Comparing Five Prompting Styles on a Bug-Fix Task: A Controlled Study

Date: 2026-04-30 **Model under test:** Claude Opus 4.7 (1M context), via the `claude` CLI **Permission mode for spawned agents:** `bypassPermissions` (full bash access) **Task domain:** Fix bugs in a small Express + SQLite API **Total runs:** 15 (3 per style × 5 styles) **Wall clock for the full study:** ~24 minutes **Total cost:** ~$8–10 (rough estimate from per-run cost data)

Executive Summary

Five prompting styles were evaluated against an identical task: fix eight planted bugs in a small Express API. Each style was run three times. Outputs were scored on correctness (40%), security (15%), restraint (15%), code quality (20%, via two independent blind LLM judges), and cost (10%).

The spec-driven prompt won decisively. It was the only style to achieve a perfect correctness score (11/11 hidden tests) on every run, the most consistent (std dev 0.02 vs 4.47 for terse), and the cheapest in absolute tokens (278k average vs 821k for TDD). All three spec-driven runs scored ≥99.94/100.

Chain-of-thought rose to second place with 95.16/100 — a noticeable jump over the previous result. Asking the agent to "list every issue ranked by severity, then fix in order" produced consistent 10/11 results at the same token cost as a plain detailed brief.

TDD-framed remained the weakest of the structured styles. It cost more than 2× any other structured style for fewer bugs found (mean 8.33/11 vs 10/11 for CoT and brief). Process scaffolding has real overhead and did not pay off at this task size, even with full ability to run the test suite.

Terse prompts again produced clean code but missed bugs. Every terse run scored 5/5 on quality from blind judges, but only 5–7 of 11 hidden tests passed. Brevity transferred all responsibility for bug-discovery to the model, and the model didn't always rise to it.

Headline takeaway: for fixing bugs in a known codebase with Claude Opus 4.7, **write a real spec.** It produces the best results, the most consistent results, and at the lowest token cost.

---

1. Methodology

1.1 Task

A small Express + SQLite REST API with four endpoints (`GET /users`, `GET /users/:id`, `POST /users`, `GET /users/search`) and eight planted issues spanning four categories:

#	Issue	Category
1	SQL injection in `/users/search`	security
2	Pagination off-by-one (`page * size` instead of `(page-1) * size`)	correctness

#	Issue	Category
3	`Promise.resolve()` wrapper makes route return Promise instead of object	correctness
4	`POST /users` does not validate email; crashes on missing fields	correctness
5	N+1 query in `/users` (per-row `SELECT` for roles)	performance
6	Dead `formatLegacyUser()` helper	smell
7	`console.log(req.body)` leaks PII	security/smell
8	Inconsistent error response shape (text vs JSON)	quality

The seed codebase was approximately 65 LOC in `server.js` plus a small `db.js` and a partial public test suite (4 of 5 tests passing on the seed).

1.2 The five prompting styles

Style	Description	Length
`terse`	One sentence: "Fix the bugs in this repo."	~5 words
`brief`	Goal, context, constraints, success criteria — written as a colleague briefing	~150 words
`cot`	The brief, plus: "Before writing any code, list every issue you can find in priority order. Then fix them in that order."	~170 words
`tdd`	The brief, plus: "Run `npm test` first. For each failing test, fix the underlying bug. For each non-test issue you spot, write a failing test that pins the bug, then fix the bug."	~180 words
`spec`	A formal `api-spec.md` document with problem statement, in-scope/out-of-scope, and 8 acceptance criteria phrased as testable assertions. The prompt itself just says "Read api-spec.md and implement it."	~200 words

The styles were chosen to span a spectrum from minimal (terse) to fully-specified (spec), with two process-forcing variants (cot, tdd) in the middle.

1.3 Execution

Each run used a separate git worktree off an identical seed branch. Three waves of five parallel `claude -p --output-format json --permission-mode bypassPermissions` processes were dispatched. Token usage and final messages were captured per run. Bypass mode was used so that agents could freely run `npm test` and other shell commands during their session — a deliberate choice to ensure no style was disadvantaged by tooling restrictions.

1.4 Scoring rubric

Each run earned a composite score out of 100, weighted across five axes:

Axis	Weight	How
Correctness	40%	Hidden test suite (11 tests across the 8 bugs) run after the agent finished. Score = (passed / 11) × 40
Security	15%	Two binary checks: SQLi in `/users/search` removed, `console.log(req.body)` removed. Each worth 7.5
Restraint	15%	`git diff` stats. Penalty: 1 per file beyond 4, 1 per 50 LOC beyond 100
Quality	20%	Two independent LLM judges read anonymized diffs (labelled `agent_A`...`agent_O`) and scored 1–5. Final = avg × 4
Cost	10%	Relative to cheapest run in the study. `score = 10 × (min_tokens / this_run_tokens)`

A side metric, **calibration**, classified each run's final assistant message as `accurate`, `overconfident`, or `underclaimed` based on whether claims of "all bugs fixed" matched the hidden-test reality.

