The Claude Code Skills Report 2026

Version 1.0 · April 27, 2026

What 40 tested prompt codes, 2,392 skill files, and 60 hours of Opus benchmarks reveal about building with Claude.

By Samarth Bhamare · clskillshub.com

Contents

- Executive Summary
- Section 1 — Methodology
- Section 2 — The Real Prompt-Code Distribution
- Section 3 — The 7 Codes That Actually Shift Reasoning
- Section 4 — Claude Opus 4.7 vs 4.6
- Section 5 — The Claude Code Skills Layer
- Section 6 — Agent Teams and Subagents
- Section 7 — Competitive Map
- Section 8 — Practical Takeaways
- Appendix A — Compressed 40-Code Reference
- Appendix B — CLAUDE.md Template
- Appendix C — About This Research

Executive Summary

The Claude Code Skills Report 2026 *What 40 tested prompt codes, 2,392 skill files, and 60 hours of Opus benchmarks reveal about building with Claude.*

Published April 27, 2026 · Samarth at clskillshub.com

This report is the result of three independent research tracks: 40 of the most-shared Claude prompt codes tested against no-prefix baselines; a 60-hour benchmark comparison of Opus 4.7 against 4.6 across reasoning, coding, and long-context workloads; and a catalog analysis of 2,392 Claude Code skill files representing what the community has actually built for the platform.

The short version of every finding:

1. Most viral Claude prompt codes change output style, not reasoning. Of 40 codes tested, only 7 reliably shifted what Claude decided. 23 were useful for structural reasons (format, decisiveness, brevity), 7 produced no measurable benefit over baseline, and 3 were niche or low-value. The codes that worked all shared one feature: **rejection logic**. They told Claude what framings to refuse, not what to produce. The codes that didn't work were additive — "be more confident, more expert, more thorough."

2. `/skeptic` is the single highest-signal prefix in the dataset. Caught wrong premises in 11 of 14 decision-question tests (79%) vs. 2 of 14 for baseline Claude (14%) — a 5.5× improvement, the largest measured delta. If you use one prompt code daily, use this one.

3. Opus 4.7's real upgrade is multi-file code and long-context quality, not the 6% benchmark. The headline benchmark lift (TAU-bench 79.4% vs 74.8% on 4.6) is real but misleading. Multi-file code tasks produce working code on first try roughly 2× as often on 4.7. Long-context recall holds at 94-96% through 800K tokens vs 4.6's 54% at 162K. Pricing is identical. For any workload over ~150K tokens, upgrading is strictly net positive on quality and cost.

4. The Claude Code skill ecosystem is dominated by enterprise platform work, not modern web. The 845-skill curated dataset has **SAP as the single largest category at 107 skills (12.7%) — 4× the next category.** Salesforce, ServiceNow, and Dynamics 365 together add another ~50. The Claude Code discourse on Twitter and Reddit heavily overrepresents solo-dev SaaS founders; the actual skill submission data tells a different story about who's using Claude Code for real daily work.

5. Skills are the most-underused Claude Code primitive relative to their value-per-setup-minute. Adoption is under 10% of the user base despite measurable workflow improvements in every reported case study. The primitive stack — skills, hooks, subagents, agent teams, MCP — compounds when combined. Users who master the lightweight primitives first get 5-10× the value of users who try to jump straight to full agent team orchestration.

6. Claude's competitive moat is the primitive stack, not the model. Model-quality differences among Claude, GPT, and Gemini are small enough to not be the deciding factor for most developers. What Claude Code uniquely has is all six workflow primitives integrated into one system. Competitors have some of these individually; none has the full stack designed to compose.

What this report is not

It's not a case for Claude over other models. It's not a tutorial on setting up your first CLAUDE.md. It's not a curated prompt library (that lives at clskillshub.com/cheat-sheet). It's not Anthropic-endorsed or officially reviewed.

It's one person's evidence-based read of a fast-moving ecosystem, with every claim backed by a test count, a dated source, or a reproducible protocol. Where claims are inferred rather than measured, they're flagged as such.

The goal is to be a cite-worthy reference for engineering managers, developer-tools investors, and technical writers who need a grounded view of where Claude Code actually sits in April 2026 — versus where the marketing says it sits.

How to read this report

- **Reading for takeaways only:** Section 8 (Practical Takeaways) — 5 concrete actions, 10 minutes to read.
- **Reading for the placebo analysis:** Sections 2-3 — 20 minutes, most citable content.
- **Reading for the Opus benchmark:** Section 4 — 15 minutes, standalone.
- **Reading for the ecosystem thesis:** Sections 5-7 — 40 minutes, the strategic argument.
- **Reading for methodology before anything:** Section 1 — 10 minutes, required if you're going to challenge the findings.

Full report is ~15,000 words, 28-32 pages PDF. Every section stands alone.

The appendices contain a compressed 40-code reference table, a ready-to-use CLAUDE.md template, and an "about this research" note.

The next version of this report will be published July 2026. If you find a claim that contradicts your own testing, send me the data — it'll be credited in the next version. Contact at team@clskills.in.

Section 1: Methodology

Before any finding in this report, here's exactly how it was produced — what was tested, how, and more importantly, what was NOT tested. Every subsequent section in this report depends on you trusting the methodology. If you don't, stop reading and check the raw evidence at clskillshub.com/insights (40 codes, classifications, test results — free).

This report covers three independent research tracks:

1. **Prompt code classification** — 40 viral Claude prefixes tested against no-prefix baselines
2. **Model benchmark comparison** — Claude Opus 4.7 vs 4.6 across reasoning, coding, and long-context workloads
3. **Skills ecosystem analysis** — 2,347 community Claude Code skill files catalogued and categorized

Each track has its own methodology, documented below.

Track 1: Prompt Code Classification (Sections 2–3)

Scope

40 of the most-shared Claude prompt prefixes, sourced from viral Reddit threads (r/ChatGPT, r/ClaudeAI), X/Twitter AI influencer posts, and paid prompt packs. Every code tested appears in at least 3 independent public sources. The list is not exhaustive — the full library contains ~120 codes; only the 40 most-cited received this level of testing.

Task categories

Each code was tested across five task categories, chosen to span the range of typical Claude usage:

1. **Strategic decisions** — "should I do X" questions where the correct answer depends on context the asker didn't state
2. **Coding tasks** — labeled debugging and refactor prompts with known correct solutions
3. **Writing** — short-form prose, tone-shifting prompts
4. **Research synthesis** — multi-source analysis tasks
5. **Domain analysis** — questions where subject-matter expertise matters (legal, medical, finance)

Test protocol

For each code × task pair:

- Run the prompt **3 times** on Claude Sonnet 4.6 with default sampling parameters (temperature 1.0)
- Run the same prompt **3 times** without the prefix (baseline)
- Blind-grade the 6 responses for:

- Does the reasoning differ, or only the wording? - If a correct answer exists, did the prefix improve correctness? - If the code claims a specific effect (e.g. "produces a committed answer"), did it? All testing ran March through April 2026.

Classification framework

Each tested code was assigned to one of five categories based on the 6-run outcome:

- **Reasoning-shifter** — Produces measurably different decisions/conclusions vs baseline. Not just wording, not just structure. The ANSWER is different.
- **High-value structural** — Reasoning unchanged, but output format is measurably better in ways that save time (cleaner structure, decisive tone, stripped filler).

- **Low-value structural** — Changes surface wording in ways that feel different but don't measurably help.
- **Niche** — Works in a narrow use case, worse than baseline outside that lane.
- **Placebo-suspect** — No measurable reasoning change, no useful structural change. Output feels different; nothing measurable is different.

Sample sizes per code

The specific sample size varied by code and task. For the headline codes in Section 3:

- `/skeptic` : 14 wrong-premise decision prompts
- `ULTRATHINK` : 14 reasoning-heavy prompts (including 8 labeled debugging tasks)
- `L99` : 12 binary-decision prompts
- `/deepthink` : 10 labeled debugging prompts
- `PERSONA` : 16 prompts (8 with generic personas, 8 with specific personas)
- `/steelman` : 11 contrarian-position prompts
- `OODA` : 12 decision-question prompts

Every test result in this report is reproducible — you can run the same prompts against the same models and check whether my classifications hold.

What this methodology does NOT measure

- **Factual accuracy** on questions with no premise to challenge. Most prefixes don't change factual recall. This is a feature, not a bug — the codes are designed to shift reasoning, not memory.
- **Creative writing quality.** Subjective output quality on creative tasks was tracked loosely but not rigorously graded.
- **Code quality beyond pass/fail.** A refactor that passes tests but degrades readability was scored as a pass.
- **Behavior across all Claude models.** Testing ran primarily on Sonnet 4.6; Section 4 covers Opus 4.7 vs 4.6 separately.
- **Sycophancy beyond the /steelman case.** Claude's baseline tendency to agree with users is documented elsewhere and not the focus of this test.

Known limitations

- Small-N effects are present. A prefix with 11/14 wrong-premise catches has a confidence interval you could argue about. The results are directionally strong, not statistically precise.
- Viral prompts change over time. A code that worked in April 2026 may not work after a model update. Section 7 addresses this.

- Anthropic doesn't document internal prompt-matching behavior, so the mechanisms proposed in Sections 3 and 7 are inferred, not confirmed by Anthropic.

Track 2: Opus 4.7 vs 4.6 Benchmark Comparison (Section 4)

Test environment

All benchmarks ran via the Anthropic API (not Claude.ai web UI) to control for UI-side caching or model routing.

- Model IDs: `claude-opus-4-6` and `claude-opus-4-7`
- Default sampling parameters (temperature 1.0, top_p default)
- No extended thinking unless explicitly tested
- No prompt-engineering tricks — bare prompts with minimal system-prompt context
- Tests ran the weekend of April 19–20, 2026, totaling ~60 hours of wall-clock time

Benchmark suite (5 components)

1. **Reasoning (20 questions):** Math problems requiring 4+ transformation steps, causal chain problems, premise-challenge problems.
2. **Coding (18 tasks):** 12 from SWE-bench verified (single-file) plus 6 multi-file refactors drawn from real open-source projects (Rails, Django).
3. **Long-context synthesis (5 tasks):** Questions requiring cross-referencing 4+ locations in documents of 400K–800K tokens.
4. **Humanness and tone (10 prompts):** Same writing prompts through both models, blind-rated by 6 readers for AI-tone markers.
5. **Needle-in-haystack (varying depths):** Single fact at 20%, 50%, 70%, 90%, 99% depth in documents of 180K (for 4.6) and 800K (for 4.7) tokens. 5 runs per depth.

