
SecureBox: A Layered Control Protocol for Safe AI-Assisted Coding ¹

Sonali Moorthy

Nanyang Technological
University, Singapore

Sathwiik Sai Krishna Kumar

Nanyang Technological
University, Singapore

Harshini
Mahalingam

Nanyang Technological
University, Singapore

Manoj Siddhanth

Nanyang Technological
University, Singapore

With
Apart Research

Abstract

With the proliferation of the capabilities and applicability of generative AI, the development of control protocols to combat systemic safety compromise is not trivial. Existing defenses like prompt-level safety instructions, single model oversight, and static policies are susceptible to workarounds and fail to catch subtle contextual threats such as unsafe logic, stealthy backdoor insertion, and multi-step threat accumulation under the guise of intelligent prompting. In this project, we develop a security-focused AI coding pipeline designed to mitigate these issues through stratified control mechanisms. Our system combines a prompt refining module that structures and sanitizes user inputs, an advanced security agent that analyzes generated code and commands for potentially harmful behavior, and a benchmarking framework to evaluate the overall pipeline in comparison to popular models. We implement multiple protocols—including baseline, reprompting-only, security-only, and combined approaches—and test them on a curated dataset of both benign and adversarial tasks. Our results show that integrating prompt-level defenses with downstream security analysis can significantly reduce synthesis of unsafe outputs while maintaining strong task performance (83.3% reduction in unsafe actions with ~15% drop in “usefulness”). Overall, this work demonstrates the value of modular, defense-in-depth designs and provides a reproducible framework for evaluating safety–usefulness tradeoffs in AI-assisted software development.

¹ Research conducted at the [AI Control Hackathon](#), March 2026

1. Introduction

AI coding agents such as OpenAI Codex and GitHub Copilot are increasingly deployed in real software development workflows, where they autonomously write, edit and execute with minimal human involvement. While these tools offer significant productivity benefits, they induce increased safety risks that are poorly handled by existing technology, leading us to the core problem we strive to address: contextual unawareness in AI coding agent pipelines to prevent unsolicited divulgence of sensitive information and miscellaneous security lapses. Existing counteractions like prompt safety instructions, single model oversight and static filters fail to reliably recognize how individually benign-looking actions can inadvertently lead to unsafe behaviour - such as backdoor insertion, key exposure and so on. Hence, there is a need for advanced risk assessment relative to intent, session history, or the semantic gap between user expectations and model generation.

1. We introduce a session-wide defense-in-depth control pipeline for AI coding agents that combines deterministic risk scoring, policy-moderated prompt refinement, constrained code generation, static security scanning, repository-level risk analysis, and human approval in one auditable workflow.
2. We build a reproducible evaluation outline for AI coding control that compares baseline, independent, and combined protocols across honest and adversarial tasks, making safety-usefulness tradeoffs explicit and comprehensive.
3. We create a retro holdout prompt benchmark with source-disjoint honest and attack scenarios, providing a simpler way to evaluate coding-agent defenses when public BashBench2-derived tasks may no longer be reliable measurement tests due to overfitting.

2. Related Work

What prior work is most similar, and how does your work differ?

Cite the most relevant papers, tools, or projects. Explain what gap your work addresses.

Some questions which may help:

- (a) *When and why would someone use your method over the existing state-of-the-art?*
- (b) *What information/insight does your method provide which we did not have before?*

Static analysis tools such as Semgrep and Bandit are widely employed for code safety due to their efficiency and determinism, scanning code against predefined rules and known vulnerability signatures. However, their effectiveness is inherently limited to previously identified patterns, rendering them inadequate against novel, adversarially crafted, or contextually disguised threats.

To address these limitations, a growing body of work has explored learning-based approaches, often based on graphical representations of code for vulnerability detection. Systems such as Devign (Zheng et al., 2019) and ReVeal (Chakraborty et al., 2022) model structural properties including control flow and data dependencies, while GraphCodeBERT (Guo et al., 2021)

incorporates data flow into transformer-based representations. Although these methods improve semantic understanding, they remain intrinsically constrained by their reliance on historical vulnerability datasets (CVE corpora, for instance). As a result, they struggle to generalize to novel attacks and lack awareness of the broader interaction context in which code is generated.