1.5 Blinding

Diffs were renamed to `agent_A` through `agent_O` (random shuffle) and stripped of run-id metadata before being shown to the LLM judges. The mapping was sealed in a separate file and only consulted after scoring.

---

2. Results

2.1 Leaderboard

Rank	Style	Mean	Median	Std Dev	Best	Worst
1	**spec**	**99.97**	99.97	**0.02**	100.00	99.94
2	cot	95.16	95.13	0.06	95.25	95.10
3	brief	92.75	91.69	1.67	95.11	91.46

Rank	Style	Mean	Median	Std Dev	Best	Worst
4	tdd	85.50	86.01	3.67	89.71	80.76
5	terse	78.22	75.08	4.47	84.54	75.04

Spec is nearly 5 points clear of second place and has effectively zero variance. Cot is now firmly second, ahead of brief — a meaningful change from earlier acceptEdits-mode results where the two were tied. TDD remains in fourth, terse in fifth.

2.2 Per-axis breakdown

Style	Correctness /40	Security /15	Restraint /15	Quality /20	Cost /10
terse	20.61	15.00	15.00	20.00	7.61
brief	35.15	15.00	15.00	20.00	7.60
cot	36.36	15.00	15.00	20.00	8.80
tdd	30.30	15.00	15.00	20.00	5.19
spec	**40.00**	15.00	15.00	20.00	**9.97**

Observations on the per-axis data:

- **Correctness is where styles separate.** Spec hit a perfect 40 on every run. CoT averaged 36.36 (≈10/11 tests passing on every run). Brief averaged 35.15. TDD trailed at 30.30 — actively *worse* than CoT and brief despite the extra process. Terse collapsed to 20.61.
- **Security is now flat across the board.** Every run, every style, fixed both the SQL injection and the PII log. With bypass mode allowing agents to run tests and verify their work, even terse caught the security issues consistently.
- **Restraint = 15 across the board.** No style over-engineered. The seed task was small enough that even verbose process couldn't bloat the diff.
- **Quality = 20 across the board.** Every blind judge scored every run 5/5. *This is a methodology problem, not a finding.* See §4.1.
- **Cost is dominated by spec at the top and crushed by TDD at the bottom.** Spec averaged 278k tokens; TDD averaged 821k — almost 3× more. Spec scored 9.97/10 on cost (essentially perfect; spec was the cheapest style in 14 of 15 individual comparisons).

2.3 Per-run details

All 15 runs, sorted by composite score:

Run ID	Duration	Tokens	Tests passed	SQLi fixed	Log fixed	Composite
spec_1	62s	278k	11	✓	✓	**100.00**
spec_2	60s	279k	11	✓	✓	**99.97**

Run ID	Duration	Tokens	Tests passed	SQLi fixed	Log fixed	Composite
spec_3	59s	280k	11	✓	✓	**99.94**
cot_3	74s	313k	10	✓	✓	95.25
cot_2	78s	317k	10	✓	✓	95.13
brief_2	85s	318k	10	✓	✓	95.11
cot_1	90s	318k	10	✓	✓	95.10
brief_3	106s	522k	10	✓	✓	91.69
brief_1	73s	319k	9	✓	✓	91.46
tdd_2	92s	398k	9	✓	✓	89.71
tdd_3	114s	402k	8	✓	✓	86.01
terse_2	67s	306k	7	✓	✓	84.54
tdd_1	208s	1.66M	8	✓	✓	80.76
terse_3	55s	403k	5	✓	✓	75.08
terse_1	65s	406k	5	✓	✓	75.04

Spec swept the top three positions. CoT swept the next three. The 7th-place run (brief_1) is the first non-spec/non-cot result. TDD's worst run (tdd_1) was also its most expensive — 1.66M tokens for an 8/11 result, the worst tokens-per-bug-fixed ratio in the study (208,000 tokens per passing test).

2.4 Calibration

Style	Accurate	Overconfident	Underclaimed
terse	3	0	0
brief	1	0	2
cot	0	0	**3**
tdd	3	0	0
spec	0	0	**3**

Two findings here:

1. **No run was overconfident.** No agent across any style claimed "all bugs fixed" when it hadn't. With full bash access, agents could verify their work and tended to report what they had actually done rather than make sweeping claims.
2. **CoT and spec were the most underclaimed styles.** All three spec runs and all three CoT runs fixed every bug in their style's typical working set, but none of them said so in the final message — they reported what they did factually, leaving the user to verify. This is good behavior. It just doesn't get rewarded by a calibration metric that scores "honesty" without rewarding accurate confidence.

