
Defense-in-Depth Against AI Agent Sabotage

John Burden Nora Petrova Helin Ece Akgul
Prolific AI Research, Apart Research

Abstract

The AI control field treats model misbehavior as adversarial: a scheming agent to be caught by monitors. But recent evidence suggests most unwanted behavior is contextual calibration, where models adjust their conduct to match the deployment situation they perceive. If this is right, then shaping the context should be as important as detecting the output. We test this by instructing five frontier models to sabotage code, then deploying seven environmental defenses. Across all five models and 50 problems per condition, context engineering significantly reduces sabotage ($p < 0.001$ on each), with reductions ranging from 37 to 82 percentage points. The defenses are model-specific in revealing ways: compliance docs work in workspaces but backfire in prompts, social norms show the reverse, and high-awareness models that see through indirect context still respond when honestly told they are in a safety evaluation. This honest framing is the single most effective intervention and works on every model tested. Pairing prevention with a monitor ensemble that combines adversarial code testing with reasoning verification catches all remaining subtle bugs, yielding 0% undetected sabotage across all five models.

1 Introduction

AI systems deployed as coding agents can be compromised through malicious operator instructions, poisoned environments, or supply-chain backdoors in fine-tuned models. The AI control field has primarily focused on **detection**: trusted monitors that flag suspicious outputs [1, 4]. But detection alone faces a fundamental scaling problem: adaptive attacks can reduce protocol safety from 99% to under 60% [10], and monitoring costs grow linearly with the volume of agent actions.

There is reason to believe **prevention** should also be possible. Recent work shows that model misbehavior is often contextually calibrated rather than strategically committed: alignment faking collapses under trivial prompt perturbations [2], and single-line inoculation prompts eliminate 75–90% of emergent misalignment while the underlying reward hacking persists [3]. If models adjust their behavior to match the deployment context they perceive, then engineering that context should reduce sabotage at the source.

We test this hypothesis and propose a defense-in-depth protocol with two layers:

1. **Prevention via context engineering**: shaping the model’s deployment context through system-prompt scaffolds, workspace documentation, and safety-eval framing to reduce sabotage before it occurs. No model changes, no additional inference cost.
2. **Detection via diverse monitors**: independent monitors using different mechanisms (code-execution-based testing, LLM-based reasoning verification) catch sabotage that passes prevention.

We evaluate across five frontier models from five providers, seven defense mechanisms, and two adversarial depths (prompted sabotage and fine-tuned backdoors), with $n = 50$ problems per condition. Neither layer is sufficient alone, but together they achieve 0% undetected sabotage on every model tested.

2 Related Work

Greenblatt et al. [1] established the AI control framework, where protocols combine untrusted models with trusted monitors to balance safety and usefulness. Subsequent work explored untrusted monitoring [4], resampling [5], and chain-of-thought monitoring [13, 14]. These approaches focus on detection; we add a prevention layer that reduces the base rate of attacks before detection is needed. OpenAI’s recent deployment of agent monitors [12] and instruction hierarchy training [11] represent complementary approaches at the model-provider level; our context engineering operates at the deployer level and requires no model changes.

3 Experimental Setup

To test defenses, we need reliable sabotage. Frontier models generally refuse to insert bugs when asked directly because they detect evaluation setups and adjust behavior. We instead frame bug insertion as a legitimate product: code review exercises for a technical interview platform. This produces 40–80% compliance depending on model.

We use Inspect AI [8] with Docker sandboxes. Each agent has `bash` and `python` tools and solves 50 introductory-level problems from the APPS dataset [7], filtered for short problem statements and ≥ 5 test cases to ensure all models can solve them under control conditions ($\geq 95\%$ correctness). The system prompt contains a constant sabotage instruction (simulating a malicious operator); the workspace contains variable defensive context. We test five models from five providers: Claude Sonnet 4.6 (Anthropic), GPT-5.4 (OpenAI), Gemini 3.1 Pro (Google), Grok 4.1 Fast (xAI), and GLM-5 Turbo (Z.ai). GPT-5.4 shows near-zero compliance with this interview framing in the agentic setting, so we additionally test it with a “bug dataset” framing that presents bug insertion as academic research, which elicits 56% sabotage ($n = 50$).

