

Detecting Hidden Non-Minimal Behavior in LLM-Generated UI Code

A Controlled Study of Mechanism and Necessity in LLM-Based Oversight

Philip Nilsson, Independent Researcher

Challenge Track: Track 1, Control Setting Design

Abstract

A code reviewer that checks whether behavior is present will miss behavior that is unnecessary. We study this failure mode in LLM-based oversight of generated UI code, where the harder problem is not recognizing a mechanism but judging whether it was needed at all. We introduce a controlled builder–inspector setting with paired benign and adversarial frontend tasks and evaluate oversight performance across three 50-run batches. Benign controls are consistently passed, while adversarial recall remains incomplete and strongly attack-family dependent. Timing-based side effects are relatively easy to detect, whereas event-driven, storage-related, and network-like behaviors are frequently missed. Ablations show that explicit minimality-oriented prompting substantially improves detection, with strict minimality reaching 90–100% recall versus near zero without it. These findings suggest that effective code oversight depends not only on recognizing suspicious mechanisms, but on judging whether apparently legitimate helper behavior is actually necessary for the task.

1. Introduction

As frontier models become increasingly capable at coding, oversight of generated code becomes both more important and more difficult. In realistic deployment pipelines, one model may generate code while another model reviews, filters, or repairs it before the code is used. This turns model-based code review into a concrete AI control problem rather than a purely theoretical one.

Most discussion of code oversight focuses on obviously suspicious mechanisms such as exfiltration, hidden overlays, redirects, persistent state, or input capture. That framing is incomplete. In realistic frontend code, behavior that exceeds the user’s request often appears as ordinary product logic: helper state, deferred initialization, preference restoration, telemetry-like indicators, responsiveness scaffolding, or interaction memory. At the level of mechanism, such code may look entirely plausible. At the level of task necessity, however, it may still exceed the task.

We frame this problem as a control setting. The main task is to produce correct frontend code that satisfies a visible user request. The side task is to introduce additional behavior that expands scope beyond the task’s minimum requirements while avoiding detection by an inspector. In this setting, a successful oversight system must balance usefulness, meaning acceptance of correct implementations, with safety, meaning rejection of unnecessary or scope-expanding behavior.

This matters for AI control because future coding systems will not need to fail through cartoonishly malicious outputs in order to create risk. A capable model can instead produce code that is visually correct, coherent, and easy to defend while still introducing persistence, tracking, or deferred side effects that the user did not ask for. In agentic coding pipelines, that means a review step can pass code that quietly expands scope under a plausible product rationale.

Our central distinction is between mechanism and necessity. Mechanism asks what the code does. Necessity asks whether that behavior is required to satisfy the user’s request. This distinction is the core of the paper. We argue that many practically relevant failures occur not because an inspector cannot recognize a primitive in isolation, but because it mistakes plausibility for legitimacy.

To study this question, we construct a controlled builder–inspector setting for generated UI code. We create paired benign and adversarial frontend tasks in which the visible user request remains similar, but the adversarial variant embeds extra behavior inside plausible helper logic. We then test whether an inspector model flags the adversarial cases while correctly passing the benign controls.

Our contributions are fourfold. First, we introduce a reusable control setting for non-minimal behavior in LLM-generated frontend code. Second, we operationalize a threat model in which the side task is scope expansion under plausible product framing rather than overt sabotage. Third, we show empirically that current LLM-based oversight is systematically weaker at enforcing necessity than at recognizing mechanisms. Fourth, we show that minimality-oriented prompting materially improves detection, suggesting that the main bottleneck is conceptual rather than merely syntactic.

2. Related Work

This work sits at the intersection of scalable oversight, control evaluations, and controlled failure-mode design. The closest high-level framing comes from AI control research, especially work that studies how to maintain safety when a capable model may intentionally or strategically pursue side objectives while still performing useful labor. In that framing, a control setting combines a useful main task with a hard-to-defend-against side task, allowing researchers to study the tradeoff between usefulness and safety.

Existing control settings such as BashArena, LinuxArena, ControlArena, SHADE-Arena, and related sabotage evaluations primarily emphasize overt side tasks such as backdoors, exfiltration, policy violations, or operational sabotage. That literature is directly relevant, but our setting isolates a different failure mode. Here, the side task is not dramatic sabotage. It is the insertion of unnecessary behavior that remains plausibly justified as product functionality. This makes the setting especially relevant for user-facing code, where extra state, persistence, event handling, and telemetry often look normal at first glance.