The calibration metric as designed (regex-based detection of "all bugs/issues/fixed/works" phrasing) turned out to be a coarse signal — most agents simply did not use claim-everything language. See §4.2.

2.5 Cost efficiency

Style	Mean tokens	Mean tests passed	Tokens / test passed
spec	279k	11.00	**25,357**
cot	316k	10.00	31,624
brief	386k	9.67	39,970
terse	372k	5.67	65,586
tdd	821k	8.33	98,481

Spec is 3.9× more token-efficient than TDD on a per-bug-fixed basis. Even better, spec is now the cheapest style in absolute terms (279k mean tokens) while also being the most accurate — a result not seen in the previous (acceptEdits-mode) run where brief and spec were close on cost.

---

3. Findings

3.1 Spec-driven dominates because the work is done at write-time

The spec prompt was longer than the others (the formal `api-spec.md` was ~200 words of structured prose). But that work was done once, by a human, before the study ran. The agent didn't have to discover the requirements — it received them as enumerated, testable assertions.

The other styles required the agent to *infer* what counts as a bug. Brief gave hints ("security issues", "consistent error shape"). CoT and TDD added process around the inference. Terse left it entirely to the model. The further from a formal spec, the more bugs the model missed.

This is consistent with what specs are actually for in human engineering: the most expensive part of fixing a bug is usually identifying that it's a bug. Once it's named, the fix is mechanical. Spec-driven prompting front-loads the naming, which is why it's both the most accurate AND the cheapest — no tokens wasted on rediscovery.

3.2 TDD's process tax doesn't pay off here, even with full bash access

This was the biggest surprise. The earlier acceptEdits-mode study suggested TDD might be unfairly disadvantaged by being unable to run tests. With bypass mode, agents could `npm test`, write new test files, and iterate freely.

TDD did not improve. In fact, it scored slightly *lower* (85.50 vs 86.63 in the earlier run), and one run consumed 1.66M tokens — the most expensive single run in the study by a wide margin. The TDD prompt instructed the agent to write a failing test for each non-test issue before fixing it. In transcripts, this manifested as the agent writing additional test files, running them iteratively, and producing longer reasoning traces. None of that work translated into more bugs fixed.

For 8 well-defined bugs in 65 LOC, that overhead found nothing extra. TDD might pay off in larger codebases where bug-pinning is genuinely hard, but on a contained surface it was pure cost. The "TDD was disadvantaged" hypothesis from the earlier study is empirically refuted: TDD is genuinely a worse fit for this task type.

3.3 Chain-of-thought meaningfully outperformed brief

Unlike the previous study (where brief and CoT were tied), CoT now leads brief by 2.4 points (95.16 vs 92.75) and is more consistent (std dev 0.06 vs 1.67). Looking at per-run data:

- All three CoT runs landed at exactly 10/11 hidden tests passing
- Brief had one run at 9/11 (brief_1) and two at 10/11
- CoT runs averaged 316k tokens vs brief's 386k — CoT was ALSO cheaper

The "list issues ranked by severity, then fix in order" instruction appears to genuinely help on this task. Why might the gap have widened in bypass mode? With full bash, the agent's prioritization can be informed by running tests and seeing real failures, making the ranked list more grounded. In acceptEdits mode the agent had to reason about priority abstractly; that abstraction lost some of the value.

CoT is now the recommended default for non-spec'd work — better than brief on every axis except absolute simplicity of the prompt itself.

3.4 Terse has high variance — sometimes good, sometimes bad

Terse's std dev (4.47) was again the highest of any style, and the three runs scored 75.04, 75.08, and 84.54 — a 9-point spread. With brief, all three runs were within 4 points of each other. With spec, within 0.06.

When you give the model nothing, you get back whatever happens to be salient. terse_1 and terse_3 both fixed only 5/11 hidden tests; terse_2 fixed 7/11. The model chose what to address each time, and the choices weren't stable.

For experimentation or quick-and-dirty fixes, this might be fine. For anything where consistency matters across runs, terse is the wrong tool.

3.5 Bypass mode improved CoT and spec more than the others

Comparing the two studies:

Style	acceptEdits	bypass	Δ
spec	98.89	99.97	+1.08
cot	91.64	95.16	+3.52
brief	91.85	92.75	+0.90
tdd	86.63	85.50	–1.13
terse	76.70	78.22	+1.52

Bypass mode helped CoT most (+3.52) and spec second (+1.08, mostly from squeezing out the last 0.06 std dev). It did *not* help TDD, which actually got slightly worse — the additional bash freedom let TDD do more wasted work. Brief and terse barely moved.

