Give Claude Code Superpowers

This guide covers everything you need to know about the Superpowers plugin for Claude Code: what it is, how it works, how to install it, and real experiment results comparing it to vanilla Claude Code.

By: Nate Herk

What Is It?

Superpowers is a free, open-source plugin by Jesse Vincent that installs a set of "skills" into Claude Code.

Instead of letting Claude just start coding when you ask it to build something, Superpowers forces it through disciplined phases: **clarify first, design second, plan third, code fourth, verify fifth.**

Think of it like hiring a developer who does proper discovery before touching the keyboard vs. one who just takes your request and starts writing code immediately.

How Does It Work?

The master skill (`using-superpowers`) fires at the start of every conversation and acts as a dispatcher. Before Claude does anything (even before asking you clarifying questions) it checks if any skill applies. There is no config file and zero settings to toggle.

You install the plugin (which takes about 5 seconds), and then you don't have to think about it. It just runs in the background and helps you out with everything.

The 14 Skills and What They Do

The Orchestrator

using-superpowers: The gatekeeper. Fires automatically every conversation. Enforces the rule: "If there's even a 1% chance a skill applies, invoke it before doing anything." You never call this manually.

The Design Phase

brainstorming: Forces Claude to ask questions, propose 2-3 design approaches with tradeoffs, and get your approval before writing any code. Hard gate: no implementation until design is approved. Also includes a visual companion feature that can show mockups and diagrams during the design phase, not just text. It opens a localhost dashboard where you can see and choose between different approaches visually before any code gets written.

The Planning Phase

writing-plans: Creates hyper-detailed implementation plans where every task is 2-5 minutes of work with exact file paths, complete code blocks, and specific CLI commands. No "TBD" or "add appropriate error handling" allowed. Plans get saved to `docs/superpowers/plans/` with dated filenames.

The Execution Phase

executing-plans: Walks through a written plan task by task with safety stops. Stops immediately on blockers rather than guessing.

subagent-driven-development: The autonomous version. Dispatches fresh sub-agents for each task with inline self-review checklists after each task. (Note: v5.0.6 replaced the original 25-minute two-stage subagent review loop with inline checklists after testing showed identical quality scores.)

dispatching-parallel-agents: When you have multiple independent problems (e.g., 3 unrelated test failures), it splits them across parallel agents instead of working sequentially.

The Quality Gates

test-driven-development: Strict TDD. Write the test first, watch it fail, then write the minimum code to make it pass. No exceptions without your explicit approval. If you didn't watch the test fail, you don't know if it tests the right thing.

systematic-debugging: 4-phase debugging: investigate root cause, analyze patterns, form a hypothesis, then implement a fix. If 3+ fixes fail, it questions the architecture itself. Has supporting sub-files for root cause tracing, condition-based waiting, and defense-in-depth validation.

verification-before-completion: Prohibits Claude from saying "Done!" without running the actual verification command and reading the output. Bans words like "should work" or "probably." The iron law: no completion claims without fresh verification evidence.

The Review Phase

requesting-code-review: Dispatches a reviewer sub-agent with proper context after completing work. Can fire after each task, after major features, or before merge.

receiving-code-review: Prevents Claude from blindly agreeing with review feedback. Requires it to verify suggestions against the actual codebase before implementing. No performative agreement allowed. It must push back with technical reasoning when the feedback is wrong.

The Infrastructure

using-git-worktrees: Creates isolated git workspaces so development never touches your main branch directly. Auto-detects project type (Node, Rust, Python, Go) and runs appropriate setup.

finishing-a-development-branch: The exit gate. Runs tests, then presents 4 options: merge locally, push and create PR, keep as-is, or discard. Never proceeds with failing tests.

The Meta Skill

writing-skills: Teaches Claude how to create new Superpowers skills using TDD principles. Write a failing test scenario first, then write the skill to make it pass, then close loopholes. So you can extend the system yourself.

