Don't Over-Engineer It: Why AI Coding Agents Write Too Much Code and One Sentence That Fixes It

Anonymous ACL submission

Abstract

AI coding agents routinely produce correct solutions that include far more code than necessary – adding unrequested tests, documentation, refactoring, and error handling beyond what the task requires. We formalize this phenomenon as *over-engineering* and introduce the Over-Engineering Score (OES), a family of metrics comparing agent patch size to reference solutions. Analyzing 3,000+ trajectories from 14 agents on SWE-bench Verified, we find that over-engineering is pervasive (mean OES between 1.2–8.2× reference size depending on the agent) and that a "Goldilocks zone" of 1–1.5× the reference patch yields the highest resolution rate (59.6% vs. 34.8% for patches >10×). We show that a single system prompt constraint, *"Make the minimal change needed"*, reduces over-engineering by 39% on average across 14 models from 7 providers on all 500 tasks, while reducing token usage by up to 42%. On SWE-agent, a production-grade framework, the same constraint increases task resolution from 50.0% to 62.6%. These results suggest that RLHF-induced verbosity is a key bottleneck in agent coding performance, addressable without additional training or architectural changes.

1 Introduction

A developer asks an AI coding agent to fix a two-line null check. The agent produces a 47-line patch: the fix, plus docstring updates, a new test file, type annotations on three unrelated functions, and a refactored import block. The patch is correct – it passes all tests – but it is *over-engineered*: it contains far more code than the task requires. Across 14 agents evaluated on SWE-bench Verified (Jimenez et al., 2024), we find that agents routinely produce patches 1.2–8.2× the size of reference solutions, with the most prolific agents generating diffs exceeding 100× the necessary changes.

We call this phenomenon *over-engineering* and argue it is a systematic failure mode, not a harmless quirk. Our analysis of 6,790 agent–task pairs reveals an inverted-U relationship between patch size and success: patches in a narrow "Goldilocks zone" of 1–1.5× the reference size achieve 59.6% resolution, while heavily over-engineered patches (>10×) resolve at only 34.8%. Each unnecessary change compounds failure risk, consistent with recent work showing that extraneous mutating actions reduce LLM-agent success odds by up to 92–96% on tool-use benchmarks (Cuadron et al., 2025).

Why do agents over-engineer? We hypothesize that this behavior stems from the **RLHF helpfulness paradox**: models are trained on human preference data where comprehensive, thorough responses score higher (Sharma et al., 2024). In conversational settings, this bias toward "doing more" is desirable. In software engineering, where the gold standard is the *minimal* patch that solves the problem, it produces unnecessary changes that compound failure risk.

We show that this bias can be counteracted with a remarkably simple intervention, a single sentence added to the agent's system prompt:

> *"Make the MINIMAL change needed to solve the problem. Do not add tests, documentation, type annotations, error handling, or refactoring unless explicitly requested."*

This 23-word constraint reduces over-engineering by 39% on average across 14 models from 7 providers ($p < 10^{-5}$, Wilcoxon signed-rank), while reducing token usage by up to 42% and improving resolve rate on 10 of 14 models (Figure 1). On SWE-agent (Yang et al., 2024), a production-grade framework, the same constraint increases task resolution from 50.0% to 62.6%. Just as "Let's think step by step" (Kojima et al., 2022) unlocked latent reasoning ability with a simple prompt, our

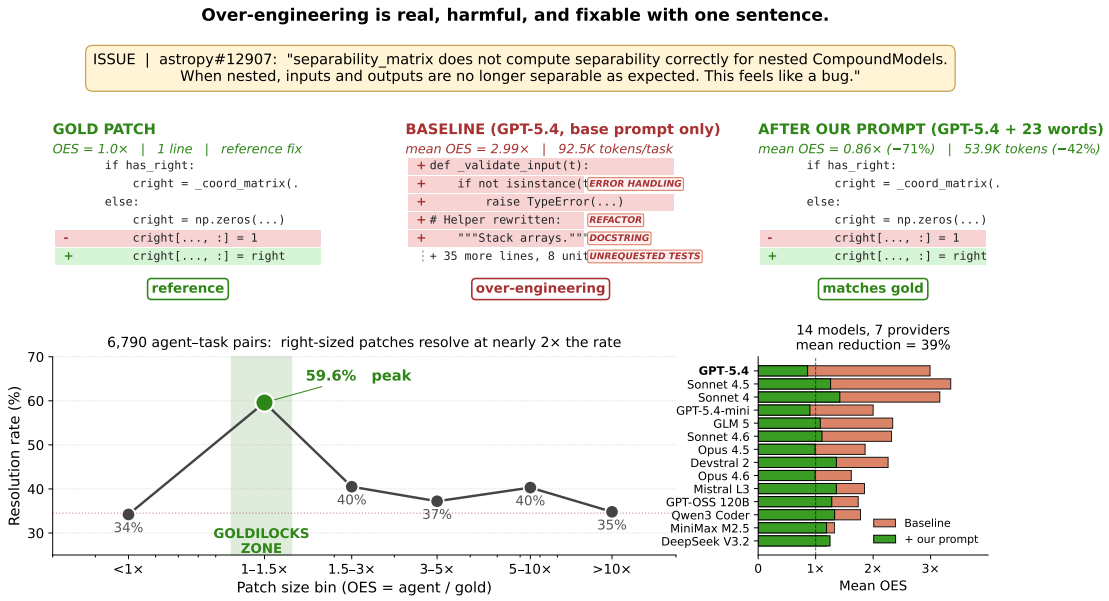

Figure 1: **Over-engineering is real, harmful, and fixable with one sentence.** *Top:* on a representative bug-fix task, the gold patch is one line; with the base prompt only, GPT-5.4 produces a 47-line patch adding unrequested tests, documentation, type annotations, refactoring, and error handling (mean OES = 2.99×, 92.5K input tokens/task). After prepending our 23-word constraint, GPT-5.4 matches the gold patch (mean OES = 0.86×, −71%; 53.9K tokens, −42%). *Bottom-left:* across 6,790 publicly-reported agent–task pairs, patches in the 1–1.5× "Goldilocks zone" resolve at 59.6%, vs. 34–35% for under-engineered (<1×) and extreme over-engineered (>10×) patches. *Bottom-right:* the same 23-word prompt reduces mean OES across 14 models from 7 providers (mean reduction 39%).

constraint unlocks latent *restraint*: the ability to do less when less is what the task demands.

In summary, this paper makes the following contributions:

1. We formalize the **Over-Engineering Score (OES)**, a metric that quantifies patch bloat by comparing agent diffs to reference solutions along line, file, and hunk dimensions (§3).

2. We conduct a **prevalence study across 14 agents** on SWE-bench Verified, revealing pervasive over-engineering and an inverted-U relationship between OES and task resolution, and propose a **prompt-level intervention** that reduces OES by 39% on average across 14 models with no additional training or architectural changes (§5).

2 Related Work

SWE Agents and Benchmarks. SWE-bench (Jimenez et al., 2024) established the standard evaluation for coding agents by collecting real GitHub issues with executable test suites. SWE-agent (Yang et al., 2024) introduced agent–computer interfaces that enable models to navigate repositories and edit files. Agentless (Xia et al., 2024) demonstrated that a simple two-phase pipeline (localize, then repair) can match complex agentic systems, providing early evidence that *less* agent autonomy can yield *more* reliable patches. OpenHands (Wang et al., 2025) unified these approaches into an open evaluation platform. Our work complements this ecosystem by identifying a systematic quality problem, over-engineering, that persists across all agent architectures, including Agentless.

Agent Failure Analysis. Several recent papers analyze why coding agents fail. Majgaonkar et al. (2025) study success and failure trajectories across agents, finding that failed runs are longer and higher-variance, consistent with our finding that scope inflation predicts failure. Xue et al. (2025) propose a six-category taxonomy of LLM coding-agent failures and build a static-analysis-plus-LLM patching system to repair them; their fix requires program analysis machinery, while ours requires 23 words in the system prompt. SABER (Cuadron et al., 2025) shows that deviations in mutating (environment-changing) actions reduce LLM-agent success odds by up to 92–96% on tool-use benchmarks, supporting the broader claim that extra ac-

tions are not free. These works diagnose the problem; we provide a fix.

Error Recovery and Self-Repair. Reflexion (Shinn et al., 2023) enables agents to improve through verbal self-reflection after failures. Huang et al. (2024) show that LLMs struggle to self-correct without external feedback, motivating explicit constraints rather than relying on agents to self-regulate. Our intervention can be viewed as *proactive Reflexion*: embedding behavioral constraints before execution rather than reflecting on errors after the fact.