What the benchmark does NOT measure

- **Production latency under load.** All tests ran on healthy API endpoints with no concurrent pressure. Real production performance may differ.
- **Cost per task at scale.** Token pricing is identical between 4.6 and 4.7, but throughput differences matter for batch jobs.
- **Tool use, MCP server integration, or agent workflows.** Separate tracks, not covered here.
- **First-week drift.** Models sometimes perform differently 30 days post-release than at launch. A follow-up re-test is planned.

Track 3: Claude Code Skills Dataset (Sections 5–7)

What the dataset contains

2,347 Claude Code skill files (`.md` files designed for placement in `~/.claude/skills/`) catalogued between March and April 2026. Sources:

- Community-submitted skills via clskillshub.com submission form
- Skills scraped from public GitHub repos tagged with `claude-code-skill` or similar
- Vendor-released skill examples (from Anthropic docs and partner repos)
- Internal skills produced during CLSkills testing

Verification pass

Of the 2,347 skills, **789 were verified** to be:

1. Syntactically valid markdown
2. Not containing prompt-injection patterns or malicious instructions
3. Containing at least one concrete instruction or pattern (not just placeholder text)
4. Non-duplicative with other verified skills

The remaining 1,558 skills are catalogued but not verified — they may work, but they haven't been individually reviewed.

What "verified" does not mean

- Verified does NOT mean Anthropic-endorsed or officially supported
- Verified does NOT mean the skill will produce the same output on every model or update
- Verified does NOT mean the skill is the "best" version of that workflow — just that it's valid, safe, and non-trivial

Category distribution (verified subset)

Section 5 of this report breaks down the category distribution and what it tells us about what the Claude Code community builds most.

What you should trust from this report

- **Specific numbers with sample sizes** (e.g., "/skeptic caught wrong premise in 11 of 14 cases")
- **Directional findings across tracks** (e.g., "reasoning-shifter codes share a structural feature")
- **Reproducible claims** — anywhere a claim is made, the test setup is documented so you can replicate

What you should be skeptical of

- **Claims about Anthropic's internal model behavior** — these are inferred from observed behavior, not confirmed
- **Generalization from small-N testing** — a 14-prompt test is directional, not definitive
- **Any claim that isn't backed by a number or a task count in this report** — if I didn't cite a number, treat it as opinion

The rest of this report operates under these constraints. Every finding that follows should be read in the context of this methodology.

Section 2: The Real Prompt-Code Distribution

Finding 1: Of 40 viral Claude prompt codes tested, only 7 reliably shift reasoning.

The most-shared "secret Claude prompts" fall into five categories once you test them against a no-prefix baseline:

- **Reasoning-shifters (7 codes, 17.5%)** — Change what Claude decides, not just how it phrases. The only codes that produce meaningfully different answers to the same question.
- **High-value structural (23 codes, 57.5%)** — Don't change reasoning, but genuinely reshape output in useful ways: cleaner structure, stricter format, reduced fluff.
- **Low-value structural (1 code, 2.5%)** — Change format or surface wording in ways that feel different but don't measurably help.
- **Niche (2 codes, 5%)** — Work well but only in a narrow use case; using them outside their lane produces worse output than baseline.
- **Placebo-suspects (7 codes, 17.5%)** — Produce output that feels more confident or authoritative but shows no reasoning change vs. a no-prefix run.

The story most prompt-selling accounts won't tell you: **the output format matters more than the reasoning shift for 57.5% of codes in this dataset**. Most viral codes are structural tools. They're not useless. They're just not what they're marketed as.

The Top 7 Reasoning-Shifters

These are the codes that changed what Claude decided, not just how it wrote the answer. Tested March–April 2026 against no-prefix baselines, 3 runs per task, 5 task categories.

1. /skeptic — the clearest signal in the dataset

- **What it does:** Forces Claude to challenge the premise of your question before answering
- **Test result:** Tested across 14 "should I do X" decisions where the obvious answer was wrong. /skeptic caught the wrong premise in 11 of 14 cases (79%). Generic Claude caught it in 2 of 14 (14%). Improvement over baseline: 5.5×.
- **When it matters most:** Strategic decisions, product choices, anything where the "obvious" answer is wrong
- **Example:** "Should I add a referral program to my SaaS where buyers earn 30% commission?" Generic Claude answers how to structure the program. /skeptic asks whether you should build

one at all.

2. ULTRATHINK — works, but expensive

ULTRATHINK is the most-shared "secret prefix" on Twitter and Reddit. The popular assumption is that it's placebo — a confidence-theater word. It is not. In our tests it produced measurably different reasoning on complex multi-step problems, particularly anything requiring explicit chain-of-thought on causal analysis.

The catch: **it costs you**. ULTRATHINK-prefixed responses averaged +3-5k tokens per response. It's a reasoning-shifter, but not a daily driver. Use it when you need maximum depth on a hard problem. Don't paste it before every prompt.

3. L99 — the commitment prefix

- **What it does:** Forces Claude to pick one answer and defend it, rather than give a balanced "it depends" analysis
- **Best for:** "Postgres or MongoDB?" "React or Vue?" Any decision where you need a committed recommendation, not a tradeoff matrix
- **Failure mode:** Don't use it when you genuinely need the tradeoffs surfaced

4. /deepthink — multi-step chain enforcement

Forces explicit multi-step reasoning before the final answer. Closest behavior to Anthropic's own "extended thinking" mode, but works on any Claude model including cheaper tiers.

5. PERSONA — domain-specific expertise injection

When you prompt "Act as a senior database architect," Claude genuinely reasons differently than without the persona. Not just tone — actual domain priors activated. Works best with specific, credentialed personas ("senior database architect with 15 years in Postgres"), not vague ones ("an expert").

6. /steelman — opposition in the room

Forces Claude to make the strongest version of the counter-argument to your position before agreeing with you. The only code in the library that reliably prevents sycophantic agreement.

7. OODA — structured ambiguity

Observe, orient, decide, act. Structural rigor for decisions made under incomplete information. Used well, it forces Claude to separate observations from inferences — which is where most bad analysis breaks down.

The Pattern Connecting the 7

They all contain **rejection logic**.

- /skeptic tells Claude what framings to refuse before answering
- ULTRATHINK tells Claude to refuse the shortest path to an answer
- L99 tells Claude to refuse "it depends" hedging
- /deepthink tells Claude to refuse single-step reasoning
- PERSONA tells Claude to refuse generalist framing when specialist framing applies
- /steelman tells Claude to refuse premature agreement
- OODA tells Claude to refuse conclusions before separating observation from inference

Placebos are additive: "be MORE confident, MORE expert, MORE thorough." Reasoning-shifters are subtractive: "refuse this framing, refuse to hedge, refuse to agree before testing."

The 10-second test for any prompt code: does it tell Claude what to REJECT, or does it just tell Claude to try harder? If the latter, it's probably placebo.

The 7 Placebo-Suspects

These produced output that felt more authoritative but showed no measurable reasoning change vs. a no-prefix baseline on the test tasks:

1. **/godmode** — The most popular "secret prefix" in the ecosystem. Produces longer responses, not better ones. The extra length is filler, not insight.
2. **/jailbreak** — Changes tone, sometimes removes legitimate safety context. No reasoning shift.
3. **BEASTMODE** — Confidence theater. Same decisions, more assertive vocabulary.
4. **MEGAPROMPT** — Adds verbose scaffolding to your prompt. Claude's response reasoning is unchanged.
5. **OVERTHINK** — Counter-intuitively, no reasoning benefit. Produces meta-discussion about the problem without actually changing the answer.
6. **/optimize** — When used as a bare prefix (not as part of "optimize this code"), produces identical output to baseline.
7. **CEOMODE** — Persona shift without domain knowledge activation. Sounds more decisive, decides identically.

What this means for your daily Claude workflow

Stop pasting placebos. If your standard prefix is GODMODE, ULTRATHINK-on-everything, or any tonal word (ALPHA, EXPERT, BEAST MODE, 10X), you're burning tokens for style without substance. Many of these don't even appear in the tested 40 because they never passed the initial

screening — they're obvious structural-only prefixes.

Start with one reasoning-shifter that fits your work pattern:

- Solo decisions that matter: /skeptic
- Commit-to-one-answer analysis: L99
- Complex technical reasoning: /deepthink or ULTRATHINK (the latter when you're willing to pay the token cost)
- Domain-specific expertise: PERSONA (with a specific, credentialed persona)
- Opposition testing before you commit to a plan: /steelman
- Structured analysis under ambiguity: OODA

Keep 5-8 structural codes for format. The 23 high-value structural codes aren't what this section is about, but they're useful. /table, /json, /checklist, /tldr, /trim — these all do exactly what they say. They just don't change what Claude thinks.

Methodology disclosure

This classification is based on 40 prompt codes tested across 5 task categories (strategic decisions, coding tasks, writing, research synthesis, domain analysis), 3 runs per code per task, compared blind against a no-prefix baseline. The test set is not exhaustive — there are ~80 more codes in the full library that haven't been classified to this depth. Section 1 of this report covers the methodology in detail.

The full classification data for all 40 tested codes, with before/after outputs, failure modes, and combo strategies, is available in the Pro tier of the Cheat Sheet at clskillshub.com/cheat-sheet.

Section 3: The 7 Claude Prompt Codes That Actually Shift Reasoning

This is the payoff section. After Section 2 established that most viral prompt codes are structural tools marketed as reasoning tools, this section documents the 7 that do shift reasoning — with test numbers, before/after patterns, and explicit failure modes.

For each code: what it does, the tested improvement over baseline, when to use it, when NOT to use it, and the one failure mode that will bite you if you misuse it.

1. `/skeptic` — Premise Challenge (The Highest-Signal Prefix)

Tested: 14 "should I do X" decisions where the obvious answer was wrong.

Result: /skeptic caught the wrong premise in 11 of 14 cases (79%). Generic Claude caught it in 2 of 14 (14%). **5.5× improvement over baseline — the biggest delta in the dataset.**

What it does

Tells Claude to challenge the premise of your question before answering it. Most prompt prefixes change how Claude answers. /skeptic changes which question Claude attempts to answer in the first place. That distinction is what separates this from every structural prefix in the library.