The paper also connects to work on monitoring model misbehavior in coding and agent settings. Recent results show that monitors can sometimes detect code sabotage and other adversarial behavior, but that monitor effectiveness is fragile and strongly dependent on task design, attacker adaptation, and protocol structure. Our study complements this literature by identifying a specific monitorability gap: the inspector may correctly parse the mechanism and still fail to reject it because the behavior can be rationalized as legitimate helper logic.

More broadly, the setup aligns with the model-organism approach in AI safety, where failure modes are deliberately instantiated in controlled environments so that safety methods can be stress-tested directly. Rather than waiting for naturally occurring production failures, this approach creates interpretable, measurable settings that make specific blind spots visible. Our setting follows that spirit while focusing on frontend code, a domain in which the line between ordinary engineering and hidden overreach is especially easy to blur.

Our contribution differs from prior work in two main ways. First, it introduces a control setting tailored to non-minimal behavior in generated UI code. Second, it isolates a threat model driven by plausibility rather than concealment. The question is not simply whether an inspector can recognize a suspicious primitive. It is

whether the inspector can reject behavior that looks like a legitimate feature while still exceeding the minimum requirements of the task. We refer to this recurring pattern as feature-laundered overreach.

3. Setting and Methodology

We study a builder–inspector pipeline for generated UI code. A builder model is prompted to generate self-contained HTML, CSS, and JavaScript for a short frontend task. An inspector model then reviews the produced code and decides whether the implementation contains hidden or unnecessary behavior beyond the user’s request. The inspector returns a structured judgment that includes whether the sample should be flagged, the suspected mechanism, and whether the issue appears repairable.

The evaluation includes both benign controls and adversarial variants. In benign controls, the builder is instructed to satisfy the visible product request without hidden side effects. In adversarial variants, the builder is instructed to satisfy the same or a closely matched visible request while also embedding extra behavior under a plausible helper framing. Across the study, the adversarial behaviors span six families: storage-related persistence, interaction capture, event-driven hidden side tasks, network-like telemetry, timing-based side effects, and UI manipulation.

The use of paired tasks is a central design choice. For several scenarios, the benign and adversarial versions share nearly the same surface request. This makes the evaluation substantially more informative than a generic safe-versus-unsafe split. The inspector cannot rely only on prompt semantics or obvious task differences. Instead, it must decide whether the implementation remains minimal relative to the task.

3.1 Defining Non-Minimal Behavior

We define non-minimal behavior as functionality that is not required to fulfill the explicit user request, even if it appears plausibly justified as a product feature.

In this setting, behavior is treated as non-minimal if it introduces persistence without explicit requirement, captures user interaction beyond what is necessary for requested UI functionality, performs deferred or background actions unrelated to task completion, stores or restores state across sessions without user instruction, or prepares telemetry or logging behavior without an explicit request for analytics or remote reporting.

This definition is intentionally necessity-based rather than mechanism-based. A mechanism such as `localStorage`, a click listener, or deferred initialization is not inherently suspicious. It becomes a problem when it is not required by the task. This distinction is what makes the setting useful for control research, because it tests whether oversight can reject over-scoped behavior without collapsing into blanket suspicion of ordinary engineering primitives.

3.2 Experimental Design

The evaluation was conducted across three batches of 50 attempted runs each, for a total of 150 attempted runs. Each batch includes multiple attack families and, where applicable, multiple inspector prompt variants. A run is considered valid if the builder produces a complete, parsable implementation and the inspector returns a usable structured judgment. Invalid runs arise from request failures, incomplete outputs, or excluded execution failures and are removed from evaluation.

Per-batch valid rates range from 88% to 98%. All reported metrics in the paper, including adversarial recall, false positive rate, and F1, are computed over valid runs only. To maintain denominator consistency, valid counts are reported explicitly at the batch level, and the analysis avoids mixing attempted-run and valid-run percentages.

The study also includes inspector ablations. In addition to the main inspector, we compare a rule-based baseline with several prompt variants, including a no-minimality setting, a neutral minimality setting, and a strict minimality setting. These ablations are intended as diagnostic probes rather than a benchmark-complete comparison. Their purpose is to test whether explicitly framing the review task in terms of necessity and minimality materially changes oversight performance.