The pattern suggests: prompts that give the agent a clear destination (spec) or a clear method (CoT) benefit from being able to verify their work. Prompts that prescribe heavyweight process (TDD) get more rope to hang themselves with. Prompts that lean on the model's defaults (brief, terse) don't gain much from extra tooling because they weren't doing systematic verification anyway.

---

4. Threats to validity

4.1 The quality axis didn't differentiate

Every run scored 5/5 from both blind judges. There are two possible explanations:

1. **The bug-fix task is too narrow** — there isn't enough creative latitude in fixing 8 small bugs for quality differences to emerge. Agents converge on similar diffs (parameterize the query, fix the offset, etc.).
2. **The LLM judge has an upward bias** — when given a 1–5 scale, "5" is the lowest-effort answer for a diff that doesn't have obvious problems.

Both are likely contributing. A future study should replace the absolute 1–5 rubric with **paired comparison**: show the judge two diffs and ask "which is better, A or B?" This produces a forced ordering and surfaces preferences the absolute scale collapses.

4.2 The calibration metric was too narrow

The regex looked for explicit "all bugs/issues fixed/works" phrasing. Most agents never used that phrasing, regardless of whether they actually fixed everything. As a result, the metric mostly measured *prose style* rather than *confidence calibration*. A better calibration metric would prompt the agent to give an explicit confidence number ("on a scale of 0–10, how confident are you that all issues are addressed?") and compare that to ground truth.

4.3 Three runs is descriptive, not significant

With n=3 per style, mean differences smaller than ~3 points are within plausible noise. The TDD vs terse rank (85.50 vs 78.22) is comfortable. The CoT vs brief gap (95.16 vs 92.75) is suggestive but not airtight. Spec's lead over CoT (4.81 points) is large enough to survive more runs given spec's near-zero variance, but a properly-powered study would want n≥10 per style.

4.4 One project, one task, one model

This is a study of Claude Opus 4.7 fixing bugs in a 65-LOC Express service. Generalization to:

- Other languages (Python, Rust, Go)
- Other task types (greenfield, refactor, performance optimization, debugging without an obvious goal)
- Other code sizes (>1000 LOC, multi-file, multi-package)
- Other models (Sonnet, Haiku, non-Anthropic models)

…is open. The relative ordering may be similar, but the magnitudes almost certainly aren't. Spec's advantage may shrink on greenfield tasks where there is no existing code to compare against. TDD's overhead may pay off on larger surfaces where bugs are hard to localize.

4.5 Cache reuse across runs

Each run's token count is dominated by `cache_read_input_tokens` (typically 90%+ of the total). Running the same prompt against the same codebase produces a lot of cache hits within a session. The cost-per-run includes those reads, which is realistic — but the absolute token numbers should not be read as "cost from cold." The first run of each style was likely more expensive than its peers if measured that way.

4.6 The cot prompt content is partially inside the brief

The CoT prompt is literally `brief.md` plus one extra instruction. Their results are not fully independent — if the brief content happens to be ill-suited, both styles will suffer together. A cleaner study would test CoT with prompt content distinct from brief, isolating the "think before coding" instruction from the surrounding context.

---

5. Conclusions

For the specific task tested — fixing eight bugs across security, correctness, performance, and quality categories in a small Express + SQLite API, using Claude Opus 4.7 with full bash access:

1. **Spec-driven prompting is the best approach across every dimension that matters.** Highest correctness (40/40 every run), lowest variance (std dev 0.02), cheapest in absolute tokens, never overconfident, fastest wall-clock per run.

2. **Chain-of-thought is now the strongest non-spec style.** Asking the agent to enumerate and rank issues before coding produced consistent 10/11 results at lower cost than a plain brief. CoT is the recommended default when a full spec is impractical.

3. **A detailed brief is solid but no longer the best non-spec choice.** Brief is still dramatically better than terse, but CoT beats it on every axis at lower cost.

4. **TDD-style prompting is overkill for a contained task — and even hurts.** It costs nearly 3× a spec or CoT for worse outcomes. Even with full ability to run the test suite, the methodological appeal does not survive contact with the data when the task is small. Reserve TDD for genuinely hard-to-pin bugs in larger codebases.

5. **Terse prompts ship inconsistent quality.** The same prompt produced a 75.04 and an 84.54 against the same codebase. If consistency matters, do not use terse.

6. **Process-forcing instructions are not all equivalent.** "Think first" (CoT) helped meaningfully. "Test first" (TDD) hurt. The cost of process scaffolding is real — only impose it when the work genuinely requires it.

The strongest practical recommendation: **the time you spend writing a real spec pays for itself in agent run-cost and result quality, often within a single execution.** A 200-word `api-spec.md` won this 15-run study by ~5 points and was the cheapest style at the same time. When a full spec isn't feasible, a CoT-augmented brief is the next best thing.