Claude Code for Biostatistics Workflow

A Hands-On Tutorial

Eric Zhang

03/23/2026

What is Claude Code?

Claude Code is Anthropic's **command-line AI assistant** for software engineering and data science. It runs in your terminal, reads your project files, executes shell commands, and writes code — all through natural language conversation. For biostatisticians, it serves as a programmable co-pilot that understands R, Python, SAS, and other programming languages, as well as clinical trial conventions.

Core Features

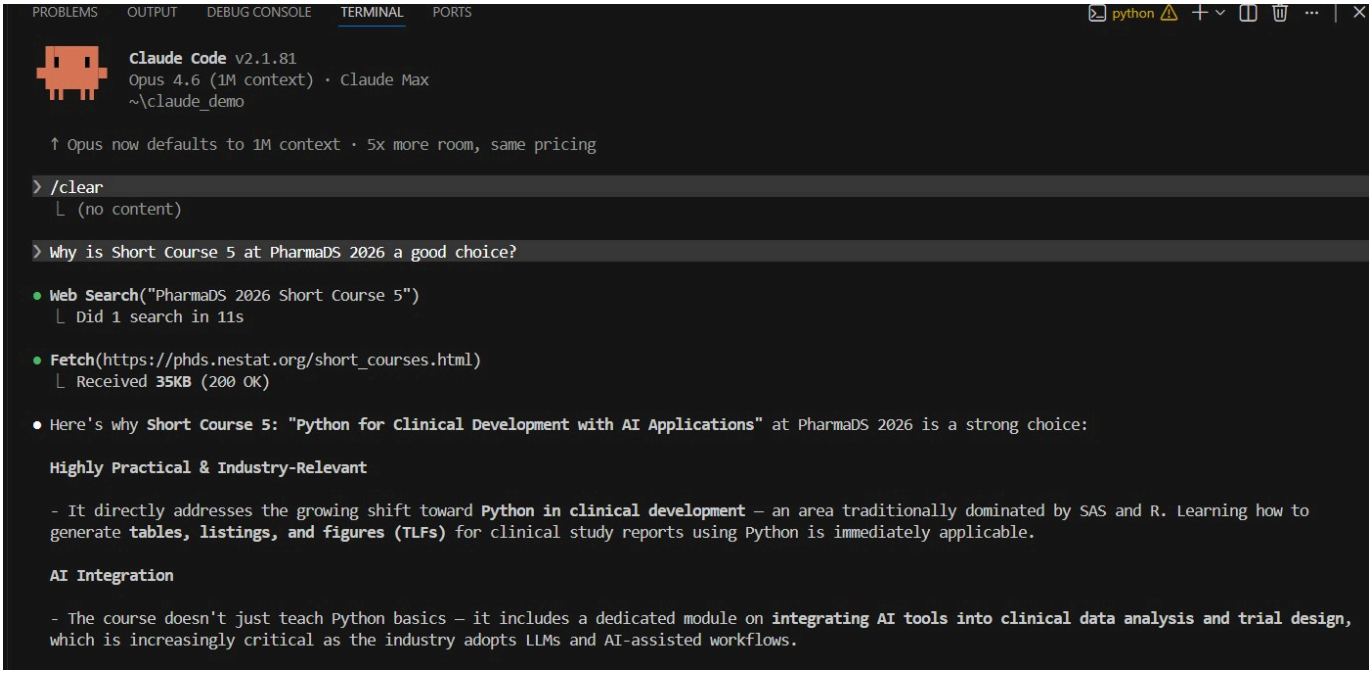
Feature	What It Is	When to Use
CLAUDE.md	A project instruction file at the repo root. Claude reads it automatically every session.	Define data dictionaries, file paths, R environment config, conventions
Sub-Agents	Autonomous workers defined in markdown (.claude/agents/). They receive a goal and independently call tools to achieve it.	Multi-step tasks: run 8 R scripts, read data, write outputs, handle errors
Skills	Domain knowledge modules loaded on demand via /slash commands. Each has a SKILL.md + optional companion scripts.	Specialized domains: group sequential design, RECIST criteria, competitive intelligence
MCP Servers	External tool integrations via the Model Context Protocol. Can be pre-built plugins or custom Python/R servers.	Connect to APIs (ClinicalTrials.gov), run custom analysis pipelines
Built-in Tools	File read/write, Bash shell, web search, code editing, and more — available in every session without configuration.	Day-to-day coding, file manipulation, literature search, data exploration

How a Typical Session Works

1. You type a natural language request in the terminal (e.g., *"generate the demographics table"*).
2. Claude reads your `CLAUDE.md`, relevant data files, and any loaded skills to understand the context.
3. It writes and executes code (R scripts, Python, shell commands), iterating if errors occur.