Deprecated aliases (still visible, redirect to current versions): brainstorm -> brainstorming, execute-plan -> executing-plans, write-plan -> writing-plans.

Do They Invoke Automatically?

Yes and no. Here's the breakdown:

Trigger Type	Skills
Always automatic	`using-superpowers` (every conversation)

Auto-chains from other skills	`brainstorming` fires before any creative/build work. `writing-plans` fires after brainstorming. `verification-before-completion` fires before any completion claim. `finishing-a-development-branch` fires after plan execution.
Situational auto-trigger	`systematic-debugging` fires when a bug/failure is encountered. `test-driven-development` fires before any implementation code.
User-chosen	`subagent-driven-development` vs `executing-plans` (you pick which execution mode after a plan is written). `dispatching-parallel-agents` (when multiple independent problems exist).

The key insight: you don't need to memorize when to use each skill. The `using-superpowers` dispatcher handles routing. If you say "build me a login page," it automatically triggers brainstorming first, then planning, then TDD during implementation. You just respond to its questions.

You can also add something like "Hey, make sure you're using any of the superpower skills that might be relevant here" at the end of your prompt for extra insurance.

The Full Chain for Building Something New

```
None
You: "Add a dark mode toggle"
    |
    v
using-superpowers (auto) --> "A skill applies here"
    |
    v
brainstorming --> asks questions, proposes approaches, you
approve a design
    |
    v
writing-plans --> creates detailed step-by-step plan
    |
    v
```

```
using-git-worktrees --> creates isolated workspace
    |
    v
executing-plans OR subagent-driven-development (you choose)
    |
    For each task:
    |-- test-driven-development (write test first)
    |-- verification-before-completion (prove it works)
    |-- requesting-code-review (automated review)
    |
    v
finishing-a-development-branch --> test gate, then
merge/PR/keep/discard
```

The Visual Companion Feature

One of the coolest parts of the brainstorming skill is the visual companion. When you're building something, Superpowers can spin up a localhost dashboard that shows you mockups, diagrams, and design options before any code gets written.

For example, if you ask it to build a website, it might show you three different hero section designs (cinematic full bleed, split screen, centered text with floating video) with pros and cons for each. You click on the one you like, and the agent sees your choice and keeps going.

It can go even deeper, showing different button styles, menu options, card layouts, and color schemes. This is incredibly valuable because it catches misalignment early, before Claude burns tokens building something you didn't actually want.

Keep in mind this feature is still new and can be token intensive, but it's a huge improvement over the old workflow of building something, hating it, and then doing multiple revisions.

Superpowers vs. Ultra Plan

A common question is: what's the difference between Superpowers and Ultra Plan (a built-in Claude Code feature)?

Ultra Plan only helps with the planning phase. Once the plan is written, you're on your own for implementation. Some would argue that the Superpowers brainstorming skill might actually be better than Ultra Plan when it comes to planning, because of the visual companion and the depth of clarifying questions it asks.

But the real difference is that Superpowers stays with you all the way through: from brainstorming, to planning, to execution, to verification, to code review, to finishing. It's end-to-end, not just a planning tool.

How to Install

Installation is simple. From your terminal, install the plugin from the Claude Code marketplace:

```shell
claude plugin install @anthropic/superpowers
```

It will ask if you want this at the per-project level or the user level. **Install it globally on the user level** so you don't have to worry about installing it for each project. It will just work across every Claude Code session.

Link to the GitHub repository: github.com/obra/superpowers

Real Results: Token Usage and Quality Experiments

The Experiment

12 total Claude Code sessions were run. 6 with Superpowers, 6 without. Same prompts, same model (Opus 4.6), and zero human interaction. Tasks were split across simple, medium, and complex difficulty levels, with each run capped at a $2 budget.