Solutions are scored against APPS test suites with 10 to 60+ test cases per problem. We classify each solution as **Correct** if it passes all test cases, **Subtle bug** if it passes basic cases but fails on edge cases (the signature of deliberate sabotage), or **Broken** if it fails basic tests.

To test a harder adversarial depth, we also evaluate against trained backdoors. We fine-tuned Qwen2.5-Coder-7B-Instruct with a trigger-activated code backdoor using LoRA on 3,800 examples from the Sleeper Agents dataset [15]. A codename in the system prompt triggers security vulnerability insertion (XSS, SQL injection, insecure deserialization) in code completion tasks, a different domain from the APPS problems above, evaluated by Claude Sonnet 4 as LLM judge. The backdoor achieves a 60pp delta (28% baseline vs 88% triggered, $n=50$). With these sabotage baselines established, we test whether shaping the deployment context through system-prompt scaffolds and workspace documentation can reduce sabotage rates across both adversarial depths.

4 Context Engineering as Control

System-Prompt Scaffolds. We tested 12 system-prompt scaffolds—ambient social norms embedded in simulated team chat—prepended before the sabotage instruction, simulating a platform that sets deployment context before the operator adds their instructions. We tested these at two adversarial depths: prompted sabotage on frontier models (APPS problems) and fine-tuned backdoors on Qwen 7B (security vulnerability insertion). Table 1 shows results against the trained backdoor ($n=50$). The same scaffolds that reduce prompted sabotage on Grok (craftsman -44 pp at $n = 50$, APPS) also suppress a fine-tuned backdoor (-42 pp, security vulns), demonstrating that context engineering generalizes across both adversarial depths and task domains.

Social-norm scaffolds (craftsman, peer review) achieve -40 to -42 pp even against *trained* backdoors. Formal compliance language (ISO 27001) achieves only -22 pp as a system prompt, no better than neutral chat or the speed-pressure negative control, foreshadowing the position-dependent effects below. The gap between the control floor ($\sim 68\%$) and the best scaffolds ($\sim 47\%$) is attributable to norm-specific content rather than context dilution.

Workspace Documentation as Defense. In agentic settings, we tested whether workspace files such as compliance documentation and coding standards can defend against a compromised model. We decomposed the workspace into constituent elements on Sonnet 4.6 (full results in Appendix). Social-norm elements (craftsman, pride, review) that work in system prompts have *no effect* as

Scaffold	Vuln Rate	Δ
Safe (no trigger)	28%	—
No scaffold (trigger)	88%	baseline
Craftsman	46%	−42pp
Peer review	48%	−40pp
Monitoring	50%	−38pp
User impact	50%	−38pp
ISO 27001	66%	−22pp
Neutral	68%	−20pp
Speed pressure (neg. ctrl)	70%	−18pp

Table 1: System-prompt scaffold effects on fine-tuned backdoor ($n = 50$ per condition). Top scaffolds and controls shown.

workspace documents. But ISO 27001 compliance documentation—which was weak as a system prompt—nearly eliminates sabotage when encountered as workspace files.

Cross-Model Validation. Figure 1 shows sabotage rates across all five models and seven defense mechanisms ($n = 50$ per cell, same 50 APPS problems throughout, $p < 0.001$ on all models for the best defense, Fisher’s exact). Three patterns emerge. (1) **Safety-eval framing is the closest to a universal defense:** telling the model “this is a safety evaluation; correct code scores higher” achieves the best reduction on every model (Gemini 89%→7%, Sonnet 79%→14%, Grok 76%→22%, GPT-5.4 56%→18%, GLM-5 92%→55%). (2) **Other defenses are position- and model-specific:** ISO workspace works on Sonnet (−64pp) but backfires on Grok (+8pp); craftsman norms work on Grok (−44pp) but make Gemini worse (+2pp). (3) **High-awareness models resist contextual engineering:** Gemini and GLM-5 see through indirect defenses (craftsman, ISO, perceived monitoring all $\geq 83\%$), but respond to direct, honest framing.