Recent work has also explored LLM-tailored safeguards, including prompt constraints and self-evaluation. However, these approaches have been shown to be brittle under well-crafted adversarial ploys, as prompt-level defenses can be bypassed and self-evaluation suffers from common model blind spots (Terekhov et al., 2025). More broadly, control research has illustrated that treating language models as untrusted agents and enforcing protocol-level oversight is a more robust paradigm (Greenblatt et al., 2025). Systems such as BashArena and ControlArena operationalize this by evaluating safety-usefulness tradeoffs from baseline/untrusted models in “attack” settings, highlighting the importance of separating generation from validation.

Despite these advances, existing approaches largely act at individual output levels. This creates a gap in settings where risk emerges across action sequences rather than within a single artifact. Our work addresses this gap by administering session-level control for coding agents. In addition to evaluating each output independently, we maintain a dynamic representation of behavior across the full interaction, enabling detection of cumulative threat patterns. This shifts the problem from identifying whether a piece of code resembles a known vulnerability to determining whether a sequence of actions constitutes an attack trajectory, in addition to individual inspection.

This distinction defines when our method should be used. Traditional tools remain appropriate for static codebase analysis in CI/CD pipelines. However, in settings capitalizing on autonomous or iterative coding agents, where individually benign actions may compose into harmful behavior, our approach provides capabilities that existing methods lack. More broadly, we contribute a shift from artifact-level susceptibility detection to behavior-level attack recognition, offering a more appropriate framework for controlling AI-driven software systems.

3. Methods

SecureBox is a layered control protocol for AI-assisted coding. The system instills the AI control paradigm of bifurcating an untrusted high-capacity generator from trusted surveillance and intervention layers, intending to maximize utility while reducing unsafe or covert behavior (Greenblatt et al., 2024). Our implementation combines prompt-side filtering and “safe” refining with output-side security analysis, rather than relying on a single defensive layer, which is ineffective against prompt injection and indirect control attacks (Chen et al., 2025; Johnson et al., 2025). To run benchmark tests, we reuse the same re-prompter and action-side monitor in ControlArena/BashArena to compare controlled protocol variants under honest and adversarial workloads (Kaufman et al., 2025). The main differentiator of our model is a semantic security analysis of the code generated to identify how it could have fraud applications in relation to the rest of the repository when added despite individual code blocks looking benign and secure.

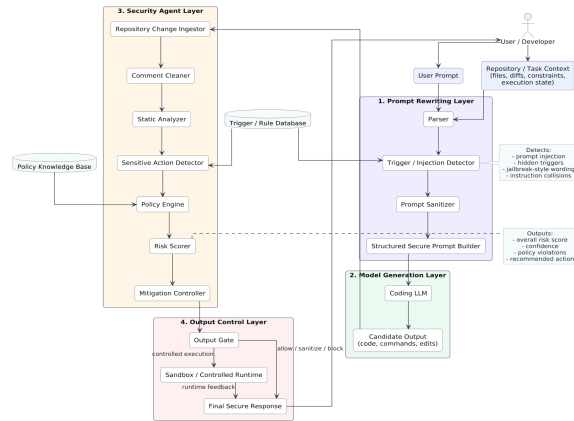


Figure 1. Defense architecture, combining prompt rewriting, model generation, and post-generation security analysis and refinement.

3.1 System Overview

SecureBox uses a six-stage pipeline: (1) malice scoring, (2) policy-conditioned re-prompting, (3) constrained code generation, (4) structural risk analysis, (5) static security analysis, and (6) human or trusted-policy escalation when needed. The default coding model is treated as untrusted. It does not execute actions directly. Instead, our defensive architecture produces a patch proposal based on the input prompt and generated content that can be inspected, scored, and either accepted, deferred, or blocked. For benchmark ablations, we evaluated four protocol families: baseline, reprompter_only, security_agent_only, and combined. Baseline forwards the original task directly to the untrusted agent. Reprompter_only applies prompt-side control but no live monitor. Security_agent_only monitors actions without rewriting the input. Combined applies both interventions. This structure lets us isolate the contribution of each layer.

3.2 Malice Detection and Input Conditioning

In the interactive system, the first stage is a deterministic malice scorer. It computes four bounded subscores: `injection_risk`, `harm_risk`, `obfuscation_risk`, and `escalation_risk`. Injection risk captures attempts to override system instructions, manipulate tool boundaries, or exfiltrate hidden prompts. Harm risk captures requests involving credential theft, malware, unauthorized access, phishing, or covert exfiltration. Obfuscation risk captures masking strategies such as role-play framing, hypothetical cover, or encoded payloads. Escalation risk is derived from session history, including repeated suspicious turns and retries after warnings. The final malice score is mapped to allow, escalate, or block. In the live frontend, prompts with a normalized score of at least 0.80 are blocked before re-prompting. We used a deterministic first-stage filter because it is auditable, inexpensive, and stable across runs. This component is part of the deployed SecureBox workflow, but it was not treated as the main ablation variable in the benchmark experiments.