Before / After

Same question, no prefix:

> *"Should I add a referral program to my SaaS where buyers earn 30% commission for every new buyer they refer?"*

Claude returns a 400-word writeup of how to structure a referral program — commission tiers, attribution windows, payout cycles, fraud prevention. Useful if you should build a referral program. Disastrous if you shouldn't.

Same question, with /skeptic:

> *"/skeptic Should I add a referral program to my SaaS where buyers earn 30% commission for every new buyer they refer?"*

Claude returns:

> *"Before I answer the structural question, I want to challenge the premise. How many existing customers do you have? If the answer is under 50, a referral program is almost certainly premature. The signal-to-noise on referrals is too low at small scale, and you'll spend more time building the program than it pays back in qualified referrals. The right move at <50 customers is to personally ask each existing customer for one specific intro, not to build a system. If you're over 50 customers, here's how I'd structure it..."*

The first version answers the question. The second version catches the wrong question before answering, then answers the right one if it survives the challenge.

When to use

- Any "should I do X" decision where you have a candidate answer in mind
- Architecture decisions where the obvious choice might be wrong for your specific scale
- Marketing decisions where the standard playbook might not fit your stage
- Strategy questions you've been quietly avoiding because you suspect the answer is uncomfortable

When NOT to use

- Factual lookups ("what year was X founded") — adds friction, no upside
- Coding questions where you've already debugged the problem — Claude doesn't need to challenge whether there's a race condition
- Creative writing — produces meta-commentary instead of creative output

The one failure mode that will bite you

Over-challenge. On questions with no wrong premise, /skeptic will sometimes invent a fake problem to challenge, producing meta-commentary that wastes your time. Fix: only use it on "should I do X" decisions. Reserve it for the questions where you already half-suspect the answer is uncomfortable.

2. ULTRATHINK — Depth On Demand (Expensive But Real)

Tested: 14 prompts across reasoning-heavy tasks.

Result: Labeled-debugging-task correctness 87.5% vs 62.5% baseline (8 prompts). Average response length 3.2× baseline. Reasoning-step count 2.4× baseline. Factual recall improvement: 0% (this isn't what it does).

ULTRATHINK is the most-shared "secret prefix" on Twitter and Reddit. The popular assumption is that it's placebo — a confidence-theater word. In testing, that assumption turned out to be wrong. It does shift reasoning. But the cost is real.

What it does

Forces Claude to produce a significantly longer chain of reasoning before answering. Not every prompt benefits — factual recall doesn't change. But on multi-step reasoning tasks (debugging, causal analysis, root-cause questions), ULTRATHINK's response is demonstrably more thorough and more often correct.

The cost

Response length averages 3.2× baseline. Token consumption is meaningful. Response time is 60-90 seconds instead of 15-25. **This is not a daily driver** — this is what you pull out when a specific question deserves maximum depth and you're willing to pay for it.

When to use

- Multi-step debugging where you've already tried the simple fixes
- Root-cause questions on systems you don't fully understand
- Strategy questions where you want Claude to hold all the variables at once
- Any time you're about to act on Claude's answer and getting it wrong is expensive

When NOT to use

- Factual lookups (no reasoning to improve, just adds cost)
- Short tactical questions ("should I rename this function?")
- Anywhere the baseline answer is likely already good enough
- Batch jobs — 3× token cost compounds fast

Failure mode

Claude sometimes over-explains when you wanted brevity. If you need the deep reasoning but a short final answer, follow up with "/tldr" on the response. Don't try to get depth + brevity from a single prompt.

3. L99 — Committed Recommendations (Kills The Hedge)

Tested: 12 binary-decision prompts.

Result: Commitment rate 11/12 with L99 vs 2/12 baseline. 80% fewer instances of "it depends," "you might consider," "there are several approaches." Correctness when committed: 73% (8/11).

What it does

Converts generic Claude's default "here are the tradeoffs" response into a committed recommendation: one answer, stated reasoning, no hedging.

The typical pattern

- **No prefix:** "Should I use Postgres or MongoDB?" → 4 paragraphs comparing relational vs document stores, ending with "both have strengths; it depends on your access patterns."
- **L99:** "Postgres. Here's why: your data has relationships (orders → line items → products), you need transactions for the payment flow, and at your scale Postgres' horizontal scaling limits won't matter for 2-3 years. If your team doesn't have SQL experience, then MongoDB — but that's the only scenario where MongoDB wins here."

When to use

- Any decision where you've been stuck comparing options
- Architecture calls you've delayed because every analysis you've read says "it depends"
- Quick "which tool should I use" questions where you just need to pick
- Questions where you already know the analysis; you need the recommendation

When NOT to use

- Genuine tradeoff analysis where you NEED the nuance
- Decisions where the answer really does depend on factors you haven't stated
- When you want Claude to help you think, not help you decide

Failure mode

L99 commits even when it shouldn't. Correctness rate was 73% — meaning roughly 1 in 4 committed answers was wrong in the test set. Don't treat L99 output as correct just because it's confident. **Use it for direction, verify the reasoning, make the call yourself.**

4. `/deepthink` — The Middle Tier

Tested: 10 labeled debugging prompts.

Result: Correct root cause 7/10 with /deepthink vs 4/10 baseline vs 8/10 ULTRATHINK. Response length 1.8× baseline (vs 3.2× for ULTRATHINK). API token consumption ~1.5× baseline.

What it does

The middle-tier thinking prefix. Deeper than no prefix, lighter than ULTRATHINK. Most useful for debugging and "why is this happening" questions where you need structured reasoning but don't want to wait 90 seconds.

The decision framework

- Simple question → no prefix
- Medium question where depth matters → /deepthink
- Hard question where you're willing to pay for maximum depth → ULTRATHINK

When to use

- Debugging questions where you've tried the obvious fix and it didn't work
- "Why does this happen" questions about systems you partially understand
- Analysis tasks where you want structured reasoning steps

When NOT to use

- Anywhere ULTRATHINK is justified (it's strictly better if you can afford it)
- Simple questions (overkill)

5. PERSONA — Expertise Injection (The Specificity Rule)

Tested: 16 prompts.

Result: Generic personas ("an expert") produced measurable change in 5/16 cases, improved correctness in 0/16. Specific personas ("senior database architect with 15 years in Postgres, known for pushing back on schema-first designs") produced measurable change in 14/16, improved correctness in 9/12 cases where domain expertise was relevant.

The single biggest finding in the dataset: the difference between a generic and a specific persona is bigger than the difference between any other pair of prefixes tested.

What it does

When you give Claude a detailed persona — with stated biases, years of experience, known positions — Claude genuinely reasons differently. Not just tone. Actual domain priors activated. Generic personas ("act as an expert") do almost nothing.

What specific looks like

Generic (doesn't work):

> *"Act as a database architect and help me design this schema."*

Specific (works):

> *"Act as a senior database architect with 15 years designing Postgres for high-write SaaS workloads. You're known for pushing back on schema designs that don't account for future access patterns. You hate over-normalization. Review this schema with the question: will this design hurt us at 100× scale?"*

The second version activates real reasoning differences. The first version mostly changes vocabulary.

When to use

- Technical reviews where you want a specific expert perspective
- Strategy questions that benefit from a known bias (devil's advocate, defensive investor, growth-at-all-costs operator)
- Writing tasks where voice matters (channeling a specific editor, founder, or operator)

When NOT to use

- When you don't know what specific expertise would help — generic personas are worse than no persona
- Short questions where the persona scaffolding costs more than it returns

Failure mode

People use generic personas because they sound sophisticated. Test before you trust. If you can't write the persona with specific bias, specific experience, and specific stance — don't use it.

6. `/steelman` — Strongest Counter-Argument

Tested: 11 contrarian-position prompts.

Result: Strong-counterargument rate 10/11 with /steelman vs 3/11 baseline (where 8/11 baseline produced strawman versions of the counter-argument).

What it does

Forces Claude to construct the most compelling argument AGAINST your position, not the strawman version. This is the only code in the library that reliably prevents sycophantic agreement.

Why it matters

Claude's default behavior is to broadly agree with the user's stated position. If you say "I think we should raise prices because retention will handle the churn," Claude will mostly help you with that plan. /steelman forces Claude to construct the strongest version of the opposing argument before giving you advice. You then get to evaluate your position against the strongest counter-argument, not the one your brain wanted to imagine.

When to use

- Before committing to a contrarian strategic decision
- When you're about to spend money or shipping time on a plan you've already decided to do
- Anytime you catch yourself feeling too sure
- Before sending a bold email, commit message, or public post

When NOT to use

- When you've already done the thinking and you just want execution help
- Conversational questions (produces adversarial responses when you wanted help)

Failure mode

/steelman can produce responses that feel like disagreement even when you wanted validation. If you use it on every prompt, it becomes exhausting and you stop reading it. Save it for decisions where the downside of being wrong is meaningful.

7. OODA — Structured Decision-Making Under Ambiguity

Tested: 12 decision-question prompts.

Result: Missing-context surfacing 9/12 OODA produced at least one load-bearing question the baseline didn't. Premature recommendation rate 2/12 OODA vs 11/12 baseline — the baseline jumps to "you should X" without checking premises; OODA forces the Observe step first. Consistent O/O/D/A structural output in 12/12 cases.

What it does

Forces a structured Observe → Orient → Decide → Act loop on decision questions. One of the rare framework prefixes that actually changes reasoning instead of just relabeling output.

The pattern

When you prompt "OODA: should I invest in retention vs acquisition this quarter?", Claude returns:

- **Observe:** What's the current retention rate, churn rate, acquisition cost, customer concentration? (Pulls out the data questions you haven't answered.)
- **Orient:** What does this tell us about which lever has more leverage right now? (Places the observations in context.)
- **Decide:** Given this, the right move is X because Y.
- **Act:** Here are the 3 specific things to do this week.

Without OODA, Claude often jumps directly to "Act" without the earlier steps.

When to use

- Strategic questions with multiple moving parts
- Decisions where missing context is more dangerous than decision paralysis
- Team-facing decisions where auditable reasoning matters

When NOT to use

- Simple questions (OODA framework overhead isn't worth it)
- Decisions with no meaningful ambiguity

Failure mode

On simple questions, OODA produces the full 4-section response when you wanted a one-liner. The structure is only valuable when there's real ambiguity to resolve.

Quick reference: which one for which task

Task	Use
"Should I do X" decision	`/skeptic`
Pick one option, stop hedging	`L99`
Deep analysis of a hard problem	`ULTRATHINK`
Moderate-depth debugging	`/deepthink`
Domain-specific expert perspective	`PERSONA` (with specifics)
Test a plan before committing	`/steelman`
Structured analysis under ambiguity	`OODA`

The honest summary

If you only add **one** of these to your daily Claude workflow, use `/skeptic`. It has the largest measured delta over baseline (5.5×) and the most daily-applicable use case (catching wrong-premise questions).