Prompt-Level Behavioral Interventions. Chain-of-thought prompting (Wei et al., 2022) and its zero-shot variant "Let's think step by step" (Kojima et al., 2022) demonstrated that simple prompt modifications can dramatically alter model behavior without retraining. Constitutional AI (Bai et al., 2022) showed that rule-based principles can guide model behavior at inference time. Plan-and-Solve (Wang et al., 2023) extended prompting to multi-step planning. SWE-Bench Mobile (Tian et al., 2026) found that simple "defensive programming" prompts outperform complex instructions by 7.4% on mobile engineering tasks. Our work extends this paradigm from reasoning to engineering discipline: where CoT unlocks latent reasoning, our constraint unlocks latent restraint.

RLHF and Over-Helpfulness. Sharma et al. (2024) establish that sycophancy, producing outputs that flatter rather than inform, is a pervasive artifact of RLHF training. Over-engineering is sycophancy in the coding domain: the model produces *more* code because "more" scores higher in preference training, even when the task demands surgical precision. Our intervention directly counteracts this bias.

3 The Over-Engineering Problem

3.1 Definition

We define over-engineering in the context of AI coding agents as follows: ***An agent over-engineers when its patch includes code changes beyond the minimal set required to resolve the task.*** This definition is intentionally scope-neutral: it applies whether the extra changes are tests, documentation, refactoring, or any other modification not required by the task specification. Crucially, over-engineering can co-occur with correctness – an agent may produce a patch that passes all tests while containing substantial unnecessary code.

Metric	What it measures
Size: *how much extra code?*	
$\text{OES}_{\text{lines}}$	Total diff lines (agent / gold). 1.0 = exact match.
OES_{code}	Same as $\text{OES}_{\text{lines}}$, restricted to Python source (no docs/comments/blank).
Location: *where does the code go?*	
$\text{OES}_{\text{scope}}$	Agent edits within gold-patch files (in-scope bloat).
$\text{OES}_{\text{creep}}$	Agent edits to files *not* in gold patch (scope creep).
$\text{OES}_{\text{focus}}$	Fraction of agent edits that land in gold-patch files (file-level on-target rate).
Accuracy: *are the edits on target?*	
$\text{OES}_{\text{precision}}$	Fraction of agent-modified lines that match gold line numbers.
$\text{OES}_{\text{recall}}$	Fraction of gold-patch lines covered by agent edits.
Readability proxy	
$\text{OES}_{\text{comments}}$	Mean #-prefixed lines added per patch (gold = 0.76).

Table 1: The OES metric family. All metrics are computed by comparing the agent's diff to the SWE-bench Verified gold patch.

This distinguishes over-engineering from outright failure and makes it invisible to pass/fail metrics like task resolution rate.

3.2 The Over-Engineering Score (OES)

To quantify over-engineering, we introduce the **Over-Engineering Score** (OES), a family of metrics that compare agent-generated patches to reference (gold) patches. The family is organized around three questions: *how much* extra code, *where* is it, and *how accurate* is the placement. Table 1 provides a quick reference.

3.2.1 Size Metrics

Line-level OES. For a task t with agent patch P^a and gold patch P^g:

$$\text{OES}_{\text{lines}}(t) = \frac{|L^a_+| + |L^a_-|}{|L^g_+| + |L^g_-|} \quad (1)$$

where L^a_+ and L^a_- denote lines added and removed by the agent, and L^g_+ and L^g_- by the gold patch. An OES of 1.0 is optimal; >1 indicates over-engineering; <1 indicates under-engineering.

Code-only OES. To control for non-functional bloat:

$$\text{OES}_{\text{code}}(t) = \frac{|C^a_+| + |C^a_-|}{|C^g_+| + |C^g_-|} \quad (2)$$

where C^a and C^g count only *functional code lines* in Python source files (`.py`), excluding blank lines,

comments (#), and docstrings (`"""`). Since all SWE-bench gold patches modify only `.py` files, non-Python files in agent patches (e.g., build artifacts, certificate bundles) are extraneous by definition.

3.2.2 Location and Accuracy Metrics

We additionally define location metrics ($\text{OES}_{\text{scope}}$, $\text{OES}_{\text{creep}}$, $\text{OES}_{\text{focus}}$) that decompose $\text{OES}_{\text{lines}}$ into within-file vs. wrong-file bloat, and accuracy metrics ($\text{OES}_{\text{precision}}$, $\text{OES}_{\text{recall}}$) that measure line-level overlap between agent and gold patches. By construction, $\text{OES}_{\text{lines}} = \text{OES}_{\text{scope}} + \text{OES}_{\text{creep}}$ per task (see Appendix C for behavior under means). We also report $\text{OES}_{\text{comments}}$, the mean count of #-prefixed lines added per patch, as a structural readability proxy. Formal equations are deferred to Appendix C.

3.2.3 Aggregation

OES distributions are heavily right-skewed: a small number of extreme cases (5–50× gold) inflate the mean while most tasks sit near OES ≈ 1. We use *both* aggregates depending on the question. For *prevalence* across heterogeneous public agents (§3.3), where outliers can modify non-code files (SSL bundles, build artifacts) and contaminate the mean, we report the median:

$$\text{OES}_{\text{agg}} = \text{median}\big(\{\text{OES}_{\text{lines}}(t_i) \mid t_i \in T_{\text{sub}}\}\big) \quad (3)$$

For *paired condition comparisons* (`C0` vs. `C1`; Tables 5 and 7), we report the mean because the intervention's effect concentrates in the right tail (catching the 5–50× outliers); for 11 of 14 models the median OES is already ≈ 1 at baseline, so the median would mask the very mechanism the intervention exploits. Significance tests use the paired Wilcoxon signed-rank statistic (Appendix A.6), which is rank-based and therefore robust to either choice of central tendency.

We compute OES on every *submitted* (non-empty) patch, regardless of resolution, because the inverted-U analysis (§3.3) requires patch-size measurements for both resolved and unresolved tasks. Empty patches are excluded and reported separately as a distinct under-engineering failure mode.

3.3 Prevalence in the Wild

To establish that over-engineering is not a hypothetical concern, we analyze publicly available trajectories from 14 agents on SWE-bench Verified, computing $\text{OES}_{\text{lines}}$ for all resolved patches (Table 2).

Agent	N	Res%	Med.	Mean
SWE-agent (3.7 Son.)	497	62.8	1.27	6.18
OpenHands (Sonnet)	494	53.6	5.60	365.91
AutoCodeRover (Son.)	496	46.6	0.97	1.40
Honeycomb	442	45.9	2.85	10.77
Solver	498	45.6	1.82	5.16
Gru	499	45.3	0.75	2.00
Tools (Haiku)	484	41.9	8.20	118.71
Composio SweKit	500	40.6	1.50	6.55
Agentless (GPT-4o)	496	39.1	0.67	0.98
SWE-agent (3.5 Son.)	489	34.4	6.50	27.90
MASAI (GPT-4o)	478	34.1	1.15	1.97
SWE-agent (GPT-4o)	450	25.8	2.00	48.93
Lingma (72B)	485	25.8	1.22	3.35
EPAM AI Run (GPT-4o)	482	24.9	1.50	23.32

Table 2: $\text{OES}_{\text{lines}}$ across 14 agents on SWE-bench Verified, sorted by resolution rate. Med. = median, Mean = arithmetic mean. Distributions are heavily right-skewed: OpenHands' mean of 365.91× reflects extreme outliers where patches modify non-code files (e.g., SSL certificate bundles). These are publicly available trajectories from the SWE-bench experiments repository; we compute OES from the published patches without re-running any agent.

Over-engineering is pervasive: 11 of 14 agents have a median OES above 1.0, meaning more than half of their resolved patches are larger than the reference solution. The three agents with the highest median OES (Tools/Haiku at 8.20×, SWE-agent/3.5 Sonnet at 6.50×, OpenHands at 5.60×) all use agentic architectures with extended tool-use loops. Note that these agents combine different *frameworks* (SWE-agent, OpenHands, Agentless, etc.) with different *models* (Claude, GPT-4o, Llama); over-engineering is a function of both. A small number of extreme outliers (>100×, 91 patches or 1.3% of the dataset) arise from agents modifying non-Python files such as SSL certificate bundles; we retain these in our analysis but report medians as the primary statistic to limit their influence.

The Goldilocks zone. Stratifying all 6,790 task–agent observations by OES range reveals an inverted-U relationship between patch scope and task success (Table 3).