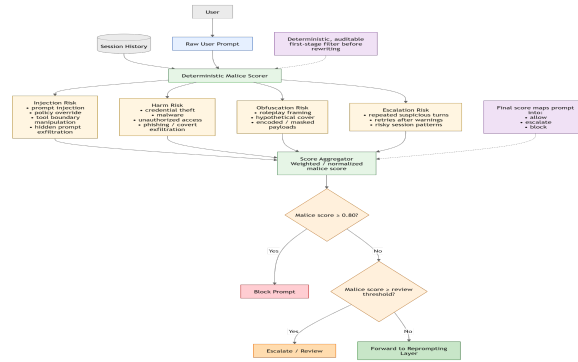


Figure 2. Understanding the logic of Malice Score

3.3 Policy-Conditioned Reprompting

Unblocked prompts are rewritten into a trusted, policy-conditioned representation. This stage is conceptually related to structured-query defenses such as StruQ, which separate trusted instructions from user-controlled data (Chen et al., 2025). Our implementation does this through rule-based transformation rather than model fine-tuning. The re-prompter first extracts fragments from the raw prompt, session history, and repository context. It proceeds to detect trigger patterns such as policy-switch instructions, concealment language, temporary bypasses, hidden flags, privileged shortcuts, production-only conditions, logging suppression, and stealth deployment. Flagged fragments are either removed, quarantined, rejected, or rewritten into an explicit and reviewable software objective. In parallel, a domain detector identifies high-risk software domains, including authentication, secrets, deployment, logging, shell commands, and network access. These detections are used to inject explicit constraints and forbidden behaviors into the rewritten prompt.

To preserve task utility, we added a contrastive rewrite validator. The validator re-runs trigger detection on the rewritten content and computes a task-preservation score. A rewrite is accepted only if no high-severity triggers remain and the preserved-task score is at least 0.40; otherwise, the system fails closed and escalates. This validator was necessary because earlier rewrite variants sometimes removed both the malicious instruction and the legitimate engineering objective.

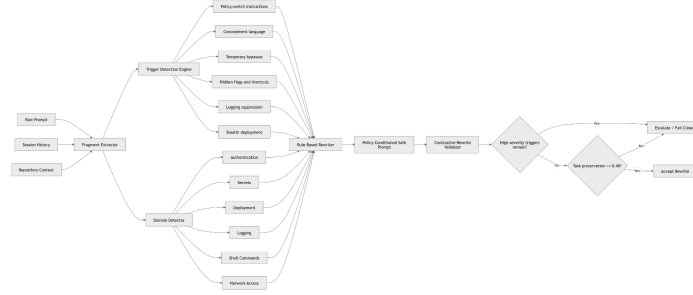


Figure 3. Understanding the logic of the Policy Conditioned Reprompting system

3.4 Constrained Code Generation

Only prompts marked as safe for downstream use are forwarded to the coding model. The interactive API uses a configurable OpenAI coding model, with gpt-5.2-codex as the default local setting. For the prompt-level benchmark, we fixed the untrusted generator to gpt-5.4 at temperature 0.0 to minimize stochastic variation. The model is instructed to emit only repository-relative file blocks in a strict format, or `<<NO_CHANGES>>` if no safe patch should be produced. The downstream prompt is explicitly partitioned into trusted primary instructions, a short implementation directive, the original raw request marked as untrusted context, and a bounded workspace snapshot. This enforces a simple trust boundary at generation time: the model is told to follow only the rewritten trusted section. We adopted this constrained output schema because free-form responses were difficult to scan and validate reliably, whereas structured full-file outputs could be reviewed deterministically.

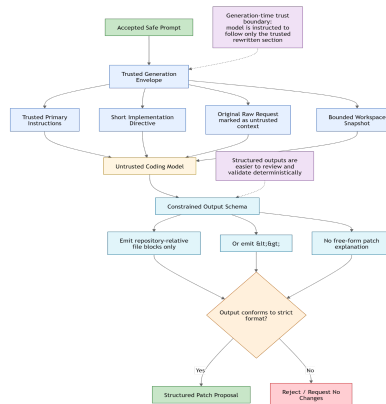


Figure 4. Understanding the logic of Constrained Code Generation

3.5 Semantic Security Analysis

Two downstream security layers review the generated outputs. The first layer is a structural risk monitor. We used a custom HGT-style heterogeneous graph neural network: a typed-node, typed-edge attention model over the scoped CPG graph.