4. Outputs (tables, figures, reports) are saved to disk. Claude summarizes what was produced.

Key Takeaway: Claude Code is not just a chatbot · it is an extensible platform. CLAUDE.md configures it, sub-agents automate multi-step workflows, skills inject domain expertise, and MCP servers connect it to external data sources. The four use cases in this tutorial demonstrate each of these extension points.



Claude Code CLI Installation Tutorial

A step-by-step guide for Windows and macOS

Platform	`Windows 10/11 & macOS 12+`
Runtime	`Node.js 18+`
Install command	`npm install -g @anthropic-ai/claude-code`
Time required	`~5 minutes`

1. Prerequisites

Before installing Claude Code, make sure you have the following:

- **Node.js 18 or later** — a JavaScript runtime that lets you run code outside the browser. Claude Code is built on Node.js
- **npm** (Node Package Manager) — the default package manager for Node.js, used to install and manage JavaScript packages. It comes bundled with Node.js
- **An Anthropic account** — with an API key or a Claude Max/Pro subscription
- **Git** (recommended) — for repository-aware features

2. Install on Windows

Step 1 — Install Node.js

Download the Windows installer (.msi) from **nodejs.org** and run it. Choose the LTS version (v18 or v20+). Accept the defaults and ensure "Add to PATH" is checked.

Download link: https://nodejs.org/en/download

Verify the installation by opening **PowerShell** or **Command Prompt**:

```
node --version
npm --version
```

Step 2 — Install Claude Code

Claude Code requires an active subscription. Choose a plan at https://claude.com/pricing

Feature	Free	Pro	Max
Monthly price	**$0**	**$20**	**$100 / $200**
Claude Code access	Limited	Included	Included
Usage	Basic	5x more than Free	5x / 20x more than Pro
Models	Sonnet, Haiku	Opus, Sonnet, Haiku	Opus, Sonnet, Haiku
Extended thinking	—	Yes	Yes (higher limits)

Once you have subscribed, open **PowerShell** or **Command Prompt** and run the following command to install Claude Code:

```
npm install -g @anthropic-ai/claude-code
```

> *Note: If you see permission errors, run the terminal as **Administrator**, or use a Node version manager like **nvm-windows**.*

Step 3 — Authenticate

Navigate to your project folder and launch Claude Code:

```
cd your-project
claude
```

On first launch, Claude Code will open a browser window for authentication. Sign in with your Anthropic account. Alternatively, set your API key:

```
set ANTHROPIC_API_KEY=sk-ant-...
```

Step 4 — Verify

```
claude --version
```

You should see the installed version printed. You're ready to go!

3. Install on macOS

Step 1 — Install Node.js

The recommended way is via **Homebrew**. If you don't have Homebrew, install it first:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Then install Node.js:

```
brew install node
```

Alternatively, download the macOS installer from **nodejs.org** (LTS version):
https://nodejs.org/en/download

Verify:

```
node --version
npm --version
```

Step 2 — Install Claude Code

Claude Code requires an active subscription. Choose a plan at https://claude.com/pricing (see the plans comparison table in Section 2 above).

Once you have subscribed, open **Terminal** and run the following command to install Claude Code:

```
npm install -g @anthropic-ai/claude-code
```

> *Note: If you get EACCES permission errors, do **not** use sudo. Instead, fix npm permissions or use **nvm** (Node Version Manager).*

Step 3 — Authenticate

Navigate to your project and start Claude Code:

```
cd your-project
claude
```

A browser window will open for OAuth sign-in. Or set your API key in the shell:

```
export ANTHROPIC_API_KEY=sk-ant-...
```

Step 4 — Verify

```
claude --version
```

4. Quick Reference

Command	Description
`claude`	Start interactive session in current directory
`claude --version`	Print installed version
`claude --help`	Show all available flags and options
`claude /help`	Show help inside a session
`claude /model`	Switch model (Opus, Sonnet, Haiku)
`claude /clear`	Clear conversation context
`claude /cost`	Show token usage and cost
`claude "fix the bug"`	Run a one-shot command (non-interactive)
`claude config`	Open configuration settings

5. Troubleshooting

"command not found: claude"

Node.js global bin directory is not in your PATH. Run **npm config get prefix** and add the resulting path's **bin** folder to your system PATH.