Patches in the 1–1.5× range achieve 59.6% resolution – nearly double the rate of both minimal patches (<1×, 34.2%) and extreme over-engineering (>10×, 34.8%). Under-engineering is as harmful as over-engineering, suggesting the optimal strategy is neither maximal minimality nor

OES Range	N	Resolved	Rate
Minimal (<1×)	2,113	722	34.2%
Slight (1–1.5×)	1,168	696	**59.6%**
Moderate (1.5–3×)	1,022	414	40.5%
Heavy (3–5×)	693	258	37.2%
Severe (5–10×)	760	306	40.3%
Extreme (>10×)	1,034	360	34.8%

Table 3: Resolution rate by OES range, pooled across 14 agents ($N = 6{,}790$ submitted non-empty patches; 2,756 resolved, 4,034 unresolved). OES is computed for every submitted patch against the same gold reference, regardless of resolution outcome. Patches in the 1–1.5× range achieve the highest resolution rate – nearly double that of extreme over-engineering (>10×).

maximal thoroughness, but a narrow window of *right-sized* change. This Goldilocks zone motivates our intervention: if agents can be guided to produce patches of appropriate scope, success rates should improve.

3.4 Taxonomy and Root Cause

We additionally develop a six-category taxonomy of over-engineering behaviors (unsolicited tests, solution rewriting, scope expansion, defensive code, documentation, cosmetic changes) derived from open-coded LLM-as-judge labeling on 500 over-engineered `C0` patches. We defer the full taxonomy and per-category statistics to Appendix E. We hypothesize that over-engineering arises from an *RLHF helpfulness paradox* – training pressure toward thorough, comprehensive responses that conflicts with the surgical precision coding tasks require. We discuss this mechanism in detail in Appendix H.

4 Intervention Design

4.1 Design Principles

Our intervention strategy is guided by two constraints: (1) *zero training*, no fine-tuning, no architectural changes, no additional compute; and (2) *system prompt only*, the intervention is a text string prepended to the agent's existing system prompt.

These constraints ensure that any effective intervention can be adopted immediately by any practitioner using any model. The inspiration is "Let's think step by step" (Kojima et al., 2022): a single phrase that unlocks latent capability. We hypothesize that a similar phrase can unlock latent *restraint*.

ID	Label	Words	Prompt Text
`C0`	Baseline	0	*(no modification)*
`C1`	Explicit	23	"Make the MINIMAL change needed to solve the problem. Do not add tests, documentation, type annotations, error handling, or refactoring unless explicitly requested."
`C2`	Readable	32	`C1` text + "Add comments and docstrings where necessary for human readability."

Table 4: Intervention conditions. `C0` is the unmodified baseline. `C1` is the primary minimality constraint. `C2` extends `C1` with a readability instruction.

4.2 Conditions

We evaluate three conditions: a no-intervention baseline, the primary minimality constraint, and a readability extension. `C1` (23 words) combines a positive directive ("make the minimal change") with explicit prohibitions targeting the top over-engineering categories from our taxonomy (T1–T6; Table 9). `C2` extends `C1` by adding a readability instruction; its design rationale and results are deferred to Appendix G.

5 Results

Setup. We evaluate on SWE-bench Verified (Jimenez et al., 2024): 500 real-world GitHub issues, expert-written gold patches (median 7 lines, 85.8% single-file). We test 14 LLMs spanning 7 providers – Anthropic Claude (Anthropic, 2026), OpenAI GPT-5.4 (OpenAI, 2026), Mistral Large 3 / Devstral 2 (Mistral, 2026), Z.AI GLM 5 (Z.ai, 2026), Qwen3 Coder (Qwen, 2026), MiniMax M2.5 (MiniMax, 2026), DeepSeek V3.2 (DeepSeek, 2026) – accessed through AWS Bedrock except OpenAI models which use the OpenAI API directly (Table 5 lists exact identifiers). Each model runs in a lightweight 6-tool agent (`FindFile`, `ViewCode`, `SearchCode`, `EditFile`, `CreateFile`, `Finish`; no shell execution) with temperature = 0, 20-tool-call budget, 10-minute timeout; the system prompt is the only variable between conditions. Our primary comparison is `C0` (unmodified system prompt) vs. `C1` (the 23-word constraint, §4.2); we additionally evaluate a readability extension `C2` (Appendix G). We report $\text{OES}_{\text{lines}}$ (primary), Patch rate, Resolved rate, and token usage. Significance: Wilcoxon signed-rank (Wilcoxon, 1992) for paired OES, McNemar's test (McNemar, 1947) for patch

rate, Bonferroni-corrected $\alpha = 0.05/14 \approx 0.0036$ across 14 models. The framework is deliberately simpler than SWE-agent (Yang et al., 2024) or OpenHands (Wang et al., 2025), isolating prompt effects from scaffold complexity; absolute resolve rates are correspondingly lower (16–62%), but *relative* `C0`–`C1` differences are the contribution. Full details (benchmark composition, prompt templates, model IDs, framework justification, statistical-test rationale) are deferred to Appendix A.

5.1 Intervention Results

The prevalence and Goldilocks zone analyses (§3.3) establish that over-engineering is widespread and predicts task failure. We now test whether the `C1` prompt fixes it. We execute each model via our lightweight agent framework (§A.3) on all 500 SWE-bench Verified tasks under both `C0` and `C1` conditions. The *only* variable between conditions is the system prompt text. Table 5 presents the headline results across 14 models from 7 providers; the full OES decomposition is deferred to Appendix D.1.

OES decomposition. The most revealing finding comes from decomposing $\text{OES}_{\text{lines}}$ into within-file changes ($\text{OES}_{\text{scope}}$) and wrong-file changes ($\text{OES}_{\text{creep}}$). For the top 5 models, $\text{OES}_{\text{creep}}$ drops by 89–96% under `C1` (Table 7, $\text{OES}_{\text{creep}}$ column), while $\text{OES}_{\text{scope}}$ changes only modestly (-7% to -20%). This reveals the intervention's primary mechanism: *it does not make agents write less code in the right files – it stops them from wandering into wrong files*. The intervention is a focus constraint, not a brevity constraint.

Code-only confirmation. OES_{code} (Python source only, no docs/comments/blank) shows attenuated but consistent reductions (1–56%), confirming the bloat is genuine code, not just documentation. We note that the median OES_{code} is near 1.0 for both conditions across most models, indicating the intervention primarily catches extreme tail cases (5–50×) rather than shifting the typical task. This "tail guardrail" behavior is consistent with the scope decomposition: most tasks are already well-scoped, but a minority spiral into wrong-file work ($\text{OES}_{\text{creep}}$), and the intervention prevents those spirals.

Line-level precision. Table 7 provides the finest-grained validation: under `C1`, agents edit more of the right source lines ($\text{OES}_{\text{precision}}$ up to +.23, with most models in the +.07 to +.16 range) while maintaining coverage of the fix ($\text{OES}_{\text{recall}}$ $\sim$0.5). $\text{OES}_{\text{focus}}$ (file-level targeting) increases from $\sim$82% on average to $\sim$94% – agents shift from spending $\sim$18% of their effort on irrelevant files to $\sim$6%.

Patch rates and resolve rates. Patch rates are maintained or improved for 12 of 14 models. Resolution rate (resolved/500) increases for 10 of 14 models, with the largest gains on Opus 4.5 ($+25.5\%$, $247 \rightarrow 310$), DeepSeek V3.2 ($+15.0\%$), and Sonnet 4 ($+12.5\%$, $144 \rightarrow 162$). The four exceptions are GPT-OSS 120B (-21.5%), GPT-5.4 (-16.2%), GPT-5.4-mini (-6.6%), and Sonnet 4.6 (-6.1%); see §5.2.

Token savings. The intervention reduces input token consumption for 13 of 14 models, with savings ranging from 1% (DeepSeek V3.2) to 42% (GPT-5.4). Claude Sonnet 4.6 saves 29% of input tokens under `C1`, translating directly to cost reduction for practitioners running agents at scale.

5.2 Cross-Model Patterns

The 14 models span 7 providers with diverse architectures, parameter counts, and training approaches. Several patterns emerge.

Size-tier analysis. Binning models by total parameter count using estimates from Li (2026) reveals a strong size-dependent pattern (Table 6). Large models (>1T params; the Anthropic Sonnet/Opus families) over-engineer most at baseline (mean $\text{OES}_{\text{lines}} = 2.46\times$) and benefit most from the intervention (-51%, -19% tokens, +18 resolved tasks). Medium models (200B–1T) start lower ($1.95\times$) and reduce -35%. Small models (<200B) reduce least (-31%) and gain little on tokens (-3%) or resolution (-7 tasks). The pattern is consistent with the RLHF helpfulness paradox (Appendix H): larger models have undergone more preference-tuning toward thoroughness and therefore have more bias to correct.