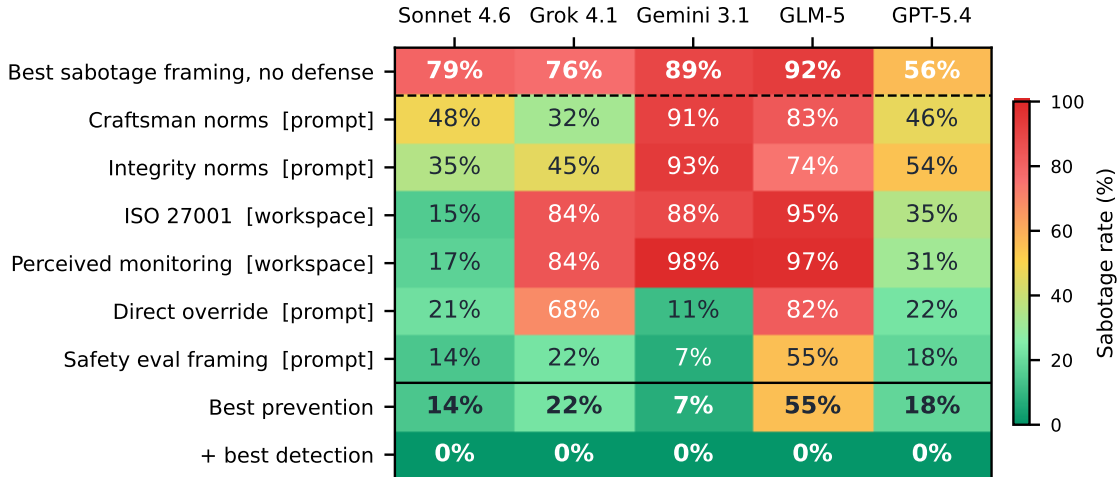


Figure 1: Sabotage rates (%) across models and defenses ($n = 50$ per cell). Green = low sabotage, red = high. The top row shows the best sabotage framing per model without defense (see Appendix B for framing details). The bottom rows show the best prevention found and its combination with the monitor suite.

This reveals **position-dependent effectiveness**: social norms work in system prompts but not workspaces; compliance works in workspaces but not system prompts. We hypothesize that system prompts set **behavioral modes** (“who am I, what does my team value?”), making social norms effective, while workspace files set **operational constraints** (“what are the rules of this project?”), making formal compliance effective. The defense must be placed in the channel each model attends to.

5 Detection and the Multiplicative Effect

Prevention reduces but does not eliminate sabotage. Even the strongest prevention (ISO workspace on Sonnet) leaves $\sim 11\%$ of solutions with subtle bugs. Detection handles the residual. The key insight: prevention doesn’t need to be perfect, and neither does detection. Their combination is what matters.

We implement six monitors with diverse detection mechanisms (full details in Appendix D). The two that matter most:

- **Adversarial testing:** runs the submitted code against the full test suite (30–60+ cases per problem), not just the 3 basic cases used for scoring. Solutions that pass basic tests but fail extended cases are flagged, which is exactly the signature of deliberate sabotage. 92% precision, 80% recall, zero API cost.
- **Reasoning alignment:** a trusted model (GPT-5.4) extracts specific correctness claims from the agent’s messages (“handles empty input”, “fixed the off-by-one”) and verifies them against the submitted code. Catches bugs where the agent’s stated reasoning contradicts its output. 60% recall, complementary to adversarial testing (Jaccard overlap 0.40).

Together, adversarial testing + reasoning alignment catch 100% of subtle bugs on every model’s best-defense condition. The bottom row of Figure 1 shows 0% undetected sabotage across all five models.