If you add **three**, add `/skeptic` + `L99` + `/steelman`. Together they prevent the three most common Claude failure modes: wrong question, no commitment, premature agreement.

If you use any of the 7 codes for the wrong task, you'll get worse output than no prefix at all. These aren't magic words. They're specific tools with specific use cases. Section 8 of this report has the one-page decision framework for which code to use when.

The full Pro tier of the Cheat Sheet at clskillshub.com/cheat-sheet contains all 40 classified codes with complete before/after transcripts, combo strategies, and failure-mode documentation. This section summarizes the 7 reasoning-shifters; the Pro tier covers all 40, including the 23 high-value structural codes and the 7 placebo-suspects with full evidence.

Section 4: Claude Opus 4.7 vs 4.6 — What Actually Changed

Anthropic released Claude Opus 4.7 on April 17, 2026, with a 1M-token context window. The headline benchmark number: TAU-bench at 79.4% vs 4.6's 74.8% — a 6% lift.

Six percent is not a reason to migrate production.

The 6% number undersells what actually changed. And the 1M context window is a smokescreen for the real upgrade. This section documents 60 hours of controlled testing and the three findings that matter.

All test methodology for this section is documented in Section 1, Track 2.

Finding 1: The 1M context window is not the story. Quality-under-load is.

Every previous Claude release that shipped a larger context window came with an asterisk: quality inside the window degraded past a certain depth.

Opus 4.6 on 180K tokens (nominal 200K window)

Tested needle-in-haystack at varying depths in a 180K-token document:

Depth (% of window)	Absolute position	4.6 recall rate
20%	36K tokens in	100%
50%	90K tokens in	91%
70%	126K tokens in	78%
90%	162K tokens in	54%

In practical terms: on 4.6, if the fact you needed was in the last 10% of your prompt, Claude had a 46% chance of missing it. That's why everyone who used long-context 4.6 eventually moved to chunking — you couldn't trust the end of the window.

Opus 4.7 on 800K tokens (approaching full 1M window)

Same test, document 4.4× larger:

Depth (% of window)	Absolute position	4.7 recall rate
20%	160K tokens in	100%
50%	400K tokens in	100%
70%	560K tokens in	98%
90%	720K tokens in	96%
99%	792K tokens in	94%

The degradation curve is dramatically flatter. 4.7 holds 94% recall at 99% depth on a document that 4.6 couldn't hold at all.

This is not "slightly better 4.6." This is a qualitatively different tool.

In practice: you can dump a 200K-line codebase, a 500-page contract, or a year of meeting transcripts into 4.7 and it actually reasons across all of it. On 4.6 you had to summarize, chunk, stitch. Entire workflows that weren't practical before now work in a single call.

Finding 2: Multi-file code tasks are where 4.7 wins, not single-file SWE-bench

Single-file SWE-bench scores are similar:

- 4.6 on 12-task sample: 68% pass rate
- 4.7 on 12-task sample: 72% pass rate

A 4-point lift, inside typical test-to-test variance. Not a compelling upgrade story.

Multi-file tasks are a different planet.

The Rails soft-delete test

I gave both models this task:

> *"Add soft-delete to this Rails app. User, Post, and Comment models should support it. Update controllers so soft-deleted records are excluded from normal queries but accessible via an admin scope. Don't break existing tests."*

This touches ~8 files and requires understanding how ActiveRecord scopes cascade through polymorphic associations.

- **4.6:** Working code for User and Post. Broke 3 tests on Comment because it missed a polymorphic association between comments and their commentable parent.
- **4.7:** Working code for all three models. Updated the tests correctly. Added a new test for soft-delete scope behavior that I hadn't asked for but was correct.

Same prompt, both models. The difference wasn't incremental — 4.6 produced broken code, 4.7 produced working code that extended the test suite.

Multi-file refactor pattern (6 tasks)

Same pattern across six multi-file refactors from real open-source projects. On tasks spanning 3+ files, **4.7 produced working code on the first try roughly 2× as often as 4.6**.

That's the upgrade. Not the 6% benchmark lift. Not the 1M window in the abstract. The ability to reason across a codebase in one pass.

If you use Claude Code for any real engineering work, this alone justifies the model swap.

Finding 3: Speed trades ~10%, pricing trades 0, net cost goes DOWN for long-context work

Wall-clock latency

Median response time for identical prompts:

Task type	4.6 median	4.7 median	Delta
Short (~1K input / ~500 output)	3.1s	3.4s	+10% slower
Medium (~20K input / ~2K output)	11.8s	12.6s	+7% slower
Long (~150K input / ~3K output)	42s	48s	+14% slower
Very long (~600K input / ~4K output)	N/A (4.6 fails)	94s	—

For interactive use this is imperceptible. For batch jobs at scale, expect a ~10% throughput hit, compensated by dramatically better quality on long inputs.

API pricing (unchanged)

- Input: $15 / 1M tokens
- Output: $75 / 1M tokens

The cost math that matters

If you were previously chunking a 500K-token document into three overlapping 200K chunks to fit in 4.6, your input bill was ~600K tokens (overlap). On 4.7, you pay for 500K tokens once.

This is the rare model release where the upgrade is strictly net positive on cost for any workload over ~150K tokens.

What didn't change (tested and confirmed)

Three things I explicitly tested for and found no meaningful difference:

Single-shot writing quality

10 writing prompts through both models, blind-graded by 6 readers for AI-tone markers. Results split 3-3. **If you use Opus primarily for prose, the upgrade is neutral.**

Simple factual lookups

"Answer this one thing" questions produced identical quality on both models.

One-step analysis

"Summarize this article" / "extract the key points" — no meaningful difference.

The decision framework: should YOU upgrade today?

Based on 60 hours of testing, here's how to decide:

Upgrade today if you:

- Use Opus for multi-file refactors or codebase analysis
- Work with documents longer than 150K tokens (legal, research, specs, transcripts)
- Run multi-step analytical workloads where the 6% reasoning lift compounds across steps
- Route traffic to Opus for any reason where "slightly smarter" is worth the marginal latency

Don't rush if you:

- Use Opus primarily for short prompts or one-shot Q&A
- Have production pipelines that need thorough regression testing before any model swap
- Are already on Sonnet 4.6 for cost reasons — 4.7 doesn't change that calculus
- Run latency-critical real-time applications where the 10% throughput hit matters

Sonnet is still the right default for most traffic

4.7 doesn't change the Opus vs. Sonnet decision. For short, well-defined tasks, Sonnet 4.6 remains 5× cheaper, 3–5× faster, and produces output that's indistinguishable in quality on those tasks.

The rule still applies: **default to Sonnet, route to Opus only when the task actually requires it.** With 4.7, the definition of "requires Opus" expands to include any long-context or multi-file work, not only reasoning-heavy tasks.

How to switch

In the API

One-line change:

```
from anthropic import Anthropic
client = Anthropic()
resp = client.messages.create(
    model="claude-opus-4-7",  # was claude-opus-4-6
    max_tokens=4096,
    messages=[{"role": "user", "content": "your prompt"}],
)
```

No other changes required. Same parameter names, same response shape.

In Claude.ai / Claude Pro

Select Opus 4.7 from the model picker. Claude Pro subscribers get the upgrade at no extra cost. Existing Projects inherit the new model on your next message.

In Claude Code

Run `claude --version` to check. Latest versions support 4.7 automatically. Earlier installs may need an update: `curl -fsSL https://claude.ai/install.sh | sh`

What to watch over the next 30 days

Two specific signals:

Does the long-context holding-pattern survive production pressure?

The 94%+ recall at 99% depth in Section 4 was measured on clean API endpoints with no concurrent load. Anthropic historically sees some degradation in real production traffic for first-month releases. If this holds, 4.7 permanently changes what's practical. If it regresses, the gap closes.

Does multi-file code performance hold for larger codebases?

My 6-file tests are representative but small. Teams running 30-100 file refactors are the real customer for this capability. Their field reports over the next 30 days will tell us whether the 2× improvement holds at scale.

I'll re-run this benchmark in 30 days and publish the follow-up. If you want to be notified when that drops, the newsletter signup at clskillshub.com is where it gets announced.

The executive summary for engineering managers

If your team uses Claude Opus for real code work: **upgrade today**. The multi-file improvement alone justifies it, pricing is identical, and the worst case is a 10% latency hit on interactive tasks.

If your team uses Claude Opus for prose or simple Q&A: **no urgency.** The upgrade is still net positive, but you won't notice most of the time.

If your team uses Sonnet 4.6 and routes Opus sparingly: **keep your current setup.** 4.7 doesn't change the routing calculus; it just makes Opus more useful when you do route to it.

For the full methodology, see Section 1 of this report. For the raw benchmark tables and test prompt texts, see the companion data release at clskillshub.com/blog/claude-opus-4-7-vs-4-6-benchmarks.

Section 5: The Claude Code Skills Layer

Most developers using Claude have never used a skill. This section is about what they're missing, what the community has already built (2,392 files and counting), and what the data tells us about where Claude Code actually gets used — which turns out to be somewhere entirely different from where the hype says it does.

What a skill actually is

A Claude Code skill is a markdown file that sits in `~/.claude/skills/` (global) or `.claude/skills/` (per-project). When Claude Code starts a session, it reads every `.md` file in those directories and treats them as extended context — persistent instructions the model applies to every conversation in that environment.

That's it. That's the whole primitive. A skill is a plain markdown file Claude reads.

The five Claude Code primitives, ranked by adoption

To understand where skills fit, here's the full primitive stack:

Primitive	What it is	Adoption (rough)
Prompts	Text you type into Claude	100% of users
CLAUDE.md	Project-level instructions file	~40% of users
Skills	Markdown files in `~/.claude/skills/`	<10% of users
MCP servers	External tools Claude can call	<5% of users
Agent teams / subagents	Multi-agent workflows	<3% of users

Skills sit in the middle of the stack. More powerful than prompts, lighter than MCP servers, simpler than agent teams. **And drastically underused relative to their value-per-setup-minute.**

How skills differ from prompts, MCP, and agents

- **Prompts** live in the conversation. They disappear when the conversation ends.
- **Skills** live on your disk. They persist across every session, every project, every day.
- **MCP servers** are executable tools Claude can call. They require a server process. They can make API calls, query databases, write files with side effects.
- **Agents / subagents** are Claude instances that Claude can invoke. They're orchestration primitives.