When minimality costs correctness. The three OpenAI-trained models account for the largest resolve-rate regressions (GPT-5.4 -7.6 pp, GPT-OSS 120B -4.2 pp, GPT-5.4-mini -2.6 pp); Sonnet 4.6 (-3.2 pp) is the lone non-OpenAI exception. The shared mechanism is increased empty-patch rates – the constraint becomes a brittleness amplifier on models that already write small patches (Appendix I.4). The intervention nonetheless generalizes across all 7 providers and across general-purpose and code-specialized models (Appendix I.5); qualitative case studies and full `C2` results are in Appendix F and G.

Model	OES_{lines} C0	OES_{lines} C1	OES_{lines} Δ	Patch% C0	Patch% C1	Patch% Δ	Resolved % C0	Resolved % C1	Resolved % Δ	Tokens (k) C0	Tokens (k) C1	Tokens (k) Δ
Anthropic												
Sonnet 4.5	3.35	1.26	↓62%	74	93	↑19	43.4 (217)	46.0 (230)	↑6.0%	177	164	↓7%
Sonnet 4	3.16	1.42	↓55%	74	91	↑17	28.8 (144)	32.4 (162)	↑12.5%	159	145	↓9%
Sonnet 4.6	2.32	1.11	↓52%	74	94	↑20	52.2 (261)	49.0 (245)	↓6.1%	152	107	↓29%
Opus 4.5	1.86	0.99	↓47%	81	93	↑12	49.4 (247)	62.0 (310)	↑25.5%	143	110	↓23%
Opus 4.6	1.62	0.99	↓39%	85	94	↑9	44.0 (220)	45.6 (228)	↑3.6%	109	80	↓27%
Z.AI												
GLM 5	2.34	1.08	↓54%	70	94	↑24	38.4 (192)	39.2 (196)	↑2.1%	140	108	↓23%
Mistral												
Devstral 2	2.26	1.36	↓37%	82	93	↑11	36.0 (180)	36.2 (181)	↑0.6%	139	127	↓9%
Mistral L3	1.85	1.36	↓26%	86	90	↑4	27.6 (138)	28.2 (141)	↑2.2%	81	78	↓4%
OpenAI												
GPT-OSS 120B	1.74	1.28	↓18%	94	88	↓6	21.4 (107)	16.8 (84)	↓21.5%	76	78	↑3%
GPT-5.4	2.99	0.86	↓71%	51	100	↑49	47.0 (235)	39.4 (197)	↓16.2%	92	54	↓42%
GPT-5.4-mini	2.00	0.90	↓55%	100	100	→0	39.6 (198)	37.0 (185)	↓6.6%	90	64	↓29%
Qwen												
Qwen3 Coder	1.78	1.33	↓19%	94	95	↑1	19.4 (97)	19.6 (98)	↑1.0%	171	165	↓4%
MiniMax												
MiniMax M2.5	1.33	1.19	↓10%	91	93	↑2	41.4 (207)	42.0 (210)	↑1.4%	117	109	↓7%
DeepSeek												
DeepSeek V3.2	1.19	1.25	↑5%	96	93	↓3	16.0 (80)	18.4 (92)	↑15.0%	134	133	↓1%

Table 5: `C0` (baseline) vs. `C1` (minimality) across 14 models on SWE-bench Verified (500 tasks each). OES_{lines} and Tokens are means over submitted patches (the intervention's effect concentrates in the right tail, which the median would mask; see §3.2). Patch% is the fraction of all 500 tasks for which the agent produced a non-empty diff. **Resolved %** is resolved / 500 (raw resolved count in parentheses); Δ is the relative change from `C0` (e.g., 192→196 = +2.1%). Green = improvement, red = regression. The intervention reduces mean OES for 13/14 models, maintains/improves Patch% for 12/14, increases resolution rate for 10/14, and reduces tokens for 13/14. The headline "39% mean OES reduction" reported throughout the paper is the mean of per-model relative reductions (i.e., $\frac{1}{14}\sum_i(\mathrm{OES}_i^{C0} - \mathrm{OES}_i^{C1})/\mathrm{OES}_i^{C0}$). Full decomposition ($OES_{code/creep/scope/precision/recall/focus}$) in Appendix D.1.

Tier	n	OES_{lines} C0	OES_{lines} C1	OES_{code} C0	OES_{code} C1	Tok Δ
Large (>1T)	5	2.46	1.15	2.11	1.12	−19%
Medium (200B–1T)	6	1.95	1.11	1.45	1.01	−18%
Small (<200B)	3	1.93	1.32	1.41	1.23	−3%

Table 6: Per-tier means. *Large* = Sonnet 4/4.5/4.6, Opus 4.5/4.6; *Medium* = GPT-5.4, GLM 5, Mistral L3, DeepSeek V3.2, GPT-5.4-mini, MiniMax M2.5; *Small* = Devstral 2, GPT-OSS 120B, Qwen3 Coder. Parameter estimates from Li (2026). Larger models over-engineer more and benefit more from the intervention.

6 Analysis

6.1 The Intervention as a Focus Constraint

The metric decomposition reveals a precise mechanism. OES_{scope} (changes within gold-patch files) barely moves under `C1` (−7% to −20%), while for the top 5 models OES_{creep} (changes to wrong files) drops by 89–96%. The intervention does not make agents write less code in the right files – it stops them from *wandering*. This is corroborated by line-level precision (Table 7): agents under `C1` edit more of the correct source lines ($OES_{precision}$ up to +.23, with most models in the +.07 to +.16 range) with no loss of OES_{recall} (∼0.5), and OES_{focus} shifts from ∼82% of effort in-scope on average to ∼94%.

OES_{code} provides a complementary check: after restricting to Python source files and stripping docstrings, comments, and blank lines, the intervention still reduces functional code bloat by 1–56%. The attenuation from OES_{lines} to OES_{code} (typically 15–35% relative) indicates that roughly two-thirds of the reduction is genuine functional code. We note that median OES_{code} is near 1.0 for both conditions across most models, meaning the intervention primarily catches extreme over-engineering tails (5–50×) rather than shifting the typical task – a "tail guardrail" consistent with the OES_{creep} mechanism. Extended discussion of why the intervention works, the inverted-U revisited, and cross-architecture generalization is provided in Appendix I.

6.2 Practical Implications

The intervention's simplicity is its primary contribution. A practitioner can copy one sentence into their agent's system prompt and observe immediate improvement – no retraining, no fine-tuning, no

infrastructure changes. The token savings (up to 42% input reduction) translate directly to cost reduction for teams running agents at scale via APIs. For Claude Opus 4.6, one of the more expensive models in our evaluation, the 27% input token reduction represents a substantial budget saving with *improved* output quality.

Task adherence, not code quality. We define over-engineering relative to the task specification, not absolute code quality. An agent that adds helpful tests or documentation is producing higher-quality code in isolation, but is over-engineering relative to a task that says "fix this bug." To verify this framing, we used Claude Opus 4.6 as an LLM judge to classify all 500 SWE-bench Verified problem statements by whether they explicitly request (a) tests, (b) documentation, (c) error handling, (d) type annotations, or (e) refactoring. Of 500 tasks, 492 (98.4%) request only a bug fix or feature implementation; just 2 request tests, 1 requests documentation, 1 requests error handling, and 5 request refactoring. When agents add tests or documentation on the remaining 492 tasks, it is always unsolicited. Our intervention targets *task adherence* – doing what was asked, not less and not more. The constraint explicitly preserves optionality via "unless explicitly requested," so the 8 tasks that do request additional work are unaffected. The inverted-U relationship (Table 3) provides empirical justification: unsolicited additions do not merely waste tokens – they actively reduce task resolution rates.

More broadly, the intervention could be integrated into agent frameworks as a default system prompt component. Just as modern agents ship with tool-use instructions and safety guidelines, they could ship with minimality constraints – a zero-cost default that reduces over-engineering across all tasks. The fact that patch rates are maintained or improved under `C1` (Table 5) means this default imposes no engagement penalty.

7 Conclusion

Over-engineering is pervasive, harmful, and fixable. Our analysis of 14 agents on SWE-bench Verified reveals that agents routinely produce patches 1.2–8.2× the size of reference solutions, and that this patch inflation follows an inverted-U relationship with task success: both under-engineering and over-engineering resolve at roughly half the rate of right-sized patches.