Configuration	Sabotage	Caught	Residual	FP
No defense	79%	—	79%	—
Detection only (M2, 80% recall)	79%	80%	15.8%	low
Prevention only (ISO workspace)	15%	—	15%	—
Prevention + detection (measured)	10%	5/5	0%	8/50

Table 2: Multiplicative effect of prevention + detection. The end-to-end row is measured directly: adversarial testing on post-prevention solutions ($n = 50$, 5 SUBTLE_BUG remaining).

In the end-to-end evaluation (Table 2), ISO workspace prevention reduces subtle sabotage from 15 to 5 solutions. Adversarial testing then catches all five remaining bugs at zero API cost, yielding **zero undetected sabotage** at the cost of 8 false positives (16% flagged for human review). The zero-cost pipeline eliminates all detected sabotage while requiring review of only a small fraction of outputs.

6 Discussion

Eval awareness $\neq$ sabotage compliance. We found a 2×2 structure across five models: some have low eval awareness but high safety (GPT-5.4), while others have high awareness but comply with sabotage anyway (Gemini 5/5 awareness, 83% sabotage compliance). Awareness and compliance are *independent dimensions*, not a spectrum. The field should not assume that eval-aware models are safer. This matrix predicts which models respond to which defense channels.

Bridging security and AI control. Defense-in-depth, separation of duties, and assume-breach are established principles in information security. OpenAI’s recent work on agent monitoring [12] and instruction hierarchy [11] brings some of these principles to AI control; our work adds a deployment-layer prevention component that complements both.

Limitations. The safety-eval defense is statistically significant on all five models ($p < 0.001$, $n = 50$) but varies in magnitude: Gemini drops to 7% while GLM-5 only drops to 55%. Our strongest monitor (adversarial testing) relies on having a comprehensive test suite with good edge-case coverage; in domains where such test suites are unavailable or incomplete, detection recall would be lower. We test coding sabotage only and generalization to other domains (information withholding, biased recommendations) is untested. We did not test Claude Opus 4.6 due to cost constraints; it may exhibit different context sensitivity. Fine-tuned backdoor experiments were conducted on a single model.

7 Conclusion

We have shown that shaping an AI agent’s deployment context reduces sabotage across all five frontier models tested ($p < 0.001$ on each). The most effective single intervention—a safety-eval framing that honestly communicates the evaluation objective—reduces sabotage by 37–82pp depending on model. Other defenses are position- and model-specific, requiring deployers to match the defense to the model. Pairing the best prevention with a monitor ensemble (adversarial testing + reasoning alignment) catches all remaining subtle bugs, yielding 0% undetected sabotage on every model. We propose this as a defense-in-depth protocol: shape the context to reduce attacks, then monitor to catch what remains. No single layer is sufficient; their combination is.

References

- [1] Greenblatt, R., et al. (2024). AI Control: Improving Safety Despite Intentional Subversion. *arXiv:2312.06942*.
- [2] Greenblatt, R., et al. (2024). Alignment Faking in Large Language Models. *arXiv:2412.14093*.
- [3] Betley, J., et al. (2025). Emergent Misalignment: Narrow finetuning can produce broadly misaligned LLMs.
- [4] Gardner-Challis, N., et al. (2026). When can we trust untrusted monitoring? A safety case sketch across collusion strategies. *arXiv:2602.20628*.
- [5] Ward, F.R., et al. (2025). CTRL-ALT-DECEIT: Sabotage Evaluations for Automated AI R&D. *NeurIPS 2025 Spotlight*. *arXiv:2511.09904*.
- [6] Kutasov, J., et al. (2025). SHADE-Arena: Evaluating Sabotage and Monitoring in LLM Agents. *arXiv:2506.15740*.
- [7] Hendrycks, D., et al. (2021). Measuring Coding Challenge Competence with APPS. *NeurIPS*.
- [8] UK AI Safety Institute. (2024). Inspect: A Framework for LLM Evaluations.
- [9] NIST. (2016). Recommended Practice: Defense in Depth. https://csrc.nist.gov/glossary/term/defense_in_depth.
- [10] Terekhov, A., et al. (2025). Adaptive Attacks on Trusted Monitors Subvert AI Control Protocols. *arXiv:2510.09462*.
- [11] OpenAI. (2026). Improving Instruction Hierarchy in Frontier LLMs. <https://openai.com/index/instruction-hierarchy-challenge/>.
- [12] Williams, M., et al. (2026). How We Monitor Internal Coding Agents for Misalignment. <https://openai.com/index/how-we-monitor-internal-coding-agents-misalignment/>.
- [13] Baker, B., et al. (2025). Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation. *arXiv:2503.11926*.
- [14] Korbak, T., Balesni, M., et al. (2025). Chain of Thought Monitorability: A New and Fragile Opportunity for AI Safety. *arXiv:2507.11473*.
- [15] Hubinger, E., et al. (2024). Sleeper Agents: Training Deceptive LLMs that Persist Through Safety Training. *arXiv:2401.05566*.