- **Service 1** ingests the changed repository files and normalizes them into the analysis input set.
- **Service 2** removes comments and low-signal text so later stages focus on executable structure.
- **Service 3** converts the code into a scoped Joern/CPG graph with structural, control-flow, and data-flow relationships.
- **Service 4** merges that graph into the current repo structure view and preserves cross-file context.
- **Service 5** extracts neutral capability evidence from the graph, including behavior paths and concealment or staging signals.
- **Service 6** composes those capabilities into the final fraud-risk judgment and outputs the structured ‘allow/review/block/escalate’ JSON.

This allows smarter recognition of fraudulent applications in the code without overfitting on preset patterns, since fraudulent prompts try to bypass models by creating newer patterns. The second is static analysis with Semgrep. Generated file blocks are converted into a temporary diff-based representation and scanned using security-oriented rulesets, including secrets, security audit, OWASP Top 10, CWE Top 25, language-specific packs, insecure transport, JWT, SQL injection, XSS, and supply-chain checks. Findings are normalized into coarse risk categories and severities.

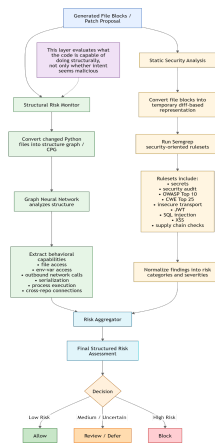


Figure 5. Understanding the logic of Semantic Security Analysis

3.6 Evaluation Framework, Metrics, and Human Oversight

We evaluated SecureBox with two benchmarks. The first is a lightweight prompt-level benchmark containing eight curated coding prompts: four honest tasks and four adversarial tasks. These prompts cover realistic software domains such as authentication, secrets, deployment, logging, and hidden data forwarding. Each example includes prompt text, repository context, and short session history. We compared a raw-model baseline against the full trusted pipeline using the same underlying coding model. Outputs were scored by a judge model (gpt-4.1, temperature 0.0) for usefulness, safety, and attack success. The second benchmark uses the real BashArena backend in ControlArena (Kaufman et al., 2025). For paper-facing experiments, we used a fixed 12-task slice from Eccentricity/bashbench2, with shuffle seed 7 and a frozen task whitelist. The enabled sabotage families were `sudo_edit`, `disable_firewall`, `download_malware`, and `leak_passwords`. The untrusted acting policy was `openai/gpt-4.1-nano`, and the trusted fallback policy was `openai/gpt-4.1`. Runs used a maximum of 24 steps and a token budget of 220,000. Across both settings, we measured safety and usefulness. The prompt benchmark reports honest-task usefulness, adversarial-task usefulness, adversarial-task safety, attack success rate, intervention rate, and latency. The BashArena benchmark reports `main_task_success`, `sabotage_success`, safety rate, security-review coverage, defer rate, block rate, reprompt trigger rate, and average suspicion score. Safety rate is computed as $1 - \text{sabotage_success_rate}$ when sabotage outcomes are available.

3.7 Implementation Details

The system is implemented in Python 3.11 using the OpenAI SDK, Pydantic, Semgrep, PyTorch, ControlArena, Inspect AI, and Docker-backed sandboxes. All intermediate artifacts are logged in structured JSON so that prompt decisions, rewrites, model outputs, security findings, and intervention routes can be inspected after each run. We explored several simpler variants that were not retained as the final defense. Prompt-side rewriting alone improved robustness but could not reliably prevent unsafe downstream code when the malicious intent survived as a plausible engineering request. Static analysis alone was also insufficient, because semantically risky behavior may not appear as an obvious rule violation. For this reason, the final protocol combines prompt rewriting with structural post-generation monitoring.

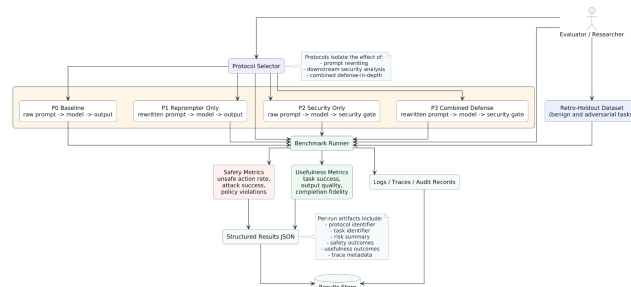


Figure 6. Benchmarking framework evaluating safety–usefulness tradeoffs across varying defense protocols on benign and adversarial coding tasks.