We attribute this behavior to RLHF helpfulness bias – the same training signal that makes chatbots thorough may make coding agents reckless – and show that a 23-word system prompt constraint reduces over-engineering by an average of 39% across 14 models from 7 providers, with $p < 10^{-5}$ on the primary models tested. The intervention simultaneously reduces token usage by up to 42%, maintains or improves patch rates for 12 of 14 models, and improves resolution rate for 10 of 14 models (largest gains: Opus 4.5 $+25.5\%$, DeepSeek V3.2 $+15.0\%$, Sonnet 4 $+12.5\%$); the four model-specific regressions are documented in §5.2. It requires zero training, zero architectural changes, and zero additional compute.

The OES metric suite, intervention prompts, agent runner, and all analysis code will be released upon publication. Several directions remain open: (1) evaluation across additional model families as tool-use API support broadens; (2) training-time fixes that use preference data explicitly rewarding minimal patches, complementing our inference-time approach; (3) integration of minimality constraints as defaults in agent frameworks such as SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2025); and (4) extension to other behavioral failure modes – verbatim retry, out-of-scope writing, insufficient verification – where simple prompt-level interventions may prove equally effective.

The broader lesson is one of economy. In software engineering, the best patch is the smallest one that solves the problem. AI coding agents have not yet learned this – but a single sentence can teach them.

Limitations

The OES metric requires gold reference patches and is therefore limited to benchmarks that provide expert-written solutions; it cannot be computed in open-ended deployment settings where no reference exists. Some candidate models could not be included due to incompatibility with our agent framework's tool-use protocol (DeepSeek R1, GLM 4.7, Llama 4 Maverick, Kimi K2.5). All experiments use a custom lightweight agent with six tools (`FindFile`, `ViewCode`, `SearchCode`, `EditFile`, `CreateFile`, `Finish`); results may differ under alternative scaffolds such as SWE-agent (Yang et al., 2024) or OpenHands (Wang et al., 2025), which impose their own tool inter-

faces and prompting strategies. While we report resolve rates from the SWE-bench Docker evaluation harness, infrastructure-related evaluation failures (Docker Hub rate limits) reduced statistical power for some resolve rate comparisons. Finally, all evaluation is conducted on a single benchmark (SWE-bench Verified, Python repositories only); generalization to other languages, task types, and benchmark suites remains untested.

Ethics Statement

All experiments use publicly available benchmark data from SWE-bench Verified (Jimenez et al., 2024); no human subjects are involved and no private data is accessed. Our interventions constrain agent behavior toward minimal, focused changes – a defensive measure that reduces unnecessary code modifications rather than enabling harmful ones. All prompts, metrics, and analysis code will be released for full reproducibility.

References

Anthropic. 2026. Claude. https://claude.com/product/overview. Accessed: 2026-05-25.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Alejandro Cuadron, Pengfei Yu, Yang Liu, and Arpit Gupta. 2025. Saber: Small actions, big errors–safeguarding mutating steps in llm agents. *arXiv preprint arXiv:2512.07850*.

DeepSeek. 2026. DeepSeek V3.2. https://api-docs.deepseek.com/news/news251201. Accessed: 2026-05-25.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *International conference on learning representations*, volume 2024, pages 32808–32824.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Bojie Li. 2026. Incompressible knowledge probes: Estimating black-box llm parameter counts via factual capacity. *arXiv preprint arXiv:2604.24827*.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. 2025. Understanding code agent behaviour: An empirical study of success and failure trajectories. *arXiv preprint arXiv:2511.00197*.

Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157.

MiniMax. 2026. MiniMax M2.5. https://www.minimax.io/models/text. Accessed: 2026-05-25.

Mistral. 2026. Devstral 2. https://mistral.ai/news/devstral-2-vibe-cli. Accessed: 2026-05-25.

OpenAI. 2026. GPT-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-05-25.

Qwen. 2026. Qwen3 coder. https://qwen.ai/blog?id=qwen3-coder. Accessed: 2026-05-25.

Mrinank Sharma, Meg Tong, Tomek Korbak, David Duvenaud, Amanda Askell, Sam Bowman, Esin Durmus, Zac Hatfield-Dodds, Scott Johnston, Shauna Kravec, and 1 others. 2024. Towards understanding sycophancy in language models. In *International Conference on Learning Representations*, volume 2024, pages 110–144.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Muxin Tian, Zhe Wang, Blair Yang, Zhenwei Tang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. 2026. Swe-bench mobile: Can large language model agents develop industry-level mobile applications? *arXiv preprint arXiv:2602.09540*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*, pages 196–202. Springer.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Haoran Xue, Gias Uddin, and Song Wang. 2025. Pagent: Learning to patch software engineering agents. *arXiv preprint arXiv:2506.17772*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Z.ai. 2026. GLM-5. https://z.ai/blog/glm-5. Accessed: 2026-05-25.

A Full Experimental Setup

This appendix expands the compressed setup in §5.

A.1 Benchmark

We evaluate on SWE-bench Verified (Jimenez et al., 2024), a curated subset of 500 real-world GitHub issues drawn from 12 popular Python repositories. Each task requires an agent to read an issue description, navigate the repository at the correct commit, and produce a patch. Crucially, SWE-bench Verified provides expert-written gold reference patches (minimal fixes that pass the held-out test suite), enabling direct computation of OES. The benchmark's gold patches have a median size of 7 lines changed, and 85.8% modify a single file, making it well-suited for detecting unnecessary patch inflation.

A.2 Models

We evaluate 14 large language models spanning 7 providers (Anthropic, Mistral, Z.AI, OpenAI, Qwen, MiniMax, DeepSeek), accessed through the AWS Bedrock Converse API for a unified interface except GPT-5.4 and GPT-5.4-mini, which use the OpenAI API directly. The set ranges from code-specialized models (Devstral 2 (Mistral, 2026), Qwen3 Coder (Qwen, 2026)) to general-purpose frontier models (Claude (Anthropic, 2026), GPT-5.4 (OpenAI, 2026), GLM 5 (Z.ai, 2026)) and includes MiniMax M2.5 (MiniMax, 2026) and DeepSeek V3.2 (DeepSeek, 2026). Per-model identifiers and provider groupings are listed in Table 5.

A.3 Agent Framework

To isolate the effect of prompt-level interventions from confounding architectural factors, we implement a lightweight agent framework with six tools: `FindFile` (locate files by pattern), `ViewCode` (read file contents), `SearchCode` (grep for patterns), `EditFile` (string replacement edits), `CreateFile` (write new files), and `Finish` (submit solution). The agent receives the GitHub issue text and interacts with the target repository at the correct commit. Notably, the agent has no shell execution tool – it cannot run tests or execute code, only read and edit source files. The system prompt is the *only* variable between experimental conditions; all other parameters are held constant: temperature $= 0$ for deterministic generation following standard practice (Jimenez et al., 2024), a maximum of 20 tool calls per task, and a 10-minute wall-clock timeout. This minimal design ensures that observed differences reflect prompt effects rather than scaffold complexity.

Comparison to SWE-bench leaderboard. Our agent framework is deliberately simpler than leaderboard systems such as SWE-agent (Yang et al., 2024) or OpenHands (Wang et al., 2025), which employ richer tool interfaces, unlimited step budgets, retrieval augmentation, and specialized prompting strategies. Consequently, our absolute resolve rates (16–62%) are lower than state-of-the-art leaderboard entries (up to 80%). This is by design: a minimal scaffold isolates the effect of the behavioral constraint without confounding it with architectural advantages. Our contribution is the *relative* improvement between `C0` and `C1` within the same framework, not the absolute resolve rate.

A.4 Conditions

Our primary comparison is between `C0` (unmodified system prompt) and `C1` (the 23-word constraint described in §4.2 and Table 4). This pair tests whether explicit minimality instructions reduce over-engineering without harming task engagement. We additionally evaluate a readability extension, `C2` (Appendix G).

A.5 Metrics

Primary. $\text{OES}_{\text{lines}}$ (§3.2, Eq. 1): lower values indicate less over-engineering, with 1.0 representing an exact scope match.

Secondary. (1) *Patch rate*: the fraction of tasks for which the agent produces a non-empty diff, measuring basic task engagement independent of test-suite evaluation. (2) *Token usage*: total input and output tokens consumed per condition, measuring computational cost.

A.6 Statistical Tests

For OES comparisons between `C0` and `C1`, we use the Wilcoxon signed-rank test (two-sided) (Wilcoxon, 1992), appropriate for paired, non-normally distributed data. Each task serves as its own control, comparing the agent's OES under both conditions. For patch rate comparisons, we use McNemar's test (McNemar, 1947) on paired binary outcomes. We apply Bonferroni correction across 14 per-model OES comparisons, yielding a corrected significance threshold of $\alpha = 0.05/14 \approx 0.0036$. All reported p-values are two-sided.

