgit` for assistant-attribution.** A small extension to the construct's git wizard (post 49) that detects when staged content was produced via Claude Code and adds a `Generated-by: claude-code` trailer to the commit message, for downstream reproducibility audits.
5. **A short follow-up on autonomous-loop discipline.** Documents the cases where the autonomous loop is the right tool (large-scale rename across files, sweep of a known mechanical change), with explicit budget and stopping criteria.

Wrapping Up

LLM-augmented editing is, in 2026, a productive part of an applied biostatistician's daily work, but its productivity depends on which integration pattern the operator adopts. The shell-CLI pattern documented here keeps the human's existing editor, the project's git history, and the construct's

review and audit layers all unchanged; the assistant participates as a deferential collaborator that proposes diffs the human reviews before committing. The graphical-AI-editor pattern (Cursor, Zed) achieves a tighter UX at the cost of editor portability and a partial fragmentation of the project's authoritative artifacts. The browser-based pattern (post 36) achieves a working result at the cost of substantial context loss between chat surface and editor.

The single most consequential configuration choice is not the integration form but the per-project `CLAUDE.md` file. A 15-minute investment per project pays for itself within hours: the assistant's output matches the project's voice, respects its conventions, and avoids the anti-patterns the project rejects. Without a `CLAUDE.md`, the assistant produces output in its default voice, which is reasonable but not aligned with any specific project.

The construct's default is conservative: diff loop, human review, secret-scanning at staging, source-code transparency where it matters most. The recommendations made here are not absolute: autonomous loops are occasionally useful, in-buffer plugins suit some workflows better, and the browser pattern remains the floor for cases where no other access is available. The defaults should hold, though, for the same reason the construct's other defaults hold: predictability, auditability, and consistency across the construct's other layers.

In conclusion, four points merit emphasis. First, the shell-CLI integration is the construct's natural fit: it composes with the existing editor, dotfiles, and git-wizard layers without disturbing them. Second, `CLAUDE.md` is the load-bearing artifact; the 15-minute investment per project should not be skipped. Third, the diff loop is the predictable default: autonomous loops should be explicit, opt-in, and bounded. Fourth, API keys should be injected per-process via `pass`; long-lived exports in `.zshrc` are an anti-pattern this integration specifically avoids.

See Also

- Posts in this repository at adjacent layers:
  - Unix Command-Line Workspace Setup: the Shell layer where the `claude` wrapper function lives.
  - Multi-Laptop macOS Bootstrap: the workstation-IaC keystone where the wrapper is version-controlled.
  - Setting Up Neovim as a Data Science IDE: the Editor layer; the in-buffer plugin extends the configuration here.
  - Prototyping a Shiny App with ChatGPT: the historical browser-based ChatGPT pattern, retained as a case study.
  - The `zzgit` staging-time secret scanner catches API-key leaks regardless of whether the diff was hand-typed or assistant-proposed.
  - Building a statistical computing textbook in the Age of AI: the cross-cutting essay on using LLM tools in a textbook-authoring workflow.
  - A Workflow Construct for the Modern Data Scientist: the construct framing.
  - AWS and pass: A Three-Tier Secrets Scheme: the secrets discipline that the API-key injection here depends on.
- External references:
  - Claude Code documentation: https://docs.claude.com/claude-code

- Anthropic API and SDK: https://docs.anthropic.com/
- `aider` (multi-provider CLI): https://aider.chat/
- `codecompanion.nvim`: https://github.com/olimorris/codecompanion.nvim
- `avante.nvim`: https://github.com/yetone/avante.nvim
- Cursor: https://cursor.com/
- Zed: https://zed.dev/
- Ollama (local models): https://ollama.com/

Reproducibility

The configuration was developed and verified on the following software stack:

Component	Version	Notes
Operating system	macOS 15 (Sequoia)	primary daily driver
Operating system	Linux Mint 22 (Wilma)	secondary verification
Shell	zsh 5.9	minimum 5.0
Claude Code CLI	1.x (current)	install via `brew install claude`
`pass`	1.7.4	for API-key injection (post 55)
Optional Neovim plugin	`codecompanion.nvim` 12.x	minimum 10
Optional editor	Neovim 0.11	for plugin route
Optional alternative CLI	`aider` 0.60	multi-provider option
Reference: model	claude-opus-4-7-20251015	the version used during authoring

Date of last verification: 2026-04-28.

Feedback

Corrections, suggestions, and questions are welcome. Please open an issue or pull request on the GitHub repository or send an email to user@example.com. Particularly welcome are contributions documenting the local-model path (`ollama` plus `aider`), per-team `CLAUDE.md` patterns that have proven useful in practice, and reports of integration patterns the construct's defaults handle poorly.

Related posts

This post is part of the *Workflow Construct* series. This post is not part of the core onboarding sequence: it lists five other posts in this repository as prerequisites (shell, dotfiles, git, editor, and secrets-management setup) that should already be in place. It belongs to a separate, smaller *Workflow Extensions* grouping alongside A pocket terminal with ttyd and Tailscale, for optional/advanced additions layered on top of a working workstation rather than day-one setup.

Core onboarding series (*Workflow Construct*), for readers who have not yet completed the base setup:

1. Post 15: A Workflow Construct for the Modern Data Scientist
2. Post 16: Unix Command-Line Workspace Setup for Data Science
3. Post 17: Multi-Laptop macOS Bootstrap
4. Post 18: Setting Up Git for Data Science Workflows
5. Post 19: Setting Up Neovim as a Data Science IDE
6. Post 21: Modern CLI Replacements for the Shell Layer
7. Post 25: Install Linux Mint on a MacBook Air