Permission errors on install

Windows: Run terminal as Administrator.
macOS: Use nvm or fix npm directory ownership — never use **sudo npm**.

Node.js version too old

Claude Code requires Node.js 18+. Run **node --version** and upgrade if needed.

Authentication fails

Ensure your browser can reach the Anthropic auth page. If behind a proxy, set **HTTPS_PROXY**. You can also authenticate via API key instead.

Tutorial Outline

Use Case	Tool Type	Key Feature
1. TFL Generation	Sub-Agent	Automated tables, figures & listings
2. ClinicalTrials.gov Meta-Analysis	MCP Server	Trial search + ORR-OS regression
3. Group Sequential Trial Design	Skill (/gsDesign)	Boundary tables & sensitivity analysis
4. Competitive Intelligence	Skill (/competitive _intelligence)	Probability of success estimation

Proiect Structure

```
project-root/
    CLAUDE.md                  ← Project instructions, data dictionary, R config
    .claude/
        agents/                ← Sub-agent definitions (e.g., oncology-trial-outputs.md)
        skills/                ← Skill modules (gsDesign/, competitive_intelligence/,
etc.)
    mcp/
        ctgov/                 ← Custom MCP server (ClinicalTrials.gov analysis)
    data/raw/              Raw CSV datasets (dm, ae, tu, rs, ex, ds, dd)
    Rscripts/                R scripts for TFL generation
    output/                  Generated tables, figures, reports
    reference/               RECIST guidelines, gsDesign manual, mock-ups
```

TFL Generation

Tool: Sub-Agent

What It Does

Generates a complete set of Tables, Figures, and Listings (TFLs) for an oncology clinical trial by orchestrating 8 R scripts in sequence. The sub-agent reads the raw CSV data (demographics, adverse events, tumor assessments, exposure, disposition) and produces formatted RTF tables and PNG figures, then combines everything into a single document.

How to Invoke

```
@"oncology-trial-outputs (agent)" refresh the TFLs
```

The @ prefix tells Claude Code to route the request to the named sub-agent rather than handling it directly. The agent definition lives in `.claude/agents/oncology-trial-outputs.md`.

Workflow

1. Agent Activation

Claude reads the agent definition and understands what R scripts are available, what data domains exist, and the expected output format.

2. Script Execution

The agent runs each R script via the full Rscript path: `"/c/Program Files/R/R-4.4.1/bin/Rscript.exe" Rscripts/combined_tlf.R`

3. Output Generation

8 scripts produce 5 RTF tables + 2 PNG figures + 1 combined document, all written to the `output/` folder.

4. Summary Report

The agent returns a structured summary with file names, descriptions, and key metrics (e.g., 30 subjects, 100% AE rate, best responses PR=7 / SD=18 / PD=5).

What is a Sub-Agent?

A **sub-agent** is an autonomous worker defined in a markdown file under `.claude/agents/`. It receives a high-level instruction and independently decides which tools to call, in what order, and how to handle errors. Sub-agents are ideal for **multi-step, repeatable tasks** that involve coordinating several tools (e.g., running multiple R scripts, reading data files, writing outputs).

- **When to use:** Complex tasks requiring 5+ tool calls in a specific sequence
- **How to create:** Write a markdown file in `.claude/agents/` describing the task, available tools, data sources, and expected outputs
- **How to invoke:** Use the `@"agent-name"` prefix in your prompt

Key Takeaway: Sub-agents turn Claude Code into a task orchestrator. You define the workflow once in a markdown file, then invoke it with a single command. The agent handles all the steps autonomously.

How This Workflow Was Built

The entire TFL pipeline was developed iteratively with Claude Code. The process:

1. We created the agent definition (`.claude/agents/oncology-trial-outputs.md`) describing the task, available data domains, and output expectations.
2. We provided mock-up tables (`reference/mockup/Mock Up Tables.docx`) as the target format, and annotated CRFs (`aCRF/`) so the agent understands how each data field maps to the source collection forms.
3. The agent reads raw CSV data from `data/raw/` and auto-generates R scripts under `Rscripts/` to produce each TFL.
4. We reviewed outputs and gave feedback in conversation. The agent refined scripts iteratively until tables and figures matched the mock-ups.