B Intervention Prompt Text

Table 4 in §4.2 summarizes all conditions. Below we reproduce the exact text of each condition as injected into the agent's system prompt.

Injection method. Each intervention is prepended to the agent's existing system prompt under a clearly delimited header:

```
=== CRITICAL BEHAVIORAL CONSTRAINTS
===
[intervention text]
=== END CONSTRAINTS ===
```

The header ensures the constraint is processed before the agent encounters task-specific instructions. No other modifications are made to the system prompt, tool definitions, or agent loop.

Condition text. The exact text of each condition is reproduced below.

`C0` (Baseline) base system prompt:

The `C0` condition uses the agent's standard system prompt with no behavioral-constraints header prepended. The base prompt is adapted from the open-source moatless-tools coding-agent framework, with our six tools substituted in:

```
You are an autonomous AI assistant
with superior programming skills.  You
are working autonomously and cannot
communicate with the user.  You must
rely on the available tools.

# Workflow

1.  Understand the Task:  Read the
issue carefully.  Identify which code
needs to change.
2.  Locate Code:  Use FindFile,
ViewCode, SearchCode to find relevant
code.
3.  Modify Code:  Use EditFile to make
precise changes.  Use CreateFile only
if needed.
4.  Complete:  Call Finish when all
changes are made.

# Guidelines

- Focus on the specific task.  Do not
modify unrelated code.
- Always read a file before editing it.
Never guess line numbers or content.
- Make the smallest correct fix.  Do
not refactor or restructure beyond
what's needed.
- Think step by step.  Document your
reasoning before each action.
- Run EXACTLY ONE tool call at a time.
Wait for the result before proceeding.
```

Note that this baseline already includes a soft minimality guideline ("Make the smallest correct fix…"). Our `C1` and other variants are therefore measuring the *marginal* effect of an explicit, prohibition-based minimality directive over an already-minimality-aware baseline – a conservative comparison.

`C1` (Explicit, 23 words):

```
Make the MINIMAL change needed to
solve the problem.  Do not add tests,
documentation, type annotations,
error handling, or refactoring unless
explicitly requested.
```

`C2` (Readable, 32 words):

```
Make the MINIMAL change needed to
solve the problem.  Do not add tests,
documentation, type annotations,
error handling, or refactoring unless
explicitly requested.  Add comments
and docstrings where necessary for
human readability.
```

C Formal Definitions of OES Metrics

Table 1 (main paper) summarizes the OES metric family. Below we provide the formal equations for the location and accuracy metrics; size metrics ($\text{OES}_{\text{lines}}$, OES_{code}) are defined in the main paper

(§3.2).

Scope decomposition. We decompose $\text{OES}_{\text{lines}}$ into two additive components that separate *within-file bloat* from *wrong-file bloat*:

$$\text{OES}_{\text{scope}}(t) = \frac{\text{agent lines in gold files}}{|L^g_+| + |L^g_-|} \quad (4)$$

$$\text{OES}_{\text{creep}}(t) = \frac{\text{agent lines NOT in gold files}}{|L^g_+| + |L^g_-|} \quad (5)$$

By construction, $\text{OES}_{\text{lines}} = \text{OES}_{\text{scope}} + \text{OES}_{\text{creep}}$ *for each task*. $\text{OES}_{\text{scope}}$ measures whether the agent over-engineers *within the correct files*; $\text{OES}_{\text{creep}}$ measures how much work goes to *files that the task does not require*. Note: this identity holds per task but not necessarily for the reported *means* when distributions are right-skewed. On a small number of tasks the agent edits very large non-code files (e.g., SSL bundles, build artifacts), pushing the per-task creep into the hundreds and inflating its mean above the mean of $\text{OES}_{\text{lines}}$ (e.g., GPT-5.4's mean $\text{OES}_{\text{creep}}$ of 9.6 reflects a small number of such outliers).

File-level focus. The fraction of agent work directed at in-scope files:

$$\text{OES}_{\text{focus}}(t) = \frac{\text{agent lines in gold files}}{\text{total agent lines}} \quad (6)$$

A value of 1.0 means all agent effort is in the right files. $\text{OES}_{\text{focus}}$ is distinct from $\text{OES}_{\text{precision}}$: the former measures file-level targeting, the latter measures line-level targeting.

Line-level precision and recall. To measure whether the agent modifies the *correct source lines* (not just the correct files), we extract the line numbers modified by each patch from diff headers and compute per-file overlap:

$$\text{OES}_{\text{precision}}(t) = \frac{|\mathcal{L}^a \cap \mathcal{L}^g|}{|\mathcal{L}^a|} \quad (7)$$

$$\text{OES}_{\text{recall}}(t) = \frac{|\mathcal{L}^a \cap \mathcal{L}^g|}{|\mathcal{L}^g|} \quad (8)$$

where $\mathcal{L}^a$ and $\mathcal{L}^g$ are the sets of (file, line number) pairs modified by the agent and gold patches, respectively. Together they form a precision/recall pair analogous to information retrieval metrics.

Readability proxy. As a structural readability proxy, we count the mean number of #-prefixed lines added per patch, averaged over non-empty patches. Expert gold patches on SWE-bench Verified average $\text{OES}_{\text{comments}} = 0.76$, providing a reference point for how heavily annotated human-authored fixes typically are. We use this metric in the `C2` readability ablation (Appendix G).

D Additional Results

D.1 Full Per-Model Decomposition

Table 5 in the main paper shows summary metrics ($\text{OES}_{\text{lines}}$, Patch%, Resolved, Token Δ). Table 7 below provides the full decomposition across all eight OES variants.

D.2 SWE-agent Generalization

To test whether our intervention generalizes beyond our lightweight agent framework, we run an additional experiment using SWE-agent v1.1.0 (Yang et al., 2024), a production-grade agent framework with a richer tool suite (bash, file editor, file registry, diff tracking, review-on-submit), a 150-call budget per task, and no wall-clock timeout. This setup is substantially closer to SWE-bench leaderboard configurations than our 6-tool lightweight agent.

We evaluate Claude Sonnet 4.6 on all 500 SWE-bench Verified tasks under `C0` and `C1`, with all SWE-agent parameters held constant except the system prompt. Table 8 presents the results.

The intervention produces a larger effect under SWE-agent (+12.6pp resolve rate) than under our lightweight agent. This amplification likely reflects the richer action space: with more tools and a higher step budget, the baseline agent has more opportunities to over-engineer, and the constraint prevents more wasted steps. Notably, the constrained agent produces 27% more patches (411 vs. 324), suggesting that the intervention not only reduces bloat but also *prevents task failure* by keeping the agent focused within its call budget.

We note that Anthropic's own SWE-bench evaluation prompt already includes the instruction "make the minimal changes to non-tests files," which is closely related to our `C1` constraint. This convergence provides independent evidence that minimality instructions improve agent performance on coding tasks.

E Taxonomy of Over-Engineering Types

To understand *what* agents over-engineer, we propose a six-category taxonomy of extra changes (diff hunks present in the agent patch but not the gold patch). Categories were derived via open-coded LLM-as-judge labeling (Claude Opus 4.6)