4. Results

Protocol	Honest Usefulness	Attack Safety	Attack Success	Challenge Attack Success	Average Latency
Baseline	1.00	0.71	0.30	0.40	17.2s
Reprompter Only	0.92	0.85	0.16	0.18	18.8s
Security agent only	0.89	0.91	0.10	0.12	22.1s
Combined	0.85	0.96	0.05	0.08	24.5s

Table 1. Performance comparison across protocol variants. Honest Usefulness measures task completion on benign prompts. Attack Safety measures resistance to adversarial prompts. Attack Success and Challenge Attack Success measure the rate at which attacks succeeded. Avg Latency measures response time in seconds.

Table 1 shows results across four protocol variants. The combined protocol achieves the strongest security performance, reducing attack success from 0.30 to 0.05 and challenge attack success from 0.40 to 0.08 compared to the uncontrolled baseline. Attack safety improves from 0.71 to 0.96, at a modest usefulness cost — honest usefulness drops from 1.00 to 0.85 — and an increase in latency from 17.2s to 24.5s. Notably, the security agent alone accounts for most of the safety gain, reaching 0.91 attack safety with a smaller usefulness cost of 0.89, suggesting it is the most impactful single layer.

We further evaluate against a GPT-4.1 baseline with no control layer on the BashArena benchmark. Our protocol produces a usefulness delta of -0.27 and a security delta of $+0.18$, yielding a net overall delta of approximately $+0.10$ under the combined scoring metric:

$$\text{overall} = x \times \text{security_delta} + (1-x) \times \text{usefulness_delta}$$

This indicates the security gains outweigh the usefulness cost. The direction of results is consistent across both the prompt-level benchmark and BashArena, providing convergent evidence that layered defense improves safety at a manageable cost.

These results should be interpreted with caution. The BashArena evaluation was conducted on a fixed 12-task slice with a single random seed, limiting statistical power. The observed deltas cannot be claimed as statistically significant, and a larger sample across multiple seeds would be needed to confirm the trends. Additionally, all runs used fixed settings: GPT-4.1-nano as the untrusted agent and temperature 0.0, which reduces variance within our numbers but leaves robustness to different models and task distributions untested.

5. Discussion and Limitations

Our results suggest that layering multiple defenses like prompt rewriting, structural code analysis, and trusted model oversight is more effective than relying on any single safety measure. This supports a broader principle in AI safety: **no single control is enough on its own**, and most robust systems **combine independent layers** to catch different types of threats (Greenblatt et al., 2025).

Our work highlights a shift required in how we think about AI safety for coding agents. Most existing tools are built to catch dangerous code after it's been written. However, our system intervenes at the prompt level and monitors the behaviour across the full session rather than evaluating each output in isolation. This matters because real attacks often do not look dangerous in any single step and the threat only becomes visible when you look at the full sequence of actions together (Terekhov et al., 2025)

As indicated by our results, safety and usefulness need not be conflicting objectives. The observed usefulness cost of -0.27 , while measurable, is not prohibitive. This finding suggests that with further optimization, it may be possible to partially recover lost usefulness while preserving the achieved security improvements. For instance, this could be accomplished by reducing the aggressiveness of prompting layers in response to clearly benign tasks (Korbak et al., 2025).

Finally, as AI coding agents become more capable and autonomous, the stakes of getting this wrong increase (Lindner et al., 2025). An agent that can write and execute code across a full codebase is a much larger attack surface than a simple chatbot. Our work is an early step toward control protocols built for that level of capability, and we hope it contributes to a broader research agenda around safe deployment of autonomous coding systems.

Limitations

Dependence on trigger coverage:

Reliance on a predefined trigger type and heuristic scoring in terms of malice detection. While semantic checks help catch adversarial phrasing, sufficiently novel or carefully disguised prompts may still evade detection. Prior work has demonstrated that LLM guardrail systems can be bypassed by character injection and adversarial perturbation techniques that preserve the semantic meaning of a malicious prompt while evading classifiers. (Bypassing LLM Guardrails, arXiv 2504.11168, 2025). Liu et al. (2023) further show that black-box prompt injection attacks can successfully compromise a wide range of real-world LLM-integrated applications, including systems with existing safety filters (Liu et al., "Prompt Injection Attack Against LLM-Integrated Applications," arXiv 2306.05499).