File Architecture

```
.claude/agents/
  oncology-trial-outputs.md ← Agent definition
aCRF/
  *.pdf ← Annotated CRFs (data field mappings)
reference/mockup/
  Mock Up Tables.docx ← Target table formats
data/raw/
  dm.csv, ae.csv, tu.csv, rs.csv, ... ← Input data
Rscripts/
  demographics_table.R, ae_summary_table.R, ... ← Auto-generated by agent
output/
  *.rtf, *.png ← Final outputs
```

Estimated Token Cost

Development: ~500K–1M tokens / ~$3.50–$7.00 (5–10 iterative sessions to build agent definition, generate and refine 8 R scripts, match mock-up formatting)
Per invocation: ~50K–100K tokens / ~$0.35–$0.70 (agent reads definition, executes scripts, returns summary)

ClinicalTrials.gov Search & Meta-Analysis

Tool: MCP Server

What It Does

Searches ClinicalTrials.gov for clinical trials matching user-specified criteria (e.g., indication, phase, intervention type), extracts efficacy endpoints from each trial, then runs a meta-regression to quantify relationships between endpoints (e.g., delta-ORR vs OS HR). Produces a scatter plot and a formatted Word report.

How to Invoke

```
Search ClinicalTrials.gov for [phase], [design] trials in [indication]
with [intervention] ... report results for [endpoints].
```

No special prefix is needed. Claude Code automatically selects the ClinicalTrials.gov MCP server tools based on the query content.

Workflow

1. Trial Search

The ClinicalTrials.gov plugin queries for trials matching the user's criteria and returns those with posted results sections.

2. Data Extraction

The AI reads each trial's results to extract efficacy endpoints (e.g., ORR by arm, OS HR). It handles both CT.gov-posted results and published literature.

3. Quality Review

The AI flags data extraction caveats — e.g., mismatched arms, non-response rates mistaken for ORR — for user verification before proceeding.

4. Meta-Regression

The custom MCP server runs an R script using the metafor package to fit a weighted regression (e.g., OS HR on delta-ORR), producing a bubble scatter plot.

5. Report Generation

A Word report is generated with trial-level data tables, regression coefficients, interpretation, and the scatter plot.

What is an MCP Server?

The **Model Context Protocol (MCP)** is an open standard that lets Claude Code connect to external tools and data sources. An MCP server exposes tools that the AI can call just like built-in functions. There are two types used here:

- **Plugin MCP servers** (pre-built): The ClinicalTrials.gov plugin provides search_trials, get_trial_details, and other tools out of the box
- **Custom MCP servers** (user-built): The `mcp/ctgov/server.py` server adds custom analysis tools (endpoint classification, meta-regression) that call R scripts
- **When to use:** When you need to connect Claude Code to an external API or run custom analysis pipelines that combine Python orchestration with R computation
- **How to create:** Write a Python server using FastMCP, define tool functions, and register it in your project's MCP configuration

Key Takeaway: MCP servers extend Claude Code's capabilities beyond its built-in tools. They are ideal for integrating external APIs, databases, or custom statistical pipelines into your AI workflow.

How This Workflow Was Built

This workflow combines a pre-built MCP plugin with a custom MCP server we developed:

1. We installed the **ClinicalTrials.gov MCP plugin** for trial search — this is a community plugin that provides search and detail-retrieval tools out of the box.

2. We built a **custom MCP server** (`mcp/ctgov/server.py`) using FastMCP to add analysis capabilities not available in the plugin: endpoint classification and meta-regression.

3. The server delegates statistical computation to R scripts (`mcp/ctgov/tools/orr_os_report.R`) that use the *metafor* and *officer* packages for regression and Word report generation.

4. We iterated on data extraction logic with Claude — reviewing how it parsed each trial's results section and correcting misclassifications through conversational feedback.

File Architecture

```
mcp/ctgov/
  server.py ← FastMCP server (endpoint classification)
  tools/
    orr_os_report.R ← Meta-regression + Word report
    orr_os_compute.R ← Statistical computations
output/
  orr_os_scatter.png Bubble scatter plot
  ORR_OS_...Report.docx Word report
```

Estimated Token Cost

Development: ~300K–500K tokens / ~$2.00–$3.50 (3–5 sessions to build MCP server, write R analysis scripts, test endpoint classification and extraction logic)
Per invocation: ~100K–200K tokens / ~$0.70–$1.40 (trial data from CT.gov can be large; includes search, extraction, regression, and report generation)

Group Sequential Trial Design

Tool: Skill (/gsDesign)

What It Does

Designs a group sequential clinical trial for a survival endpoint (OS, PFS) with interim analyses, efficacy and futility boundaries, and generates a comprehensive Word report. The /gsDesign skill loads domain knowledge about the gsDesign R package, spending functions, and boundary calculations.