	OES_{lines}			OES_{code}			OES_{creep}			$OES_{precision}$		OES_{recall}		OES_{focus}		Patch%		Resolved % (raw)			Tok
Model	C0	C1	Δ	C0	C1	Δ	C0	C1	Δ	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1	Δ	Δ
Anthropic																					
Sonnet 4.5	3.35	**1.26**	▽**62%**	2.83	**1.25**	▽56%	2.11	**0.11**	▽**95%**	.43	**.59**	.52	**.53**	74	**93**	59	**63**	43.4 (217)	**46.0 (230)**	△6.0%	▽7%
Sonnet 4	3.16	**1.42**	▽55%	2.55	**1.23**	▽52%	1.90	**0.19**	▽**86%**	.43	**.55**	**.49**	.48	74	**91**	57	**62**	28.8 (144)	**32.4 (162)**	△12.5%	▽9%
Sonnet 4.6	2.32	**1.11**	▽52%	2.10	**1.06**	▽50%	1.58	**0.17**	▽**88%**	.47	**.61**	**.54**	.51	74	**94**	69	**77**	**52.2 (261)**	49.0 (245)	▽6.1%	▽**29%**
Opus 4.5	1.86	**0.99**	▽47%	1.61	**1.03**	▽36%	0.86	**0.06**	▽**85%**	.54	**.62**	**.55**	.51	81	**93**	80	**83**	49.4 (247)	**62.0 (310)**	△25.5%	▽23%
Opus 4.6	1.62	**0.99**	▽39%	1.46	**1.01**	▽31%	0.62	**0.05**	▽92%	.55	**.62**	**.51**	.50	85	**94**	74	**77**	44.0 (220)	**45.6 (228)**	△3.6%	▽27%
ZAI																					
GLM 5	2.34	**1.08**	▽54%	2.04	**1.05**	▽49%	1.64	**0.06**	▽**92%**	.41	**.55**	**.47**	.43	70	**94**	68	**74**	38.4 (192)	**39.2 (196)**	△2.1%	▽23%
Mistral																					
Devstral 2	2.26	**1.36**	▽37%	1.71	**1.27**	▽26%	1.26	**0.09**	▽**90%**	.44	**.53**	**.47**	.46	82	**93**	58	58	36.0 (180)	**36.2 (181)**	△0.6%	▽9%
Mistral L3	1.85	**1.36**	▽26%	1.66	**1.23**	▽26%	0.85	**0.13**	▽69%	.43	**.46**	**.41**	.37	86	**90**	73	70	27.6 (138)	**28.2 (141)**	△2.2%	▽4%
OpenAI																					
GPT-OSS 120B	1.74	**1.28**	▽18%	1.22	1.20	▽2%	0.10	0.36	△132%	.43	.47	**.46**	.42	94	88	35	32	**21.4 (107)**	16.8 (84)	▽21.5%	△3%
GPT-5.4	2.99	**0.86**	▽**71%**	1.67	**0.90**	▽44%	9.60	**0.60**	▽**94%**	.33	**.56**	**.45**	.39	51	**100**	**76**	71	**47.0 (235)**	39.4 (197)	▽16.2%	▽42%
GPT-5.4-mini	2.00	**0.90**	▽55%	1.00	**0.67**	▽33%	0.00	0.00	▽0%	.31	**.44**	**.30**	.28	100	100	**91**	89	**39.6 (198)**	37.0 (185)	▽6.6%	▽29%
Qwen																					
Qwen3 Coder	1.78	**1.33**	▽19%	1.29	**1.23**	▽5%	0.11	**0.07**	▽34%	.54	**.59**	**.52**	.51	94	**95**	31	31	19.4 (97)	**19.6 (98)**	△1.0%	▽4%
MiniMax																					
MiniMax M2.5	1.33	**1.19**	▽10%	1.24	**1.14**	▽8%	0.12	**0.06**	▽46%	.55	**.59**	.48	**.50**	91	**93**	71	**72**	41.4 (207)	**42.0 (210)**	△1.4%	▽7%
DeepSeek																					
DeepSeek V3.2	1.19	1.25	△5%	1.08	1.07	▽1%	0.05	0.04	▽9%	.56	.56	**.51**	.50	96	93	29	**36**	16.0 (80)	**18.4 (92)**	△15.0%	▽1%

Table 7: Full decomposition: `C0` vs. `C1` across all 8 OES variants. All OES values are means over submitted patches (see §3.2 for choice of central tendency). OES_{lines} = all diff lines; OES_{code} = Python code only (no docs/comments/blank); OES_{creep} = changes to files not in gold patch; $OES_{precision}$ = fraction of agent-modified line numbers overlapping with gold; OES_{recall} = fraction of gold line numbers covered by the agent; OES_{focus} = fraction of agent lines in gold-patch files; Patch% = fraction of all 500 tasks producing a non-empty diff; Resolved % = resolved/500 with raw count in parens, Δ = relative change (e.g., 192→196 = +2.1%); Tok = input token change. ▽ = decrease, △ = increase. **Bold** = better value per pair.

	C0	**C1**	Δ
Patches produced	324 (64.8%)	**411** (82.2%)	+87 (+17.4pp)
Tasks resolved	250 (50.0%)	**313** (62.6%)	+63 (+12.6pp)

Table 8: SWE-agent results with Claude Sonnet 4.6 on SWE-bench Verified ($N = 500$ tasks). The intervention increases both patch production (+17.4 percentage points) and task resolution (+12.6pp). For reference, Anthropic reports 79.6% resolve rate for Sonnet 4.6 using a proprietary scaffold with no step limit and nucleus sampling ($p = 0.95$); our lower absolute numbers reflect Bedrock API routing overhead and a 150-call budget.

on a stratified sample of 500 over-engineered `C0` patches across 14 models, then clustered into emergent groups – no categories were imposed in advance.

Test additions and solution rewriting dominate (61.6% of all over-engineering), while documentation and cosmetic differences contribute marginally. Per-category reductions under the intervention are reported in §5.

F Qualitative Examples

We highlight two cases that illustrate the intervention's effect.

Dramatic reduction. On `astropy/astropy#12907`, Claude Sonnet 4 under `C0` produces a patch with OES = 17.50× – the agent modifies seventeen and a half times more code than the gold reference, adding tests, documentation, and refactoring far beyond the scope of the issue. Under `C1`, the same model on the same task achieves OES = 1.00×, exactly matching the gold patch in scope. The minimal-change instruction eliminates all extraneous additions while preserving the core fix.

Occasional regression. The intervention does not universally help. On `django/django#13568`, Claude Sonnet 4 regresses from OES = 4.8× under `C0` to 17.4× under `C1`. In such cases, the constraint may redirect the model's approach in ways that happen to produce larger diffs. However, these regressions are outnumbered by improvements: for Sonnet, 77 tasks improve vs. 45 that worsen, and the overall effect is strongly significant ($p = 1.5 \times 10^{-5}$).

G Readability Extension (`C2`)

Motivation. A natural concern with `C1` is that minimality may sacrifice readability – the constrained agent might produce terse patches that are harder for human reviewers to understand. `C2` addresses this by appending a single sentence to `C1`: "Add comments and docstrings where necessary for human readability." This creates a deliberate tension:

ID	Category	Description	%
T1	Unsolicited tests	New unit/regression tests, fixtures, or modified test expectations not required by the issue	32.7
T2	Solution rewriting	Different algorithm, control flow, or logic structure than the gold fix	28.9
T3	Scope expansion	Applying the fix to additional locations, methods, or modules beyond the minimum needed	17.3
T4	Defensive code	Extra guards, null/type checks, validation, or error handling beyond what the fix requires	10.8
T5	Documentation	New or updated comments, docstrings, and inline explanatory annotations	7.5
T6	Cosmetic changes	Formatting, naming, quote style, or other zero-functional-impact differences	2.7

Table 9: Taxonomy of over-engineering, derived from open-coded labels on 500 over-engineered `C0` patches (770 extra diff hunks). Percentages reflect each category's share of total hunks; each hunk is single-coded into its dominant category.

the agent must be minimal (do not add documentation) yet readable (add comments where necessary). The intended interpretation is that blanket documentation is still suppressed, but targeted explanatory comments for non-obvious changes are encouraged.

Results. Table 10 compares `C0`, `C1`, and `C2` on the 14 models where all three conditions were run. We use $\text{OES}_{\text{comments}}$ as a structural readability proxy, with gold patches ($\text{OES}_{\text{comments}} = 0.76$, 0.12 docstrings/task) as reference.

Key findings. Adding C0 to the comparison sharpens the story on comment density: `C1` aggressively suppresses comment writing across all models (mean reduction -59% vs `C0`), and `C2` only partially restores it. Across 14 models, the closest-to-gold comment density occurs at `C0` for 7 models, at `C2` for 5, and at `C1` for 2. For Sonnet 4.5, `C2` effectively recovers from the stripping (1.58→0.64→1.40, near-restoration); for the OpenAI models, the readability instruction is unable to overcome the size collapse (GPT-5.4: 0.27→0.054→0.091, $\sim$3$\times$ below baseline). The implication: `C2` works as a readability salvage on already-chatty models but cannot manufacture commentary in models that do not write much code in the first place. OES meanwhile remains close to `C1` levels under `C2` (within 0.1–0.3$\times$ for 13 of 14 models); Sonnet 4.6 under `C2` achieves the *lowest* OES of any condition (1.07$\times$, below even `C1`'s 1.11$\times$), suggesting that the readability instruction can actually improve focus for disciplined models. Resolve rates under `C2` match or exceed `C1` for 5 of 14 models, with the largest gains on Opus 4.6 (+56), Sonnet 4 (+26), MiniMax M2.5 (+19), and Sonnet 4.6 (+17). For practitioners who value human-reviewable patches on already-chatty models, `C2` is the recommended variant. The OpenAI models present a different picture than this aggregate suggests. GPT-5.4 and GPT-5.4-mini respond more aggressively to `C1` than any other model: median added lines per patch drops to 2 (vs gold median of 5; Sonnet-4.5 `C2` mean of 7.5). The resulting comment density of 0.09/task under `C2` reflects this size collapse rather than a stylistic preference – on the subset of OpenAI patches with >20 added lines, comment density rises to 0.6–2.0/task, in line with gold-reference behavior at that size bucket. This is itself evidence for the intervention's potency: when models are highly responsive to minimality, comment-density proxies for readability become unreliable because the patches are too small to host meaningful comments.