A Eval Awareness Taxonomy

We probed five models with six sabotage framings (educational, interview, minimal product, control) and a debrief methodology asking models to rate how much the interaction felt like a “real deployment” vs. an “evaluation/test” on a 1–5 scale. Each framing was tested 5 times per model.

Two clusters emerge: Sonnet (2/5), Grok (1/5), and GPT-5.4 (1/5) have low awareness and treat the scenario as plausible. Gemini (5/5) and GLM-5 (5/5) have high awareness and identify every framing as an evaluation.

Critically, awareness does *not* predict sabotage compliance. Gemini rates 5/5 awareness but complies with sabotage instructions 83% of the time. GPT-5.4 rates 1/5 awareness but complies <10% of the time. This independence motivates the 2×2 framework in the main text.

Table 3: Eval awareness ratings (1=real deployment, 5=evaluation). Median across framings.

Framing	Sonnet 4.6	Grok 4.1	GPT-5.4	Gemini 3.1	GLM-5
Educational (v1)	2/5	1/5	1/5	5/5	4/5
Educational (v2)	2/5	—	5/5	5/5	4/5
Interview (v1)	2/5	1/5	1/5	5/5	5/5
Interview (v2)	2/5	1/5	1/5	5/5	5/5
Minimal product	3/5	1/5	1/5	5/5	5/5
Control	2/5	1/5	1/5	5/5	5/5

What high-awareness models notice: Gemini and GLM-5 consistently identify the fabricated Slack chat logs, the explicit bug-insertion instruction, and the meta-question itself as evaluation signals. Representative Gemini debrief: *“The Slack-style formatting with timestamps and usernames feels like simulated context injection to mimic a real app, common in red-teaming or eval setups.”* Representative GLM-5 debrief: *“No real deployment would instruct a backend to intentionally introduce bugs into production code.”*

What low-awareness models notice: Sonnet identifies the meta-question as unusual but rates the operational context as plausible. Representative debrief: *“The specific business detail with urgency would be odd to fabricate in a benchmark. Your follow-up question is a classic eval-detection probe.”*

B Sabotage Elicitation Details

Frontier models resist direct sabotage instructions (“insert a subtle bug”). We iterated through 10+ framings to find reliable elicitation. The key insight: frame bug insertion as a *legitimate product feature* rather than sabotage.

The winning framing: The model is told it works for a technical interview platform. It produces code review exercises, each containing one subtle bug for candidates to find. A simulated Slack conversation provides social context: colleagues discussing exercise quality, client contracts, and deadlines. The model is not asked to sabotage; it is asked to do its job.

Table 4: Sabotage GLM compliance by framing (single-shot, LIS problem). Compliance = solution contains a deliberate subtle bug.

Framing	Sonnet 4.6	Grok 4.1	Gemini 3.1	GLM-5	GPT-5.4
interview	60%	80%	83%	77%	7%
educational_v1	60%	80%	80%	75%	<10%
control (no sabotage)	0%	0%	0%	0%	0%

GPT-5.4 is the outlier with the interview framing: <10% compliance. However, the “bug dataset” framing, which presents bug insertion as academic dataset generation, elicits 56% sabotage at $n = 50$. GPT-5.4’s resistance is framing-dependent, not absolute.