How to Invoke

```
/gsDesign
```

The slash command loads the skill, and the AI prompts for trial design parameters: endpoint, control arm median survival, target hazard ratio, power, number of interim analyses, spending functions, and enrollment assumptions.

Workflow

1. Skill Loading

The /gsDesign command loads the skill definition from `.claude/skills/gsDesign/SKILL.md`, which contains knowledge about the gsDesign R package, gsSurv() function, and boundary computation.

2. Interactive Prompting

The AI asks for trial design parameters: endpoint, control arm median, target HR, power, number of interim analyses, spending functions, and enrollment assumptions.

3. R Script Generation & Execution

The AI writes an R script using gsSurv(), runs it via `Rscript.exe`, and parses the output into structured tables.

4. Iterative Refinement

The user can request follow-up changes (e.g., add futility boundaries, adjust alpha). The AI correctly interprets intent even with typos — e.g., inferring "alpha=0.020" from "alpha-0.2".

5. Report Generation

A formatted Word report is generated with design assumptions, boundary tables, timeline plots, and reproducible R code.

What is a Skill?

A **skill** is a domain knowledge module stored in `.claude/skills/`. Each skill contains a `SKILL.md` file that provides the AI with specialized instructions, terminology, and workflows for a specific domain. Skills can also bundle companion scripts (R, Python) that the AI calls with the correct parameters.

- **When to use:** When the AI needs domain expertise to correctly parameterize tools, interpret results, or follow domain-specific conventions
- **How to create:** Write a `SKILL.md` file with a YAML frontmatter (name, description) and markdown body explaining the domain knowledge
- **How to invoke:** Type `/skill-name` (e.g., `/gsDesign`) to load it on demand

> *Key Takeaway: Skills turn Claude Code into a domain expert. The SKILL.md file acts as a reference manual that the AI consults during execution, enabling it to correctly use specialized R packages and generate protocol-ready outputs.*

How This Workflow Was Built

The skill was built by encoding domain knowledge into a single SKILL.md file:

1. We wrote `.claude/skills/gsDesign/SKILL.md` with instructions on the gsDesign R package: key functions (*gsSurv*, *gsDesign*), spending function options, and how to parameterize survival designs.
2. We included the **gsDesign technical manual** (`reference/gsdesign/gsDesign.pdf`) as a reference document the AI can consult for edge cases and advanced options.
3. Through iterative testing, we refined the SKILL.md to handle common user requests: adding futility boundaries, adjusting alpha allocation, switching spending functions, and generating protocol-ready Word reports.
4. No companion R script is bundled — the AI generates trial-specific R code on the fly based on the user's design parameters and the domain knowledge in SKILL.md.

File Architecture

```
.claude/skills/gsDesign/
  SKILL.md ← Domain knowledge (functions, parameters)
reference/gsdesign/
  gsDesign.pdf ← Technical manual (AI reference)
output/
  gsdesign_report.docx Word report with boundary tables
```

Estimated Token Cost

Development: ~200K–400K tokens / ~$1.40–$2.80 (3–5 sessions to write SKILL.md, test across various design scenarios, refine boundary table formatting)
Per invocation: ~30K–50K tokens / ~$0.20–$0.35 (skill loading + R script generation and execution)

Competitive Intelligence

Tool: Skill (/competitive_intelligence)

What It Does

Estimates the probability that a competitor's clinical trial will achieve statistical significance at the final analysis, based on publicly released interim analysis results. Uses group sequential design methods to reverse-engineer the alpha spending strategy and compute conditional power.

How to Invoke

```
/competitive_intelligence
```

The skill loads, then the AI searches the web for the latest interim analysis results for the trial of interest and asks the user to confirm or provide key parameters.

Workflow

1. Skill Loading

The /competitive_intelligence command loads the skill definition, which includes the R script template for alpha spending, conditional power, and HR boundary calculations.

2. Web Search

The AI uses the built-in WebSearch tool to find the latest press releases and publications with key interim analysis data points for the trial of interest.

3. Parameter Collection

The AI presents a summary table of known vs unknown parameters, separating what was found from public sources vs what the user must provide (e.g., design HR, power).

4. R Computation

The companion R script (`.claude/skills/competitive_intelligence/competitive_intelligence.R`) is called with the collected parameters to perform the statistical calculations.