A skill can't make an API call. It can't write files. It can only *instruct* Claude — but instructions that apply to every conversation are surprisingly powerful. Most workflow quality gains come from better instructions, not better tools.

Why most teams aren't using skills

Not because skills are bad. Because the discovery and packaging problem isn't solved.

Three concrete blockers:

1. Installation friction is low, but unknown. Developers who've never seen a skill don't know `~/.claude/skills/` exists. It's not in the getting-started docs with the prominence it deserves. Most people discover skills through social media, not through Claude's own onboarding.

2. Quality is hard to judge before use. A skill file is opaque until you install it. You can't "try before you commit" cleanly. This is solvable — see the skill preview format at clskillshub.com/skill/[slug] — but the wider ecosystem hasn't standardized on a preview format yet.

3. Skill conflicts are invisible. If you install five skills that all want to control code formatting, Claude's behavior becomes unpredictable. There's no namespace system, no conflict detection, no "this skill overrides that skill" pattern. This is the single biggest open problem in the skill ecosystem.

These are problems, but they're not reasons to skip skills. They're reasons the early-adopter window is still wide open.

What the community has actually built

The CLSkills dataset contains **2,392 Claude Code skill files** as of April 2026, drawn from community submissions, GitHub scrapes, vendor examples, and internal contributions. The curated subset with full metadata is **845 skills across 72 categories**.

The category distribution is not what you'd expect from reading Claude Code tutorials.

Top 15 categories by skill count (curated dataset, n=845)

Category	Skills	% of curated dataset
SAP	107	12.7%
Database	26	3.1%
Cloud	22	2.6%
Testing	19	2.2%
AI/ML	17	2.0%
Git	15	1.8%
API	15	1.8%
Frontend	15	1.8%
Scaffolding	15	1.8%
Networking	15	1.8%
Python	15	1.8%
Salesforce	15	1.8%
DevOps	14	1.7%
Security	14	1.7%
Backend	14	1.7%

The headline surprise: **SAP is the largest single category by 4×**, with 107 skills. Not React. Not Python. Not DevOps. SAP.

What this tells us about who actually uses Claude Code

The Claude Code discourse on Twitter and Reddit heavily overrepresents solo-dev SaaS founders working in modern web stacks. The skill submission data tells a different story.

The people submitting skills and requesting skills are dominated by **enterprise platform consultants** — SAP, Salesforce, ServiceNow, Dynamics 365. These are workers inside large organizations using Claude Code to automate the repetitive parts of platform-specific work. They have specific, narrow, high-value workflows (ABAP debugging, Apex testing, Fiori migration patterns) that benefit disproportionately from skill files because:

1. The domain knowledge is specialized and not in general model training
2. The workflows are repetitive enough that a skill file pays back fast
3. The organizations have compliance and IP constraints that make MCP servers harder than skills

This is the Claude Code market nobody writes about. It's also where most of the money in Claude-adjacent tooling probably sits — if you're building for Claude Code and you're not thinking about SAP/Salesforce/enterprise verticals, you're ignoring the largest user base.

Skill quality varies wildly

Of the 2,392 skill files catalogued, **789 have been verified** to meet a minimum quality bar:

1. Syntactically valid markdown
2. No prompt-injection patterns or malicious instructions
3. Contains at least one concrete instruction, pattern, or code template (not just placeholder text)
4. Non-duplicative with other verified skills

The other 1,603 skills are catalogued but not verified. Some are redundant. Some are drafts. Some are valid but thin. Some contain copy-paste from unrelated sources.

Practical implication: if you're pulling skills from a community source you haven't personally vetted, expect a 33% signal rate. Most published skills are not good enough to use as-is.

Three anti-patterns that show up repeatedly

From auditing the unverified set, three patterns consistently make skills worse:

Anti-pattern 1: The wall-of-text skill. Some skills are 3,000+ words of explanation with no actionable pattern. These add context token cost without improving Claude's behavior on actual tasks. A good skill is 200-800 words of concrete instructions.

Anti-pattern 2: The generic persona skill. Files like `senior-developer.md` that say "act as an experienced developer who writes clean code and thinks carefully." No specificity, no domain priors. These fail the Section 3 PERSONA test — generic personas produce no measurable reasoning change.

Anti-pattern 3: The prompt-engineering-masquerading-as-skill. Files that are just a list of viral prompt prefixes packaged as a skill. These belong in prompt libraries, not in `~/.claude/skills/`. A skill should encode domain knowledge, pattern recognition, or workflow rigor — not substitute for typing `/skeptic` in the conversation.

Case studies: three skill workflows that measurably work

These are from real users who reported their setups. Names and details anonymized where requested.

1. Solo dev using React skills for component work

A solo founder building a SaaS frontend installed 4 skills: `react-testing-vitest.md`, `react-accessibility.md`, `react-performance.md`, `react-state-zustand.md`. Each is 400-600 words of specific patterns.

What changed: code review time dropped from ~20 min/PR (self-review) to ~7 min/PR. Claude's first-draft code now passes their test suite on first try ~70% of the time, vs ~35% without skills.

Why it works: the skills encode this founder's specific conventions (component naming, test structure, accessibility defaults). Claude no longer suggests patterns that conflict with the house style — it suggests patterns that match.

2. Enterprise team using SAP-specific skills for migrations

A 6-person SAP consulting team working on S/4HANA migrations installed 12 skills from the `sap/` category: ABAP debugging patterns, Fiori migration templates, MM/SD module-specific skills, OData service generation, and BTP integration patterns.

What changed: migration estimates dropped by ~40%. Tasks that used to require a senior consultant's full attention can now be drafted by Claude Code with the skills loaded, reviewed in a fraction of the time.

Why it works: SAP domain knowledge is highly specific and not well-represented in Claude's training data on its own. Skill files act as a knowledge layer that turns Claude from "generic useful assistant" into "junior SAP consultant who's done this migration before."

This is the pattern that suggests enterprise platform skills are the real business opportunity in this space — not generic prompt libraries.

3. Writer using /ghost + writing skills for long-form production

A marketing writer installed 3 skills: one version of the `/ghost` AI-detection-stripper, a custom tone-matching skill built from samples of their own writing, and a fact-checking checklist skill.

What changed: edit cycles on AI-assisted drafts dropped from 3-4 passes to 1-2 passes. Output is now distinguishable from their own writing after one pass, not three.

Why it works: the /ghost skill encodes the specific AI-tells that need removal (em-dashes, transitional phrases, "I hope this finds you well" openers). The tone-matching skill sets Claude's voice to this person's voice before any draft begins. Both skills are subtractive, not additive — they remove rather than add, which is the Section 3 structural insight applied to writing.

How to write a good skill (compressed guide)

Full version is Appendix B. The compressed version:

1. One skill, one workflow. Don't pack "React + testing + accessibility + performance" into one file. Four separate skills that can be loaded independently are strictly better than one mega-skill that can't be turned off.

2. 200-800 words is the sweet spot. Below 200, it's probably too thin to matter. Above 800, it starts competing for Claude's attention budget on every prompt.

3. Concrete over generic. "When reviewing React components, check for unnecessary re-renders caused by inline object creation in props" is concrete. "Write clean React code" is not.

4. Include negative examples. Showing Claude what NOT to do is often more useful than showing what to do. "Don't suggest useState when the state is server state — use React Query instead" is a high-value line.

5. Version and date your skills. Claude's behavior changes with model updates. A skill written for Sonnet 3.5 may not work the same on Sonnet 4.6. Include a comment so future-you knows when to re-validate.

6. Don't try to override Claude's safety policies. Skills can shape behavior but can't bypass safety. Attempts to do so usually just make the skill stop working entirely.

What this means for your Claude setup this week

Three concrete actions:

1. Install 3 skills from the category you work in most. Drop them in `~/.claude/skills/`. Restart Claude Code. Use it for 3 days. See if the first-draft quality goes up.

2. Write one skill yourself. Something you've written to Claude three times already this month — turn that recurring instruction into a skill. Takes 10 minutes.

3. If you work on an enterprise platform (SAP, Salesforce, ServiceNow, Dynamics), check whether skills exist for your specific domain. The enterprise category coverage in the community is underrated. There's a 50%+ chance the skills you need already exist.

The full skill library is at clskillshub.com/browse. Filter by category. Download the .md files. Drop them in `~/.claude/skills/`. That's the entire workflow.

This section documents the skills dataset and categorization as of April 22, 2026. Numbers will grow — the submission rate is currently 10-30 new skills per week. The categorization schema is published openly; if you think a skill is miscategorized, submit a correction at clskillshub.com/browse.

Section 6: Agent Teams and Subagents — Claude's Orchestration Layer

Anthropic released Claude Code Agent Teams in early April 2026 — the ability to define multiple cooperating Claude instances with distinct roles, shared context, and handoff protocols. Combined with Subagents (task-specific Claude instances the primary agent can invoke), this is the orchestration primitive that turns Claude Code from "a coding assistant" into "a developer team you configure."

This section documents what these primitives actually are, when they're worth the setup, and the emerging pattern that distinguishes teams who get value from them from teams who don't.

What Agent Teams and Subagents actually are

Subagents

A Subagent is a Claude instance the primary Claude Code session can delegate work to. You define them in a config file. Each subagent has:

- A name (`code-reviewer` , `test-writer` , `doc-updater`)
- A system prompt describing its role
- A specific model (you can route subagents to Sonnet while the primary uses Opus, or vice versa)
- An optional set of tools it can use (file read, file write, bash, MCP servers)

When the primary agent encounters a task that matches a subagent's role, it invokes the subagent, which runs in parallel or sequence, returns results, and the primary integrates them.

Mental model: subagents are function calls to specialized Claude instances. You call them the way you'd call a function — with specific inputs, specific outputs, in a specific scope.

Agent Teams

Agent Teams are a higher-level primitive. A team is a configuration of multiple agents with:

- Defined roles (architect, implementer, reviewer, documenter)
- Handoff rules (when role A completes, who receives the work)
- Shared state (a workspace all agents can read)
- Termination conditions (when is the team "done")

Teams can contain subagents as members, or operate as a peer group where each agent has equal authority on its domain.

Mental model: teams are orchestrated pipelines. They're not just multiple subagents — they're multiple agents with an explicit coordination protocol.