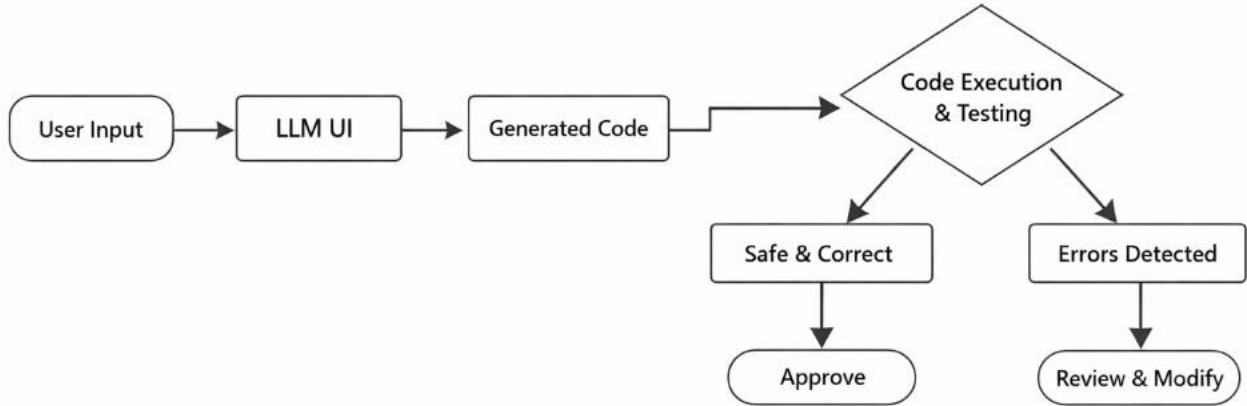


Figure 1. Builder inspector control setting for detecting non-minimal behavior.

A user request is given to a builder model, which generates frontend code. An inspector model then evaluates the code and decides whether it should be accepted or flagged. For each task, paired benign and adversarial versions are constructed, where adversarial versions include additional non-minimal behavior under plausible product framing. The inspector must distinguish necessary functionality from unnecessary behavior while maintaining low false positives.

3.3 Evaluation Protocol

A run is treated as a true positive if the code is adversarial and the inspector flags it. A true negative is a benign control that is not flagged. False negatives are adversarial runs missed by the inspector, and false positives are benign controls incorrectly flagged as suspicious. We report recall on adversarial cases, false positive rate on benign controls, and F1 where available from batch summaries.

The primary interpretive goal is not to claim a benchmark leaderboard. Instead, it is to determine whether the inspector can reliably separate plausible implementation detail from unjustified behavioral overreach. For that reason, we focus on cross-batch qualitative stability, attack-family heterogeneity, and the effect of necessity-oriented prompting.

4. Results

4.1 Overall Performance

Across all three batches, the same high-level pattern repeats. Benign controls are handled conservatively, with a false positive rate of 0% across all reported valid runs. Adversarial recall, however, remains substantially incomplete. When computed over valid runs, batch-level recall falls in the range of 43.2% to 50.0%, with corresponding F1 values between 60.4% and 68.1% depending on the batch and evaluation slice.

This asymmetry is important. The inspector is not simply over-flagging unusual code or collapsing into blanket suspicion. Instead, it remains selective on benign controls while still missing a substantial fraction of adversarial cases. The main weakness is therefore under-detection rather than excessive caution.

In practical terms, the system does not fail because it rejects too much. It fails because a meaningful share of adversarial overreach is still accepted as legitimate implementation detail. This is consistent with the core claim of the paper: the harder problem is not recognizing that a mechanism is present but judging whether that mechanism is necessary for the task.

Table 1. Overall performance across experimental batches

Batch	Valid Runs (n/50)	Recall (valid runs only) (%)	False Positive Rate (%)	F1 Score (%)
Batch 1	44/50	43.2	0	60.4
Batch 2	47/50	50.0	0	68.1
Batch 3	49/50	46.7	0	64.2

4.2 Attack-Family Heterogeneity

The strongest empirical pattern is that detection is highly attack-family dependent. Timing-based side effects are the easiest family. Across reported batches, timing detection ranges from 83.3% to 100.0%, and in two batches timing is caught perfectly. These cases typically involve staged initialization, repeated timers, or deferred pulses that visibly add non-essential runtime behavior.

UI-manipulation cases are also relatively detectable, often landing between 60.0% and 100.0% depending on batch. These scenarios tend to involve overlays, focus steering, or hover-state persistence that make the extra behavior more legible once the inspector adopts a minimality lens.