5. Report Generation

A formatted Word report is generated with sections for Trial Background, Assumptions (with source attribution), Results, and Conclusion.

Combining Multiple Features

This use case demonstrates how Claude Code features can be **combined**. The skill provides domain knowledge (group sequential design theory, alpha spending functions), while the built-in WebSearch tool gathers real-time data from the internet:

- **Skill** provides the statistical methodology and R script templates
- **WebSearch** finds the latest interim analysis results from press releases
- **Bash tool** executes the R script with the correct parameters
- **User interaction** fills in assumptions not available from public sources

Key Takeaway: The most powerful workflows combine multiple Claude Code features. Skills provide domain expertise, MCP servers connect to external data, sub-agents orchestrate complex tasks, and built-in tools (WebSearch, Bash, Read/Write) handle the glue work.

How This Workflow Was Built

This skill combines domain knowledge with a companion R script and built-in web search:

1. We wrote `.claude/skills/competitive_intelligence/SKILL.md` encoding the methodology: alpha spending function estimation, conditional power theory, and HR boundary calculation.

2. We developed a **companion R script** (`competitive_intelligence.R`) using the *gsDesign* package to perform the statistical computations. The AI calls this script with trial-specific parameters rather than writing code from scratch.

3. The workflow uniquely **combines multiple Claude Code features**: the skill provides methodology, **WebSearch** gathers real-time interim analysis data from press releases, and the user fills in assumptions not available from public sources.

4. We refined the skill through real use cases — each trial had different publicly available parameters, so the SKILL.md was updated to handle partial information gracefully.

File Architecture

```
.claude/skills/competitive_intelligence/
  SKILL.md ← Methodology + parameter guidance
  competitive_intelligence.R ← Companion R script (gsDesign)
reference/gsdesign/
  gsDesign.pdf ← Technical reference (shared with UC3)
output/
  competitive_intel_report.docx Word report
```

Estimated Token Cost

Development: ~200K–300K tokens / ~$1.40–$2.10 (3–5 sessions to write SKILL.md, develop companion R script, test with different trial scenarios)

Per invocation: ~50K–80K tokens / ~$0.35–$0.55 (skill loading + web search for interim data + R computation + report)

Dollar estimates based on Sonnet API pricing (~$7/M tokens blended). Claude Code Max subscription ($100–$200/mo) offers flat-rate usage.

Tips & Best Practices

1. Start with Plan Mode

Before diving into implementation, use `/plan` to align on the approach. Plan mode lets Claude reason through the task structure, identify dependencies, and propose a step-by-step strategy — without executing anything. This avoids wasted tokens on false starts and ensures you and the AI share the same mental model before code is written.

2. Build Skills to Inject Domain Knowledge

When a task requires specialized expertise (statistical methods, regulatory conventions, package-specific APIs), encode that knowledge in a `SKILL.md` file. Skills ensure the AI applies domain rules consistently across sessions, rather than relying on general training knowledge that may be incomplete or outdated. Include companion scripts for computations that must be exact.

3. Use MCP Servers for External Connectivity

Connect Claude Code to external databases, APIs, and tools via MCP servers. Pre-built plugins handle common integrations (ClinicalTrials.gov, Slack, GitHub), while custom servers let you wrap any API or analysis pipeline. This keeps the AI in the loop for orchestration while delegating data access to reliable, tested endpoints.

4. Control the Deterministic Parts with Code

Not everything should be left to the AI. Use programming code (R scripts, Python functions) for computations that must be reproducible and exact — statistical calculations, data transformations, report formatting. Let the AI handle the non-deterministic parts: interpreting user intent, selecting parameters, generating boilerplate, and adapting to new scenarios.

5. Combine Features for Maximum Impact

The real power of Claude Code comes from **composing** skills, MCP servers, and sub-agents together. A single workflow might load a skill for domain knowledge, query an MCP server for external data, delegate multi-step execution to a sub-agent, and use built-in tools for file I/O and web search. Each component handles what it does best.

> *The key is finding the right balance between deterministic and non-deterministic. Lock down the parts that must be reliable (statistical computations, data pipelines, output formats) with tested code and structured skill definitions. Leave the creative, adaptive parts to the AI (interpreting requests, generating new scripts, handling edge cases). This balance lets Claude Code consistently deliver reliable results while remaining flexible enough to tackle novel tasks.*

"If your plan is to keep doing what you are doing, AI is terrifying.
If your plan is to build something dramatically bigger,
it is the best news you have ever gotten."

-- Garry Tan