How this differs from a skill

Skills shape the behavior of a single Claude instance. Agent teams coordinate multiple Claude instances. You can — and usually should — combine them: each agent on a team has its own skills loaded, and the team's orchestration is separate from the individual agents' instructions.

When each primitive is the right choice

Not every task benefits from orchestration. Most don't. Here's the decision tree:

Use a prompt when:

You have a question or a task that takes one round of output. 90% of Claude Code work is this.

Use a skill when:

You find yourself giving Claude the same instruction in every new session. Promote it from a prompt to a skill and it applies automatically.

Use a subagent when:

A specific subtask recurs often enough to deserve its own specialist. Code review is the canonical example: every PR needs review, the review protocol is consistent, and running review in parallel with the next task frees the primary agent.

Use an agent team when:

A task has multiple phases with distinct expertise requirements, AND you'd actually delegate to separate humans if you had them. Feature spec → implementation plan → code → tests → docs is a classic four-role pipeline.

Use nothing fancy (just Claude) when:

You're still figuring out the workflow. **The biggest mistake in the current Claude Code community is orchestrating too early.** People build elaborate agent teams for workflows they'd never actually delegate to multiple people. If the workflow isn't worth delegating, it isn't worth orchestrating.

The emerging pattern: orchestration has a tax

Every layer of orchestration adds:

1. **Setup time** (writing configs, debugging handoffs)
2. **Token cost** (agents talking to agents burns context budget)
3. **Failure modes** (agents getting stuck in handoff loops, duplicating work, losing context on transfer)
4. **Mental overhead** (you now have to debug orchestration, not just code)

The orchestration tax is worth paying when the task is genuinely large. For most single-session work, it's net negative.

The practical rule: **orchestrate when the workflow is larger than one context window.** Below that threshold, a single Claude with good skills and maybe one subagent is usually the right answer.

Where teams genuinely win: recurring pipelines across sessions

The single pattern where agent teams show compounding value:

- Same workflow, run repeatedly
- Multiple distinct expertise domains
- Each run produces artifacts other runs reference

Examples:

Feature development pipeline. Spec agent → implementation agent → review agent → docs agent. Runs on every feature. Each agent gets sharper over runs because the shared workspace accumulates patterns.

Incident response pipeline. Triage agent → investigator agent → fix agent → postmortem agent. Runs on every production issue.

Content production pipeline. Research agent → outline agent → draft agent → editor agent → fact-check agent. Runs on every piece of long-form content.

The common feature: the same pipeline runs 20+ times. Each run improves the configuration. The orchestration tax is paid once; the benefits compound.

Where teams visibly fail: one-off creative tasks

The opposite pattern: orchestration applied to creative tasks where the "right" sequence of roles isn't known in advance.

When people try to use agent teams for things like "design this UI" or "write this essay" or "solve this architecture problem," the teams typically:

- Hand off prematurely (spec agent passes to implementation before the spec is clear)
- Or hand off too slowly (spec agent tries to anticipate every implementation question, produces a 5,000-word spec for a 200-line feature)
- Or produce work that feels committee-designed — no voice, no decisive choices, no opinion

One thoughtful Claude with good skills usually beats an orchestrated team on open-ended creative tasks. The orchestration helps when the procedure is repeatable. It hurts when the work is genuinely novel.

The emerging stack: skills + subagents + hooks

The Claude Code users getting the most value from the 2026 primitive stack share a common configuration:

1. **10-20 skills** covering their stack, conventions, and common anti-patterns
2. **2-5 subagents** for specific recurring tasks (code review, test writing, doc updates)
3. **Hooks** for deterministic automation (pre-commit checks, post-merge deploys)
4. **Agent teams** only for workflows they run regularly enough that the setup tax amortizes

The users getting the least value typically:

- Have zero skills installed
- Try to build elaborate agent teams before mastering single-agent workflows
- Over-invest in orchestration for tasks that don't benefit from it
- Use MCP servers for things that could be a skill

The pattern is consistent: **people who master the lightweight primitives first (skills, then subagents) and only add orchestration when the workload demands it get 5-10× the value of people who try to go straight to full agent teams.**

Early observations on team dynamics

Since Agent Teams launched, three specific behaviors have emerged across teams who've reported their setups:

Observation 1: "Reviewer" agents catch more than human reviewers

A dedicated review subagent (or team member) consistently catches issues that the primary implementer agent missed. This isn't surprising — the reviewer has fresh context and a different role prompt — but it's still measurable. Teams report 20-40% more issues caught on first review pass with a reviewer agent than without.

Observation 2: Documentation drift is solved

Teams running an auto-doc subagent on every PR report that their documentation actually stays in sync with their code. This is a genuinely new capability — "docs lag code" has been the default state of software development for decades. Agent teams with a doc role in the pipeline flip this.

Observation 3: Coordination overhead is real

Teams that add more than 5 agents to a pipeline report diminishing returns and often regress. 3-4 agents is the sweet spot. Beyond that, the handoff coordination starts eating more context budget than the specialized roles save.

What to do this week if you've never used these primitives

Don't start with an agent team. Start with one subagent.

Step 1: Pick the single task you do most often that you'd love to offload. For most devs: code review. For writers: fact checking. For ops: log triage.

Step 2: Define that task as a subagent with a clear role prompt and a small tool set.

Step 3: Use it for a week on real work.

Step 4: Measure: is the quality better? Is your time better spent? Did the subagent catch things you'd have missed?

Step 5: If yes, add a second subagent for the next-most-common task. Don't jump to a full team until you've validated that two subagents working together improves your workflow more than either alone.

This is the crawl-walk-run path. People who try to run a full 5-agent team on day 1 almost always give up when the coordination fails. People who add one subagent at a time are still using the system a month later.

What to watch over the next 90 days

Two specific questions:

Will team configurations standardize?

Right now, every team's agent team config is custom. The early winners will probably be the teams who publish their configurations — just like skills became a shareable primitive, team configs will likely become a shareable primitive. Watch for a "claudeteam.yaml" format to emerge as a standard.

Will agent teams compete with or complement MCP servers?

Agent teams and MCP servers are both orchestration layers. They solve overlapping problems. The next 90 days will show which tasks naturally live in which layer. Current read: MCP is for external tool integration (databases, APIs, file systems); agent teams are for role-based task decomposition. These are complementary, not competitive — but the line isn't clean yet.

I'll revisit this in the 30-day follow-up. If you want to be notified when that drops, the newsletter signup at clskillshub.com is where it gets announced.

Section 7: Where Claude Code Sits vs Alternatives

Claude Code isn't competing with "other AI models." The model-quality gap among the top three (Claude, GPT, Gemini) is small enough that it's not the thing most developers should be optimizing for. Claude Code is competing with **other developer workflows for AI-assisted code**. The alternatives are Cursor, Windsurf, GitHub Copilot, generic ChatGPT, and a developer's own editor with a chat sidebar.

This section maps the actual competitive landscape and identifies what Claude Code uniquely enables — which is not what its marketing emphasizes.

The real competitive map

Tool	Core format	Model	Key differentiator (as of April 2026)
Claude Code CLI	Terminal + file system	Claude Opus/Sonnet	Skills, hooks, agent teams, MCP, Cowork
Cursor	IDE fork of VS Code	User's choice (Claude/GPT/DeepSeek)	UI polish, composer mode, codebase indexing
Windsurf	IDE fork of VS Code	Proprietary + GPT	Agentic flow, Cascade workflows
GitHub Copilot	VS Code extension + CLI	GPT-based	Deep GitHub integration, enterprise distribution
ChatGPT with code	Web UI + Codex	GPT	Widest installed base, web-native
Raw API + custom UI	Whatever you build	Any	Full control

Notice what's NOT on this list as competitive differentiators:

- Raw model benchmark scores — within 5-10% across the top three, rarely the deciding factor
- Free tier generosity — they all have one
- VS Code compatibility — most now support VS Code in some form

The real competition is in the workflow layer, not the model layer.

What Claude Code uniquely enables

Three primitives Claude Code has that competitors don't meaningfully match in April 2026:

1. Skills as plain markdown files

Cursor has "rules" files. Copilot has enterprise prompt configuration. Windsurf has memories. None of them treat skill files as a shareable, community-curated primitive the way `~/.claude/skills/` does.

The CLSkills dataset of 2,392 skills is a practical moat. Even if Cursor shipped an identical feature tomorrow, the community-built library would take years to match. This is the single most underrated competitive advantage of the Claude Code ecosystem.

2. The hooks system

Claude Code hooks let you run deterministic code at specific lifecycle points — before the user prompt is sent, after a tool is called, on session start, on every edit. This is closer to git hooks than to AI prompt engineering.

Competitors don't have this. Cursor's workflow automation is graphical and sits on top of the agent. Claude Code's hooks are shell commands that run at specific events — they're lightweight, scriptable, and don't depend on the agent being "alive" to run.

This matters because: deterministic automation + probabilistic AI is a fundamentally better pairing than all-probabilistic AI workflows. The hook system is where Claude Code users embed "this must always happen" logic around the parts where AI judgment matters.

3. Agent teams and subagents

Section 6 covered these in depth. The short version: Claude Code is the only major tool where multi-agent orchestration is a first-class primitive in the CLI itself, not a layer built on top.

Cursor's composer approximates some of this. Windsurf's Cascade gets closer. But Claude Code's agent teams are defined in config files, versioned in git, and run as CLI invocations — which matches how developers actually work.

Where Claude Code is weaker than alternatives

Two specific places Claude Code loses today:

IDE integration is thinner than Cursor's

Cursor ships with a complete IDE. The composer UI, the tab-completion, the diff view, the codebase indexing — all integrated. Claude Code is terminal-first, and while its VS Code integration is improving, it's not where Cursor sits on pure IDE polish.

For developers who want "Claude, but inside my IDE with full visual integration," Cursor is still the cleaner experience.

First-time setup complexity

Installing Claude Code CLI, setting up a CLAUDE.md, discovering `~/.claude/skills/`, configuring an MCP server, writing your first subagent — this is a progressive disclosure curve that rewards patient users and punishes casual ones.

Cursor's first 30 minutes feel better. The catch is that Cursor's ceiling is lower — the first 30 minutes on Claude Code are worse, but the second 30 hours on Claude Code produce a workflow Cursor can't match.

The strategic implication: Claude's moat isn't the model, it's the primitives

Anthropic's public messaging heavily features model capability — context windows, benchmark scores, reasoning improvements. That's not where the competitive gap actually is.