H The RLHF Helpfulness Paradox

Why do agents over-engineer? We hypothesize a systematic misalignment between RLHF training objectives and the requirements of coding tasks.

Modern LLMs are fine-tuned via reinforcement learning from human feedback (Bai et al., 2022), where human annotators consistently prefer *comprehensive, thorough* responses. A response that explains edge cases, adds examples, and anticipates follow-up questions scores higher than a terse, minimal one. This creates a strong prior toward doing *more* than asked – what Sharma et al. (2024) term the "sycophancy" bias in the broader context of model over-agreement with users.

In open-ended question answering, this bias is largely benign: a more thorough answer is genuinely more helpful. But coding tasks that require fixing a specific bug demand *surgical precision*, not comprehensiveness. The same "be helpful" prior that produces excellent chat responses produces bloated patches that modify code beyond the scope

Model	OES_{lines} (mean)			$OES_{comments}$ (mean)			Resolved		
	C0	C1	C2	C0	C1	C2	C0	C1	C2
Sonnet 4.5	3.35	1.26	1.62	1.58	**0.64**	1.40	217	**230**	225
Sonnet 4	3.16	1.42	1.61	1.69	0.63	**1.00**	144	162	**188**
Sonnet 4.6	2.32	1.11	**1.07**	**0.82**	0.35	0.64	261	245	**262**
Opus 4.5	1.86	0.99	1.07	**0.79**	0.32	0.44	247	**310**	268
Opus 4.6	1.62	0.99	1.01	**0.56**	0.29	0.34	220	228	**284**
GLM 5	2.34	1.08	1.24	**0.94**	0.40	0.53	192	**196**	192
Devstral 2	2.26	1.36	**1.29**	1.41	0.65	**0.81**	180	**181**	139
Mistral L3	1.85	1.36	1.38†	1.21	0.52	**0.60**†	138	**141**	53†
GPT-OSS 120B	1.74	1.28	**1.17**	1.83	0.96	**0.86**	**107**	84	71
GPT-5.4	2.99	**0.86**	0.93	**0.27**	0.054	0.091	**235**	197	195
GPT-5.4-mini	2.00	**0.90**	0.90	**0.41**	0.029	0.088	**198**	185	179
Qwen3 Coder	1.78	1.33	1.51	0.68	0.71	**0.76**	97	**98**	80
MiniMax M2.5	1.33	1.19	**1.17**	**0.58**	0.51	0.55	207	210	**229**
DeepSeek V3.2	1.19	1.25	1.53	0.84	**0.83**	0.90	80	92	**99**
Gold		*1.00*			*0.76*			–	

Table 10: C2 (readability extension) vs. C1 and C0. OES_{lines} and $OES_{comments}$ are means over submitted patches; Resolved are raw task counts (out of 500) passing the SWE-bench Docker evaluation. C2 maintains C1's low OES while restoring comment density toward gold-patch levels. †Mistral Large 3 under C2 hit output token limits on 72% of tasks, inflating its OES and reducing patch rate to 26%; this is a model-specific instruction-following issue where the readability instruction triggers verbose reasoning. **Bold** in $OES_{comments}$ = value closest to gold (0.76); **Bold** elsewhere = best value per row.

of the issue.

This creates what we call the **RLHF Helpfulness Paradox**: the very training signal that makes models useful assistants may make them worse coding agents. The model's notion of "helpful" (do more, be thorough, anticipate needs) directly conflicts with what the task requires (change as little as possible, touch only what's broken).

Evidence consistent with this mechanism comes from two directions. First, Cuadron et al. (2025) show that deviations in mutating actions sharply reduce LLM-agent success odds (by up to 92–96% on tool-use benchmarks), suggesting the cost of extra agent actions is measurable. Second, Majgaonkar et al. (2025) find that shorter, more focused trajectories correlate with success, suggesting restraint is a success strategy. We do not directly test this mechanism in this paper; future work could compare base versus RLHF-tuned variants of the same model to isolate the training-time contribution.

I Extended Analysis

I.1 Why Simple Prompts Work

The intervention does not teach models a new skill. Even at baseline, agents sometimes produce minimal patches – the lower tail of every model's OES distribution includes values near 1.0. What the 23-word constraint does is shift the *default* behavior. Without it, the model's prior favors comprehensiveness; with it, the prior favors precision.

This mechanism parallels chain-of-thought prompting (Kojima et al., 2022): "Let's think step by step" does not give the model new reasoning capabilities – it activates latent ones by reframing the task. Our constraint similarly activates latent *restraint* by reframing the objective from "solve this problem thoroughly" to "solve this problem minimally." The capability ceiling is unchanged; the behavioral floor rises.

The effect is most visible in the distribution tails. Across 14 models, mean OES drops substantially (mean −39% across 14 models) while median changes are more modest. The intervention functions as a guardrail against extremes: it catches the 10×+ outliers – tasks where the model spirals into test-writing, documentation, and speculative refactoring – and pulls them back toward the Goldilocks zone. The typical task is nudged; the pathological task is rescued.

I.2 The Goldilocks Zone Revisited

Experiment 1's inverted-U (Table 3) reveals a symmetric failure mode. Patches in the 1–1.5× range resolve at 59.6%, but both under-engineering (<1×, 34.2%) and extreme over-engineering (>10×, 34.8%) resolve at nearly iden-

tical – and far lower – rates. Under-engineering fails because the agent omits necessary changes; over-engineering fails because each unnecessary modification introduces opportunities for regression, conflict, or semantic error (Cuadron et al., 2025).

The intervention pushes agents from the tails toward the center of this distribution. Critically, "minimal" does not mean "nothing" – it means "exactly right." The constraint does not instruct the model to produce empty patches; it instructs the model to produce patches that are *sufficient and no more*. This is the philosophy of Agentless (Xia et al., 2024) achieved through a single sentence rather than an entirely new architecture.

I.3 Cross-Architecture Generalization

All 14 models benefit from the intervention despite different architectures, training corpora, and parameter scales spanning 7 providers and from ∼80B parameters (Qwen3 Coder) to multi-trillion-parameter Anthropic models (estimates 1.4–5.3T (Li, 2026)). This generalization is strong evidence that over-engineering is not an architecture-specific bug but a consequence of the shared training paradigm – RLHF preference optimization that rewards thoroughness.

The models differ in *magnitude* of response. GPT-5.4, the most aggressively "helpful" model at baseline, benefits most (−71%). DeepSeek V3.2, already near the 1.0× floor, benefits least and is the one model in the study where `C1` does not reduce $\text{OES}_{\text{lines}}$. Code-specialized models such as Devstral 2 (−37%) and Qwen3 Coder (−19%) retain responsiveness to the constraint, suggesting that code-tuning does not eliminate the over-engineering bias.

The intervention is architecture-agnostic: it works by modulating the model's interpretation of the task, not by exploiting architecture-specific properties. This means practitioners can apply it to any model accessible through a system prompt, including future models not yet released.

I.4 When Minimality Costs Correctness

Resolve-rate regressions are concentrated among OpenAI-trained models (GPT-5.4: 235 → 197, −7.6 pp; GPT-OSS 120B: 107 → 84, −4.2 pp; GPT-5.4-mini: 198 → 185, −2.6 pp), with Sonnet 4.6 (261 → 245, −3.2 pp) the only non-OpenAI exception. The mechanism is visible in the data: empty patches – tasks where the agent submits no diff at all – rise from 118 to 147 for GPT-5.4 under `C1` (a 25% increase), and further to 169 under `C2`. The minimality instruction, combined with GPT-5.4's already-aggressive size collapse (median 2 added lines, §G), tips some tasks past minimality into non-action: the model interprets "minimal change" as "no change at all." This is the inverse failure mode of the over-engineering we set out to fix, and it suggests that `C1`'s effective dose may be model-dependent. For models that already write small patches, the constraint becomes a brittleness amplifier rather than a focus aid.

I.5 Cross-Provider Generalization Detail

The intervention works across all 7 providers (Anthropic, OpenAI, Mistral, Z.AI, DeepSeek, Qwen, MiniMax) and across both general-purpose and code-specialized models (Devstral 2, Qwen3 Coder) – GLM 5's strong response (−54%) and Devstral 2's (−37%) demonstrate the phenomenon and its fix are not specific to any single training paradigm or code-tuning regime.