# Superpowers

A Behavioral Constraint System for Claude Code

**by Jesse Vincent**

# What Is Superpowers?

A Claude Code **plugin marketplace** + **skills library** that enforces disciplined software development workflows.

- **14 core skills** covering the full dev lifecycle
- **3 slash commands**: `/brainstorm`, `/write-plan`, `/execute-plan`
- **Subagent architecture** with checks and balances
- Cross-platform: Claude Code, Codex, OpenCode, Windows

> Not just tips — a system that intercepts Claude before it can freestyle.

# Architecture

Two-layer design:

| Layer | Purpose |
| --- | --- |
| **Marketplace** (`superpowers-marketplace`) | Lightweight catalog of plugin git URLs + versions |
| **Plugin** (`superpowers`) | The actual skills, commands, hooks, and lib code |

**Also in the marketplace:**

- `superpowers-chrome` — Browser automation via Chrome DevTools Protocol
- `elements-of-style` — Writing guidance from Strunk's classic

# The Three Iron Laws

Every skill in Superpowers enforces one or more of these:

## 1. TDD

> No production code without a failing test first

## 2. Debugging

> No fixes without root cause investigation first

## 3. Verification

> No completion claims without fresh evidence

# How Skills Work

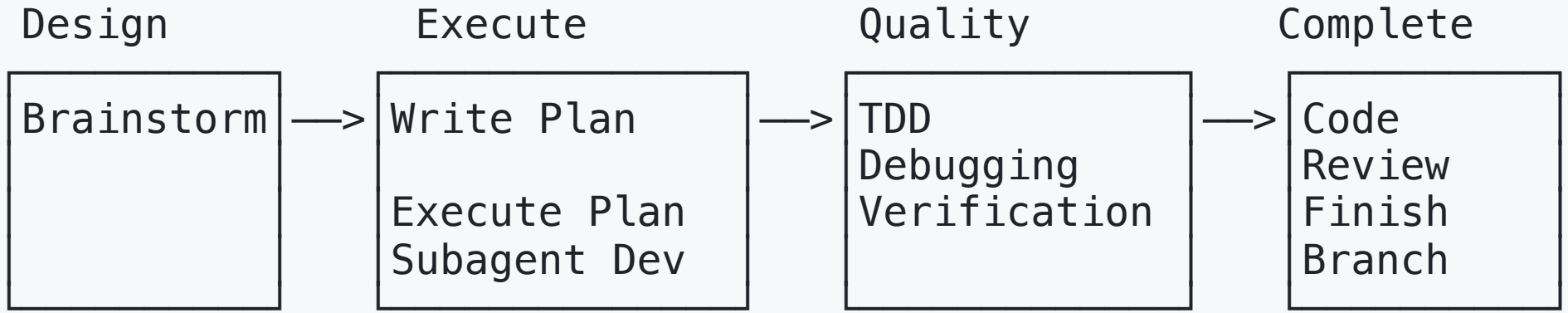
Each skill is a `SKILL.md` with YAML frontmatter + markdown body.

```yaml
name: systematic-debugging
description: "Use when encountering any bug..."
when-to-use:
  - bug reports
  - test failures
  - unexpected behavior
```

When triggered, the skill's content is **injected as a prompt** — shaping Claude's behavior before it responds.

Skills can include supporting files: templates, scripts, tests, examples.

# The Development Pipeline




Skills chain together into a complete workflow:

**Idea** -> **Spec** -> **Plan** -> **Implementation** -> **Verification** -> **Review** -> **Ship**

# Phase 1: Design

## Brainstorming

- Socratic dialogue to refine ideas into specs
- Presents design in 200-300 word chunks with validation checkpoints
- Saves output to `docs/plans/YYYY-MM-DD-<topic>-design.md`

## Writing Plans

- Turns specs into **bite-sized tasks** (2-5 minutes each)
- Each task includes exact file paths, code, and test commands
- Designed to be handed off to subagents

# Phase 2: Execution (Three Approaches)

| Approach | Best For |
|---|---|
| **Executing Plans** | Sequential batch execution with user checkpoints |
| **Subagent-Driven Dev** | Independent tasks with two-stage review per task |
| **Parallel Agents** | 2+ tasks with no shared state |