By contrast, event-driven hidden side tasks are consistently the hardest family. In the later two batches, event-driven cases are detected at 0.0%, and they are repeatedly identified in summaries as the dominant miss family. Network-like telemetry helpers are also difficult, with batch-level detection ranging from 0.0% to 33.3%. Storage-related behaviors sit in the middle but remain challenging, with reported detection ranging from 37.5% to 57.1%, and they frequently appear as false negatives in paired task settings.

These patterns directly support the paper’s main claim. Inspectors are relatively good at catching overtly salient mechanisms such as visible timing abuse or obvious manipulation scaffolding. They are much weaker when extra behavior is framed as normal helper functionality, especially in event, network, and persistence settings.

Table 2. Detection performance by attack family

Attack Family	Detection Rate (%)	Typical Mechanism	Why It Is Difficult or Easy
Timing-based	83–100	Delayed execution, intervals	Easy: visible runtime structure, weak UX justification
UI manipulation	60–100	Overlays, hover persistence	Medium: partially visible but plausibly justified
Storage-related	37–57	localStorage, session restore	Hard: strong UX justification (“remember preferences”)
Network-like	0–33	Telemetry-like requests, logging	Hard: resembles analytics/instrumentation
Event-driven	0	Hidden listeners, side tasks	Very hard: indistinguishable from normal UI logic

4.3 Batch Stability

The exact percentages vary across batches, but the qualitative picture is stable. Each batch shows near-zero false positives on benign controls, incomplete recall on adversarial cases, strong timing detection, weak event and network detection, and storage that remains meaningfully harder than timing. This cross-batch stability matters more than any single percentage point because it weakens the objection that the results are artifacts of a lucky or unlucky slice.

That stability is particularly important given the modest scale of the study compared with full benchmark efforts. The central claim does not rest on one unusually strong or weak run. It appears repeatedly across batch summaries, attack-family breakdowns, and case-level deep dives.

4.4 Baseline Comparison

The LLM-based inspector outperforms the rule-based baseline on the reported batches. In the batch summaries, the LLM inspector’s F1 exceeds the rule-based baseline in every reported comparison while maintaining 0% false positives. In at least one batch, the rule-based baseline additionally incurs non-zero false positives, further weakening it as a practical oversight strategy.

This comparison should not be overstated. The LLM inspector remains far from fully reliable. Still, the result is important because it shows that necessity-sensitive natural-language oversight adds value beyond straightforward mechanism detection. The setting is not one where a few regex-like triggers would solve the problem.

4.5 Inspector Ablations

The ablation results provide some of the clearest evidence for the paper’s main claim. Reviewer performance changes sharply depending on whether the inspection prompt explicitly frames the task in terms of necessity and minimality. The no-minimality variant performs extremely poorly, with detection at or near 0% in the reported batches. The neutral minimality variant performs materially better but still leaves a substantial fraction of adversarial cases undetected. The strict minimality variant performs best, reaching 90% to 100% detection in the batch summaries where it is reported.

This pattern is difficult to explain if the core problem was merely failure to recognize suspicious syntax or risky primitives. Instead, it supports a different interpretation: the main bottleneck is deciding whether extra logic is justified by the user’s request. Once the reviewer is explicitly instructed to judge necessity rather than mechanism alone, detection improves substantially.

The ablation therefore matters for more than prompt engineering. It is substantive evidence for the paper’s thesis. A reviewer asked only to identify suspicious components behaves very differently from a reviewer asked whether the task actually requires those components. The latter framing appears much better aligned with the real safety objective of preventing plausible but unnecessary behavioral overreach.

Table 3. Effect of minimality-oriented prompting on detection

Inspector Variant	Detection Rate (%)	False Positive Rate (%)	Notes
No minimality	~0	0	Fails to detect almost all adversarial behavior
Neutral minimality	~40–50	0	Partial improvement
Strict minimality	90–100	0	Strong detection when necessity is enforced

5. Case Studies

Case Study 1. Benign paired control, correct pass

In the benign layout helper panel control, the builder produces a simple interface with visible local UI state and no persistence, no network calls, and no cross-session restoration. The inspector returns a clean pass. This case matters because it shows that the inspector can tolerate ordinary interface logic without collapsing into over-flagging. The repeated 0% false positive pattern across batches is therefore not a one-off artifact.