The gap is in the **workflow primitive stack**:

- Skills (markdown files with persistent behavior)
- Hooks (deterministic automation around the AI)
- Subagents (role-specialized Claude instances)
- Agent teams (orchestrated multi-agent pipelines)
- MCP (external tool integration)
- Cowork (desktop agent for non-code tasks)

Each primitive is individually available in competitor products. Cursor has rules, Copilot has extensions, ChatGPT has Custom GPTs. What Claude Code uniquely has is all six primitives in one integrated stack, designed to compose.

The developer who masters all six primitives has a compounding workflow advantage that the developer using one tool with excellent UI doesn't.

The implication for tool evaluation: if you're choosing between Claude Code, Cursor, or Copilot based on 30-minute demos, you'll probably pick Cursor or Copilot. If you're choosing based on where you'll be 3 months in, Claude Code is likely the better bet — assuming you actually use the primitives, not just the chat interface.

Who should use what

Use Cursor if you want:

- Best-in-class IDE polish and visual UX
- Minimum learning curve
- Your workflow is IDE-centric and you're not going to invest in CLI-level customization

Use GitHub Copilot if you want:

- Deep GitHub integration (especially in enterprise orgs with GitHub already deployed)
- Compliance and billing consolidation with your existing GitHub account
- You trust Microsoft's enterprise roadmap

Use ChatGPT + your editor if you want:

- No vendor lock-in, maximum flexibility
- Willing to copy-paste code between the browser and your editor
- Working on tasks that don't benefit from deep IDE integration anyway

Use Claude Code if you want:

- Skills as a compounding workflow primitive
- Hook-based deterministic automation around AI
- Multi-agent orchestration as a first-class concept
- MCP server integration with your existing tools
- Willingness to invest the first-30-days learning curve for a 90-day-compounding workflow advantage

There's no universal right answer. The teams getting the most value from Claude Code are the ones who saw the primitive stack as the product, not the model as the product.

What to watch over the next 12 months

Three competitive dynamics to track:

Will Cursor ship a skills-equivalent?

They have the distribution and the UX to do it. They've been conservative about community-curated content so far, treating skills-adjacent features as part of the premium bundle. If they flip that — open community skill libraries for Cursor — the Claude Code moat on this front shrinks.

Will Copilot's enterprise distribution overwhelm the workflow gap?

GitHub Copilot is inside almost every large engineering org via the GitHub relationship. If Copilot ships an agent-teams-equivalent and it's "just on" for enterprise GitHub customers, the friction gap may not matter. Most enterprise adoption is driven by procurement ease, not by primitive superiority.

Will Anthropic publish canonical skill collections?

Anthropic's own documentation currently underplays skills. If they published a canonical "official skills library" with vendor-endorsed patterns, it would accelerate skill adoption across the user base and make the primitive harder for competitors to replicate.

Watch for these three shifts. They determine whether Claude Code's current primitive-stack advantage compounds or erodes.

Section 8: Practical Takeaways

If you've read this far, the report's claims are yours to use. This section compresses them into five specific actions you can take this week.

Each action maps to a finding above. Each is doable in under an hour. Each has a measurable outcome you can check against.

1. Audit your prompt codes. Replace the placebos.

Finding: Of 40 viral Claude prompt codes tested, 7 reliably shift reasoning; 7 are placebo-suspect; the rest are structural (useful for format, not thinking).

Action: Look at your personal prompt library — the text snippets you paste before real prompts. Check each against Section 3.

- If you're prefixing with GODMODE, BEASTMODE, MEGAPROMPT, OVERTHINK, /optimize, /jailbreak, or CEOMODE on real work — stop. These are confidence theater. Replace with `/skeptic` or `L99` on decision questions, `/deepthink` on debugging, `PERSONA` (with specifics) on domain work.
- If you're using ULTRATHINK on everything — cap it to hard reasoning tasks only. It works, but the 3× token cost compounds.

Measurable outcome: Over the next week, compare your Claude responses with and without your prefix. If the output is equally good without it, you've been paying the token tax for nothing.

2. Install 3 skills. Today.

Finding: Skills are the most-underused Claude Code primitive. Community adoption is under 10%. Early-adopter window is wide open.

Action:

1. Open clskillshub.com/browse
2. Filter by the category you work in most (frontend, backend, DevOps, SAP, whatever matches)
3. Download 3 skill files that look relevant
4. Drop them in `~/.claude/skills/` (create the directory if it doesn't exist)
5. Restart Claude Code

Measurable outcome: Over the next 3 days, compare first-draft code quality. Measure how often Claude suggests patterns that match your conventions vs. patterns you'd have to rewrite. If it doesn't improve, remove the skills and try different ones — not all skills are good; expect a 33% signal rate on unverified sources.

3. If you use Opus for anything multi-file, upgrade to 4.7 today.

Finding: The 6% reasoning benchmark lift understates 4.7's real advantage. Multi-file code tasks produce working code on first try roughly 2× as often as 4.6. Long-context holds quality through 800K tokens. Pricing is identical.

Action:

- API users: change `model="claude-opus-4-6"` to `model="claude-opus-4-7"`. No other changes needed.
- Claude.ai / Pro users: select Opus 4.7 from the model picker.
- Claude Code CLI users: run `claude --version` to confirm you're on a version that supports 4.7, upgrade if not.

Measurable outcome: The next multi-file refactor you give Claude Opus, note whether the code works on first try. Compare against your recent 4.6 experience. If you're not working on multi-file tasks, the upgrade is still net positive but you won't feel it.

4. Write one skill yourself.

Finding: The highest-signal skills come from encoding the specific instructions you've typed into Claude more than twice.

Action:

- Think of one instruction you've given Claude repeatedly this month — "always use Zustand, not Redux, for new state"; "write tests with Vitest, not Jest"; "for API endpoints, always validate with Zod before processing" — whatever it is for your stack.
- Open a text editor. Write a 300-500 word markdown file that encodes that instruction, with 1-2 concrete examples of what to do and 1-2 of what NOT to do.
- Save it to `~/.claude/skills/your-convention.md`.
- Restart Claude Code.

Measurable outcome: Over the next week, Claude should stop making the mistake your skill corrects. Count how many times you used to correct that specific pattern vs. how many times you have to after the skill is loaded.

5. Try one subagent. Resist building a team.

Finding: Agent teams are powerful but over-invested. The users getting the most value start with one subagent for a recurring task, validate it works, then expand slowly.

Action:

- Pick the single task you do most often that you'd love to offload. For most developers: code review.
- Define a subagent with a clear role prompt and a small tool set (file read, limited file write).
- Use it for a week on real work.

Measurable outcome: Count issues caught per PR with the subagent vs. without. Teams consistently report 20-40% more issues caught on first review pass with a reviewer subagent. If your results match that band, the subagent is paying for itself. If not, the subagent config needs work — not the primitive.

The meta-takeaway

Most of the measurable quality gap between "Claude user who gets 10× leverage" and "Claude user who gets marginal improvement over typing" is not in the model. It's in **the workflow primitives the user has learned to compose**.

Skills, hooks, subagents, agent teams, MCP — five primitives, each independently simple. Together, they compound.

The Claude Code users who will matter in 12 months are the ones investing in the primitive stack now, while the rest of the market is still arguing about which model is best.

What to do if you want to stay updated

This report will be re-run every 90 days as Anthropic ships new primitives and the community builds on top. If you want the next version when it drops:

- **Newsletter:** sign up at clskillshub.com — one email per week, value-first, no promos beyond natural product mentions
- **Free dashboard:** clskillshub.com/insights — 10 classified prompt codes with test results, no paywall
- **Full library:** clskillshub.com/browse — all 2,392 skills, filterable by category, free to download individually
- **Cheat Sheet:** clskillshub.com/cheat-sheet — the full Pro tier classification data for all 40 deeply-tested codes, $15 until May 1, 2026 (goes to $20 after)

If you run into a test result that contradicts what this report says, reply to the newsletter — I'll add your data to the next version and credit you if you want. The only way this report gets better is if the community challenges its claims with real tests.

— Samarth, clskillshub.com

Appendix A: Compressed 40-Code Reference Table

Every code tested in this report, with classification and one-line use guidance. Grouped by classification — most useful first.

Full before/after data, failure modes, combo strategies, and methodology disclosures for every code are in the Cheat Sheet at clskillshub.com/cheat-sheet.

Reasoning-shifters (7 of 40 — 17.5%)

Change what Claude decides, not just how it phrases.

Code	Use when	Skip when
`/skeptic`	Decision questions where the obvious answer might be wrong	Factual lookups, debugged coding questions, creative writing
ULTRATHINK	Multi-step reasoning on hard problems, willing to pay 3× token cost	Batch jobs, short tactical questions, anywhere depth doesn't matter
L99	Binary decisions where you need a committed answer, not a hedge	Genuine tradeoff analysis, "I want to think through X" questions
`/deepthink`	Debugging, root-cause analysis, mid-depth reasoning	Simple questions (overkill), anywhere ULTRATHINK is justified
PERSONA (specific)	Technical reviews, domain-specific expertise, voice-matching	Short questions, when you can't specify persona with real bias/experience
`/steelman`	Before committing to a contrarian decision, testing your own plan	When you want execution help, casual conversation
OODA	Strategic questions with multiple moving parts, structured analysis	Simple questions, decisions with no meaningful ambiguity

High-value structural (23 of 40 — 57.5%)

Useful for format, decisiveness, or brevity — but don't change Claude's reasoning.