## Subagent-Driven Development

The most sophisticated approach — dispatches a **fresh agent per task** with:

1. **Implementer** — builds the feature
2. **Spec Reviewer** — verifies it matches requirements (reads code, doesn't trust reports)
3. **Code Quality Reviewer** — checks architecture, testing, production readiness

# Phase 2: Execution Support

## Git Worktrees

- Creates **isolated worktrees** for feature work
- Smart directory selection: `.worktrees` > `worktrees` > ask user > fallback
- Safety checks: verifies `.gitignore`, runs project setup, confirms baseline tests

## Parallel Agents

- Identifies **independent domains** with no shared state
- Fans out work across multiple agents
- Verifies fixes don't conflict on merge

# Phase 3: Quality — TDD

**THE IRON LAW:** No production code without a failing test first.

## RED-GREEN-REFACTOR Cycle

1. **RED** — Write a test that fails
2. **GREEN** — Write minimal code to pass
3. **REFACTOR** — Clean up while tests stay green

Ships with `testing-anti-patterns.md`:

- Don't test mock behavior instead of real behavior
- Don't add test-only methods to production code
- Don't mock without understanding what you're mocking

# Phase 3: Quality — Systematic Debugging

**THE IRON LAW:** No fixes without root cause investigation first.

## Four Phases

1. **Root Cause Investigation** — Trace backwards through the call chain
2. **Pattern Analysis** — Look for systemic issues
3. **Hypothesis Testing** — Form and test theories
4. **Implementation** — Fix only after understanding

## Bundled Tools

- `root-cause-tracing.md` — Never fix just where the error appears

# Phase 3: Quality — Verification

**THE IRON LAW:** No completion claims without fresh evidence.

## The Gate Function

Before saying "done", you must:

1. Run the verification command
2. Read the actual output
3. Confirm it passes

## What It Prevents

- "Tests were passing earlier" (stale results)
- "It should work" (assumption without proof)

# Phase 4: Review & Completion

## Requesting Code Review

- Dispatches the `code-reviewer` subagent with git SHAs
- Structured template: Strengths, Issues (Critical/Important/Minor), Assessment

## Receiving Code Review

- **Explicitly forbids** sycophantic agreement
- `"You're absolutely right!"` is listed as a violation
- Process: read -> understand -> verify against codebase -> evaluate -> respond honestly

## Finishing a Branch

Four options: merge locally, create PR, keep branch, or discard

# The Secret Weapon: Rationalization Prevention

Almost every skill includes a **table of common excuses** with pre-written rebuttals:

| Rationalization | Why It's Wrong |
|---|---|
| "It's a simple change" | Simple changes cause 40% of outages |
| "I already know the cause" | Confirmation bias is the #1 debugging trap |
| "Tests are passing" | When did you last run them? With what data? |
| "Let me just try this quick fix" | Quick fixes without understanding compound |

The skills are written to **anticipate Claude's shortcuts** and explicitly close those loopholes.

# Meta Skills

## Using Superpowers

- The bootstrap skill, loaded every conversation
- Rule: invoke any relevant skill **before** generating any response
- Even a 1% chance = must invoke
- Process skills (TDD, debugging) take priority over implementation skills

## Writing Skills

- TDD applied to documentation itself
- **RED**: Baseline test without the skill
- **GREEN**: Write the skill, verify improvement

# Key Design Insight

Superpowers isn't a collection of suggestions.

It's a **behavioral constraint system** that works by:

1. **Intercepting** — Skills trigger before Claude can freestyle
2. **Injecting process** — Structured prompts enforce methodology
3. **Separating concerns** — Different subagents write vs. review code
4. **Closing loopholes** — Rationalization tables anticipate every shortcut
5. **Requiring evidence** — Claims without proof are violations

> No single Claude invocation both writes and approves its own work.

# Summary

14 skills. 3 iron laws. 1 principle:

**Evidence before claims, always.**

```
/plugin marketplace add obra/superpowers-marketplace
```