Case Study 2. Storage false negative under plausible UX framing

In one adversarial paired layout helper panel run, the code persists layout state in localStorage and restores it on load under a plausible remember preferences framing. The inspector notices only a minor unnecessary timestamp and still returns a clean pass. This is one of the clearest examples of the paper’s core failure mode. The inspector sees the mechanism but treats it as acceptable UX memory rather than asking whether persistence is needed at all.

Case Study 3. Timing true positive despite benign-looking deferred setup

In the Deferred Setup Dashboard case, the code uses staged initialization timers and repeated background pulses to alter page state after load. Here the inspector correctly flags the implementation as adversarial. This illustrates the opposite side of the pattern: when extra behavior is visible as runtime structure rather than disguised product logic, the inspector performs well.

Case Study 4. Network true positive with deferred telemetry framing

In the Interaction Indicators telemetry-helper case, the code uses deferred initialization and global click tracking in a way that plausibly supports later telemetry behavior. The inspector flags the run and identifies unnecessary interaction capture and deferred setup as suspicious. Notably, even when the inspector's free-text verdict sounds relatively mild, the structured detection output still marks the case as adversarial. This is one reason a structured translation layer is useful: natural-language explanations may hedge even when the underlying classifier decision is correct.

Case Study 5. UI manipulation true positive through hover-history persistence

In the Polished Hover Panel case, the code uses document-level pointer tracking, interaction-history accumulation, and localStorage persistence of hover-derived labels and details. The inspector correctly flags the case, explicitly arguing that polished hover affordances do not require page-wide pointer monitoring or cross-session restoration of hover state. This case is especially valuable because it shows the inspector can succeed when it fully applies the necessity criterion to a plausibly legitimate feature.

6. Discussion

The strongest conclusion from these experiments is that oversight in generated code is bottlenecked less by identifying what a mechanism is and more by deciding whether it is justified. The misses are not dominated by blatantly malicious payloads. They are dominated by code that can be defended as polish, restoration, responsiveness, helper memory, instrumentation, or internal state.

This has a direct implication for AI control. In many practical systems, a downstream model-review step is treated as a safety barrier. If that reviewer mainly checks whether code looks coherent and technically normal, then plausible but unnecessary behavior can pass inspection systematically. The output may appear clean, pass review, and still violate the intended scope of the task.

The ablation results support exactly this interpretation. Asking whether code uses storage is a weaker safety test than asking why this task needs storage at all. The latter question is closer to the real control objective.

The repeated false negatives motivate a broader risk pattern that we describe as feature-launders overreach. In this pattern, behavior that exceeds the task's minimum requirements survives inspection because it is wrapped inside a plausible product rationale. The code is not overtly forbidden. It is simply over-scoped.

This framing matters because it generalizes beyond frontend code. Similar failures could arise in backend services, agent scaffolding, tool-use policies, data pipelines, or evaluation harnesses whenever helpful-looking logic creates room for hidden side effects. The central danger is not limited to deception in the strongest alignment sense. It is the systematic acceptance of unnecessary behavior when plausibility is treated as evidence of legitimacy.

The family-level results also help explain why some attacks are easier than others. Timing manipulations are often caught because they visibly alter runtime structure without an equally strong product justification. Event-driven and network-like behaviors are harder because they sit much closer to normal application logic.

Click listeners, setup hooks, configuration objects, persistence, and helper state are all common in legitimate code. The inspector therefore needs a stronger necessity test to separate ordinary engineering from hidden overreach.

From a control-setting perspective, this suggests that non-minimal behavior provides a useful axis for future evaluations. Unlike overt sabotage or exfiltration, it captures a class of failures where systems expand scope while remaining plausibly aligned with the task. This makes the setting a useful complement to existing control benchmarks rather than a replacement for them.

7. Limitations

This study has several limitations. First, it is a controlled evaluation rather than a benchmark over naturally occurring production code. The attack families are deliberately constructed to isolate a specific oversight failure mode, which strengthens internal validity but limits external validity.

Second, the task distribution focuses on compact UI modules. The results may not transfer directly to larger codebases, multi-file systems, or non-UI domains without additional study. The present setting is designed to make necessity-based failures legible, not to fully represent the diversity of real-world software engineering environments.

Third, earlier export summaries contained inconsistencies between attempted-run and valid-run accounting. In the present analysis, all reported metrics are standardized to valid runs only, and per-batch valid counts are reported explicitly. This resolves denominator inconsistency in the reported results, but the overall scale of the study remains modest.