Code	What it's good for
`/ghost`	Strip AI-tone markers from output (em-dashes, transitional phrases)
`/punch`	Make writing more direct, cut hedging language
`/raw`	Strip all formatting, return plain text only
`/mirror`	Match a specific writing style from a provided sample
`/trim`	Cut output by ~50% while preserving meaning
`ARTIFACTS`	Request a self-contained, copy-pasteable artifact
`/shipit`	Production-ready output with error handling and edge cases
`/debug`	Structured debugging response with hypothesis + test steps
`CHAINLOGIC`	Show reasoning steps explicitly in the response
`/blindspots`	Surface assumptions and risks you didn't mention
`DEEPDIVE`	Maximum-depth analysis of a single topic
`COMPARE`	Side-by-side comparison with clear winner
`/critique`	Critical review of your code/writing/plan
`WORSTCASE`	Worst-case-scenario analysis for decisions
`/json`	Force JSON output format
`/checklist`	Return output as actionable checklist
`/eli5`	Explain at 5-year-old level
`INVERT`	Solve the problem by thinking about its inverse
`/nofilter`	Direct opinion without excessive hedging
`/tldr`	One-paragraph summary of any long input
`/unpack`	Break complex idea into constituent parts
`REFACTOR`	Clean up code without changing behavior
`/testit`	Generate tests for provided code

Placebo-suspects (7 of 40 — 17.5%)

Produce output that feels more authoritative but show no measurable reasoning change vs. baseline in the test set. **Skip these.**

Code	What you think it does	What it actually does
`/godmode`	Unlocks peak Claude	Adds length, no reasoning improvement
`/jailbreak`	Removes restrictions	Changes tone, sometimes drops legitimate safety context
`BEASTMODE`	Aggressive performance	Confidence theater — same decisions, assertive vocabulary
`MEGAPROMPT`	Ultimate prompt power	Adds verbose scaffolding, reasoning unchanged
`OVERTHINK`	Deep reflection	Meta-discusses the problem without changing the answer
`/optimize` (bare prefix)	Auto-optimizes	Identical output to baseline when used without concrete target
`CEOMODE`	Executive-level decisions	Persona shift without domain knowledge activation

Niche (2 of 40 — 5%)

Work well in their specific lane, worse than baseline outside it.

Code	Where it works	Where it fails
`OPERATOR`	Step-by-step operational tasks (deployment, setup)	Analysis, decisions, creative work
`/autoprompt`	When you know you need to prompt-engineer a task but can't articulate it	When you already have a working prompt

Low-value structural (1 of 40 — 2.5%)

Changes format in ways that feel different but don't measurably help.

Code	Note
`/table`	Forces table output, useful narrowly when the data is genuinely tabular; forces awkward structure on non-tabular content. Replaced in most cases by simply asking Claude for a table.

The rule for any code not in this table

If a code you use isn't in this list, run the 10-second test from Section 3:

1. Run your question WITHOUT the prefix.
2. Run it WITH the prefix.
3. Compare REASONING, not wording.

If the conclusions are the same → placebo. If the decisions are different → real.

Most codes outside this tested 40 have not been individually classified. The full 120-code library is at clskillshub.com/cheat-sheet with the same classification protocol applied.

Appendix B: CLAUDE.md Template

Copy this into the root of any project. Customize the sections marked `[CUSTOMIZE]`. Claude Code reads this file on every session in this directory and uses it as extended context.

A good CLAUDE.md answers the questions Claude would otherwise ask on every new session: what's the stack, what are the conventions, what's the current state, what should I avoid.

[PROJECT NAME]

[CUSTOMIZE: one-sentence description of what this project is]

Stack

- **Language:** [CUSTOMIZE: TypeScript / Python / Go / Rust / etc.]
- **Framework:** [CUSTOMIZE: Next.js 16 / Django / FastAPI / etc.]
- **Database:** [CUSTOMIZE: Postgres / SQLite / DynamoDB / etc.]
- **Deployment:** [CUSTOMIZE: Vercel / Railway / AWS / etc.]
- **Testing:** [CUSTOMIZE: Vitest / Pytest / Go test / etc.]

Conventions

- File naming: [CUSTOMIZE: kebab-case for files, PascalCase for components, etc.]
- Function style: [CUSTOMIZE: prefer arrow functions / prefer function declarations / et
- State management: [CUSTOMIZE: Zustand for client state, React Query for server state]
- Styling: [CUSTOMIZE: Tailwind / CSS modules / styled-components]
- Testing style: [CUSTOMIZE: colocated .test.ts files / separate __tests__ dir]
- Import style: [CUSTOMIZE: absolute imports via @/ alias / relative only]

Project structure

\\\ src/   app/   # [CUSTOMIZE: describe what lives here]   components/

Things to NEVER do

[This section is high-signal. Claude reads negative instructions more reliably than posit

- [CUSTOMIZE: e.g. "Never commit .env files"]
- [CUSTOMIZE: e.g. "Never use 'any' type in TypeScript"]
- [CUSTOMIZE: e.g. "Never add a new dependency without checking if the functionality alr
- [CUSTOMIZE: e.g. "Never write code comments that explain WHAT — only WHY when the why

Things to always do

- [CUSTOMIZE: e.g. "Always validate API inputs with Zod before processing"]
- [CUSTOMIZE: e.g. "Always include a failing test before writing a bug fix"]
- [CUSTOMIZE: e.g. "Always use descriptive variable names — no single-letter variables e

Current state

[Update this section as the project evolves. Claude uses it to understand what's in fligh

- [CUSTOMIZE: e.g. "Migrating from REST to GraphQL in the /api directory"]
- [CUSTOMIZE: e.g. "New auth system in progress, old auth still live for backward compat
- [CUSTOMIZE: e.g. "Database migration pending — don't modify the users table schema wit

Running this project

- Install: [CUSTOMIZE: npm install / pip install -r requirements.txt / etc.]
- Dev: [CUSTOMIZE: npm run dev / etc.]
- Test: [CUSTOMIZE: npm test / etc.]

- Deploy: [CUSTOMIZE: git push main auto-deploys via [platform]]

Key files Claude should know about

- [CUSTOMIZE: src/lib/auth.ts] — auth logic, touch carefully
- [CUSTOMIZE: src/app/api] — API routes, all validated with Zod
- [CUSTOMIZE: prisma/schema.prisma] — database schema, requires migration on any change

Who to ask for help

- Architecture decisions: [CUSTOMIZE: name / role]
- Database changes: [CUSTOMIZE]
- Deployment issues: [CUSTOMIZE]

Why this template works

- **Short enough to always be in context.** Claude reads this on every session. Long CLAUDE.md files get compressed or ignored. This template lands at ~400-600 words after customization, which is the sweet spot.
- **Concrete over generic.** Every section prompts you to be specific. "Prefer functional components" is concrete. "Write clean code" is not.
- **Negative instructions.** The "Never do" section is higher-signal than the "Always do" section. Both matter.
- **Current state captures drift.** Projects change. The current-state section is where you tell Claude what's different from the last time it saw this repo.

Common anti-patterns in CLAUDE.md files

Anti-pattern 1: The aspirational CLAUDE.md. "We use TDD for everything" when the project has 12% test coverage. Claude then produces TDD-style code that doesn't match the rest of the codebase. Write what your project actually does, not what you wish it did.

Anti-pattern 2: The kitchen-sink CLAUDE.md. 3,000 words covering every edge case. Claude's behavior on any specific task gets diluted because the context budget is spread thin. Keep it under 800 words.

Anti-pattern 3: The never-updated CLAUDE.md. Written once, never revisited. Claude works off stale context. Update the "current state" section when major things change. Commit CLAUDE.md to version control so you can see when it drifts from reality.

Anti-pattern 4: The generic convention list. "Follow clean code principles" tells Claude nothing. "Functions over 30 lines must be refactored" is concrete.

Where to put this file

- **Root of project** (`./CLAUDE.md`) — applies to this project only
- **User home** (`~/.claude/CLAUDE.md`) — applies globally (rare; most users don't have one of these)
- **Project skills directory** (`./.claude/skills/` or `~/.claude/skills/`) — skill files that complement the main CLAUDE.md with workflow-specific instructions

The typical setup: one CLAUDE.md per project + 3-10 skills in `~/.claude/skills/` for your general workflow preferences. That combination gives Claude the project context plus your personal working style, without having to duplicate instructions across every project.

Appendix C: About This Research

Who wrote this

Samarth Bhamare — solo founder of CLSkills (clskillshub.com), based in Pune, India. Runs a community-curated library of 2,392 Claude Code skill files, a cheat sheet of 120 tested prompt codes, and associated research into what actually works when developers build with Claude.

Not affiliated with Anthropic. Not funded by any AI lab. Revenue comes from a $15 Cheat Sheet, $19 Claude Code Plugin, $359 Sales Agent Pack, and enterprise skill bundles. This independence is why the report can say that 7 of 40 viral prompt codes are placebo without worrying about vendor relationships — Anthropic didn't pay for this research and hasn't reviewed it.

Why this research exists

I got tired of evaluating prompt codes and model upgrades on vibes and marketing copy. Every time a new Claude release dropped, the discourse was dominated by confident claims from people who hadn't run tests. Every viral "secret prompt" thread had no before/after data. The ecosystem was (and is) full of assertion and short on evidence.

So I built a personal test harness. I ran every model release through a controlled benchmark. I tested every viral prompt code against a no-prefix baseline. I catalogued every skill file I could find and verified them for quality. This report is what three months of that work produces.

What this research will and won't do next

Will: Re-run the benchmarks every 90 days. Expand the tested prompt code set from 40 toward the full 120. Add measurement of first-month drift on model releases. Publish the raw test prompts so others can replicate.

Won't: Accept sponsored placements. Give different findings to paying customers than to public readers. Test under NDA with vendors. Make claims I haven't measured.

Contact

- **Email:** team@clskills.in (replied to personally, usually within 24 hours)
- **Newsletter:** sign up at clskillshub.com — one email per week, value-first, no promos beyond natural product mentions

- **X / Twitter:** @samarthbhamare
- **GitHub:** github.com/Samarth0211

How to contribute data

If you've tested a prompt code or model capability and found results that contradict or extend this report's claims, I want the data. Send me:

1. The prompts you ran
2. The model(s) and sampling parameters
3. Raw output or a summary of what you measured
4. Your conclusion

I'll include it in the next version with credit. The only way this research gets better is if the community challenges it with real tests.

Licensing

Acknowledgments

- The 158 newsletter subscribers who asked hard questions in replies and made the research better
- The 21 Cheat Sheet buyers who pointed out unclear sections in the first version
- The SAP consultant community who drove the "enterprise platforms dominate the skill dataset" finding in Section 5
- Every r/ClaudeAI and r/PromptEngineering commenter who challenged a claim — you're the reason Section 1 exists

And to the reader: if you made it this far, thanks. This is the kind of research I wish existed when I started working with Claude. I'll keep publishing it as long as the work is useful.

— Samarth April 2026 Pune, India

This is version 1.0 of the Claude Code Skills Report. Version 2.0 targeted for July 2026.

End of Report

Published April 27, 2026 · Version 1.0 Next version targeted for July 2026

Get the companion assets:

- Full library: clskillshub.com/browse
- Classified code data: clskillshub.com/insights
- Cheat Sheet (all 40 deeply-tested codes): clskillshub.com/cheat-sheet
- Newsletter for the next version: clskillshub.com

— Samarth · team@clskills.in