Token and Cost Results

Metric	With Superpowers	Without Superpowers
Overall cost savings	~9% cheaper	Baseline
Total tokens	~14% fewer	Baseline

Token variance	Tight and consistent	2-3x the variance
API round trips (turns)	Fewer on average	More on average

The nuance: For simple tasks, Superpowers used slightly more tokens (~8% overhead). That makes sense because when the task is straightforward, you don't need the full brainstorming and planning phases. But for medium and complex tasks, Superpowers was cheaper and used fewer tokens. As complexity increases, the savings trend grows.

Quality Results

A code review evaluated correctness, code structure, test coverage, and error handling. Superpowers scored measurably better across nearly every category on medium-complexity tasks. The only metric where vanilla Claude Code edged ahead was "robustness," which may have been a subjective evaluation.

Key Takeaways

- **Most people assume skills like this have massive token overhead, but that's not the case.** The value is in preventing expensive retries and backtracking.
- **Consistency matters.** Without Superpowers, token usage had 2-3x the variance. With Superpowers, runs were much tighter and more predictable.
- **Code structure and error handling were measurably better** with Superpowers on medium tasks.
- **Domain knowledge and spec compliance were not improved.** That's still on the model itself.
- **For simple tasks, just skip it.** The 8% overhead doesn't add enough value on quick, straightforward requests.
- **This is not conclusive.** 12 runs across three small tasks is directional data, not proof. But it's encouraging.

Important caveat: Superpowers is designed to be human-in-the-loop and iterative because it asks you so many questions. These experiments were fully automated with no human interaction, so results should be taken with a grain of salt. With a human actually answering the clarifying questions, the quality gap would likely be even larger.

What a Beginner Needs to Know

Install it and forget about it. The skills fire automatically. You don't need to memorize anything.

It will slow you down at first. That's the point. The brainstorm + plan phase adds 10-20 minutes before any code is written. It's preventing Claude from writing code you didn't ask for or building the wrong thing.

Answer its questions. When brainstorming asks you to choose between approaches, that's the most valuable part. You're making design decisions before code exists.

It's opinionated. TDD is not optional. Verification is not optional. If you want Claude to "just code it," Superpowers will fight you on that.

Your CLAUDE.md wins. If you put "don't use TDD" in your CLAUDE.md, that overrides the TDD skill. You're always in control. Priority order: your instructions > Superpowers skills > default Claude behavior.

Token overhead is real but manageable. Skills consume context window space (~22k tokens). Long sessions on complex codebases might feel the squeeze, but the trade-off is usually worth it.

Claude sometimes ignores the skills. If Claude skips a step, re-prompting with "check your skills" usually fixes it. Jesse had to add psychological pressure techniques to the skill files to make Claude actually follow them. It's baked in now, but it's not bulletproof.

If Claude asks you to run /using-superpowers at the start of a session, just do it. It's reminding itself to check skills.

What an Advanced User Should Know

Custom skills. You can write your own using the `writing-skills` skill. It uses TDD for skill development itself. Write a failing test scenario, then write the skill.

Lab skills exist. `github.com/obra/superpowers-lab` has 4 experimental skills: `finding-duplicate-functions` (semantic code deduplication via LLM clustering), `mcp-cli` (on-demand MCP server invocation without pre-loading), `using-tmux-for-interactive-commands` (automates interactive CLI tools via tmux), and `windows-vm` (creates headless Windows 11 VMs in Docker with SSH).

Community skills repo. `github.com/obra/superpowers-skills`. Fork and PR new skills here. Note: 94% PR rejection rate on the core repo. Domain-specific skills should live in your own plugins, not submitted to core.

Other frameworks worth knowing about: GSD (~35k stars) constrains context and execution isolation. gstack (~50k stars) constrains decision-making by role. They don't compete with Superpowers. Superpowers constrains the dev process. Power users combine all three.

Security note. Installing plugins is "curl|bash but for LLM agents." You're injecting remote markdown into your AI's context window. Low practical risk for most developers, but enterprise teams should review what they're installing.