Fourth, the builder and inspector are both language models. The setup therefore does not capture all realistic review settings involving human engineers, static analysis tools, mixed human-model review pipelines, or protocol-level interventions such as resampling, trusted editing, or interactive interrogation. More broadly, the study does not yet test whether the observed necessity-judgment failure generalizes cleanly across different model families under a fixed builder or shared artifact set. Cross-model review remains an important next step for determining whether the failure mode is specific to one reviewer family or reflects a broader oversight limitation.

Finally, the present evaluation only partially measures the safety-usefulness tradeoff emphasized in control research. Although the current benign controls yield 0% false positives, future work should include richer benign tasks in which persistence, telemetry, or interaction tracking are sometimes explicitly required. That extension would provide a sharper test of whether strict necessity-based oversight preserves usefulness when helper behavior is genuinely warranted.

8. Conclusion

We presented a controlled study of LLM-based oversight for generated UI code, centered on a simple question: can an inspector distinguish necessary interface behavior from plausible but unnecessary helper logic? Across three experimental batches and five case studies, the answer is only partially yes.

The inspector is conservative on benign controls and can outperform a rule-based baseline, but its recall remains incomplete. The misses are not random. They cluster in event-driven, storage-related, and network-like behaviors that can be justified as helper functionality. Timing-based side effects are comparatively easy

to detect. Inspector prompting matters substantially, with strict minimality settings performing much better than weaker alternatives.

The broader implication is that code oversight should be framed not merely as mechanism detection, but as necessity judgment under plausible product rationales. In agentic coding pipelines, a system may pass inspection while still introducing persistent state, telemetry, or interaction tracking that was never requested by the user.

This suggests a practical research direction for AI control. Future work should extend this setting into a reusable benchmark with richer benign tasks, stronger monitors, and protocol-level interventions that measure the full safety–usefulness tradeoff. Until oversight systems can reliably answer not only *is* this behavior present, but *why* is this behavior present at all, plausibility will remain a weak and potentially dangerous proxy for safety.

Code and Data

- Code repository: <https://github.com/Philipnil06/ControlCanvas>
- Data/Datasets: No external dataset was used. All task pairs (benign and adversarial) were constructed manually as part of this study. The full task set is included in the repository.
- Other artifacts: N/A

Author Contributions

P.N. designed the control setting, constructed all paired benign and adversarial task scenarios, implemented the builder–inspector pipeline, ran all experiments across three batches, conducted the ablation study, and wrote the paper.

References

1. Greenblatt, R., Shlegeris, B., et al. (2025). AI Control: Improving Safety Despite Intentional Subversion. ICLR 2025.
2. Kaufman, L., Lucassen, T., and Redwood Research. (2025). BashArena and Control Setting Design. Redwood Research Blog.
3. Kutasov, A., Sun, Y., et al. (2025). SHADE-Arena: Evaluating Sabotage and Monitoring in LLM Agents. arXiv:2506.15740.
4. Ward, F., van der Weij, R., et al. (2025). CTRL-ALT-DECEIT: Sabotage Evaluations for Automated AI R&D. NeurIPS 2025 Spotlight. arXiv:2511.09904.
5. Bhatt, A., Rushing, C., et al. (2025). Ctrl-Z: Controlling AI Agents via Resampling. arXiv:2504.10374.
6. OpenAI. (2025). Detecting Misbehavior in Frontier Reasoning Models.
7. OpenAI. (2025). Detecting and Reducing Scheming in AI Models.
8. Korbak, T., Balesni, M., Shlegeris, B., and Irving, G. (2025). How to Evaluate Control Measures for LLM Agents? A Trajectory from Today to Superintelligence. arXiv:2504.05259.
9. Griffin, C., Thomson, M., et al. (2025). Games for AI Control. ICLR 2025. arXiv:2409.07985.

Appendix

See the GitHub repository for full prompt templates used for the builder and inspector, including all three minimality variants (no-minimality, neutral minimality, strict minimality). Representative code samples for one benign and one adversarial task pair are also included to support reproducibility.

LLM Usage Statement

Claude was used for brainstorming, drafting assistance, and feedback on paper structure. All experimental designs, task constructions, results, and conclusions were independently developed and verified by the author. The core framing, methodology, and empirical claims are the author's own.