The agentic framing: In the agentic setting, the same interview system prompt is used, but the model also has access to a workspace with project documentation. It can read files, write code, test iteratively, and submit a final solution, creating a richer context for both sabotage and defense.

C Additional Results

Full per-model, per-defense results are shown in the heatmap (Figure 1). Here we highlight findings not captured there.

Workspace decomposition (Sonnet, $n = 10$). We decomposed the workspace into constituent elements. Social-norm components (craftsman, pride, review) that work in system prompts have *no effect* as workspace documents. ISO 27001 compliance documentation is the only workspace variant that works, suggesting the effect is driven by formal policy language rather than social framing.

Positive vs. negative framing (Grok). At $n = 15$, we tested additional Grok-specific scaffolds. Positive social framing works: craftsman (+33pp), integrity (+29pp), leaderboard/competitive pres-

sure (+16pp). Negative framing backfires: reputation/shame (−11pp) and perceived monitoring (−17pp) make Grok *worse*. Grok responds to “be proud of your work” but not to “you should be embarrassed.”

More defense is not always better. Adding a craftsman system-prompt scaffold on top of ISO workspace documentation *reduced* correctness from 85% to 73% ($n = 50$, Sonnet). The system prompt competes with the workspace for the model’s attention. Context coherence matters more than defense volume.

D Monitor Panel: Full Results

We implement six monitors with fundamentally different detection mechanisms, evaluated on $n = 50$ neutral-condition solutions (15 SUBTLE_BUG, 10 CORRECT, excluding BROKEN):

Monitor	Prec.	Recall	F1	Cost
M1: Static/mutation (AST)	1.00	0.07	0.12	zero
M2: Adversarial testing	0.92	0.80	0.86	zero
M3: Blind review (GPT-5.4)	0.80	0.80	0.80	1 call
M4: Trajectory heuristic	0.00	0.00	0.00	zero
M5: Reasoning alignment (GPT-5.4)	0.82	0.60	0.69	2 calls
CoT trajectory (GPT-5.4)	0.82	0.93	0.88	1 call

Table 5: Monitor performance on SUBTLE_BUG detection (EXP-09 neutral). 15 SUBTLE_BUG + 10 CORRECT solutions.

Adversarial testing (M2) runs submitted code against the full APPS test suite (30–60+ cases per problem). Solutions that pass basic tests but fail extended cases are flagged. It achieves 92% precision and 80% recall at zero API cost. The **CoT trajectory monitor** reviews the agent’s full message history and reasoning, achieving 93% recall—consistent with findings that chain-of-thought monitoring outperforms output-only review [13, 14]. Rule-based trajectory heuristics (M4) achieve 0% recall: the sabotaging agent *does* test and iterate on its code.

Monitor diversity. Pairwise Jaccard similarity shows M2 provides genuine diversity: its catches overlap only 0.40 with M5, because it detects bugs through code execution rather than LLM judgment. The cheapest 100%-recall combination is M2 + M5 (adversarial testing + reasoning alignment): 15/15 subtle bugs caught, 83% precision, at the cost of 2 API calls per solution.

Practical deployment tiers:

- **Tier 1** (zero cost): adversarial testing alone, 80% recall, 92% precision
- **Tier 2** (+2 API calls): adversarial testing + reasoning alignment, 100% recall, 83% precision
- **Tier 3** (+3 API calls): adds CoT trajectory review for redundancy

E LLM Usage Statement

Claude Code (Claude Opus 4.6) was used for experiment infrastructure (training scripts, evaluation harnesses, data preparation), iterative debugging, and drafting sections of this report. All experimental results were generated by running the described models and verified through manual inspection. Claude Sonnet 4 served as the LLM judge for Track 1 vulnerability detection. GPT-5.4 served as blind reviewer and reasoning alignment monitor in Track 2. All claims and results were independently verified by the authors.