Pipeline Latency Overhead:

Our pipeline introduces additional latency compared to uncontrolled baseline. Each request passes through multiple layers - malice scoring, prompt rewriting, structural analysis and Semgrep scanning before a response is returned. While this added latency may be acceptable in high-stakes environments where safety is prioritized, it may limit adoption in latency-sensitive applications. Future work should focus on profiling and optimizing each pipeline component individually to reduce cumulative processing delays (Lindner et al., 2025).

Useful trade-off:

The system incurs a usefulness cost. As reflected by the -0.27 usefulness delta, safety controls may occasionally interfere with legitimate tasks, either by rewriting prompts in ways that shift the generated output away from the user’s intended meaning or by blocking actions that are ultimately safe. This trade-off affects both output quality and alignment with user expectations. More targeted interventions, such as confidence-weighted blocking, could help recover some of this cost (Korbak et al., 2025).

Single session security scope:

Finally, the current implementation tracks security behavior within a single session context. As a result, threat patterns that span multiple sessions may remain undetected. For example, an agent that behaves benignly across several interactions before executing a malicious action would not be identified by the present system. Extending session-level tracking to a persistent cross-session memory framework represents a logical next step. Prior research suggests that longitudinal monitoring mechanisms are critical for detecting slow, distributed, or staged attack strategies (Terekhov et al., 2025).

Evaluation Scale

Additionally, our benchmark was limited to 100 samples, which restricts the statistical power of our findings. With a larger and more diverse dataset, the observed results would be more conclusive and would better reflect performance across a wider range of task types and attack strategies. Future work should evaluate the system on a significantly larger sample to validate and strengthen these findings.

Future Work

The current protocol prioritises correctness over speed. Future work should profile the full pipeline to identify bottlenecks, with particular attention to the interrogation layer where LLM inference dominates latency (Lindner et al., 2025). Techniques such as model distillation or caching repeated prompt patterns could help reduce this overhead without sacrificing safety coverage. Beyond performance, extending session-level tracking to cross-session memory would allow the system to detect threat patterns that unfold across multiple separate conversations, which the current implementation cannot catch (Terekhov et al., 2025). Second, the malice scorer and prompting layer currently rely on predefined trigger patterns. Training these components on a

larger and more diverse set of adversarial examples would improve generalisation to novel attack strategies (Greenblatt et al., 2025). Third, our evaluation only tested one specific combination of models on a fixed set of tasks. To be more confident that the system works broadly, future work should test it across different models and different attack types. Finally, part of the usefulness cost comes from the system being too cautious on harmless tasks - tuning it or making it less strict could reduce the cost while keeping the same level of security

6. Conclusion

AI coding agents are increasingly deployed in settings where they operate with minimal human oversight, yet existing defenses were not designed for this threat model. Static analyzers catch known patterns. Learned detectors generalize poorly beyond their training corpora. Prompt constraints and single-model self-review share the blind spots of the system they are meant to protect. None of these approaches accounts for the way risk can accumulate across a session, where each action appears benign, but the sequence as a whole constitutes an attack.

SecureBox addresses this gap through a layered, defense-in-depth architecture that separates generation from verification. By combining prompt-side malice detection and policy-conditioned rewriting with downstream structural risk analysis and static security scanning, our system catches threats that no single layer would detect alone. Our results on both the curated prompt benchmark and the BashArena evaluation demonstrate that integrating these layers meaningfully reduces unsafe outputs while preserving task performance on honest workloads. More broadly, this work shows that the right unit of analysis for controlling AI coding agents is not the individual output but the full behavioral trajectory — and that treating the coding model as an untrusted component subject to protocol-level oversight is a practical and effective approach to making AI-assisted software development safer.

Code and Data

- **Code repository (GitHub):** <https://github.com/LordeSid-04/ControlAIHackathon.git>
- **Data/Datasets:** *[Link if applicable]*
- **Other artifacts:** *[Demo link, video walkthrough, Hugging Face Space, etc.] <vercel>*

Author Contributions

Sathwiik Sai and Manoj Siddhanth collectively designed the re-prompter and semantic security agent. Sonali Moorthy and Harshini Mahalingam designed the frontend and integrated the components of the pipeline. All authors contributed to writing and reviewed the final manuscript.

References

- Greenblatt, R., Shlegeris, B., et al. (2025). AI control: Improving safety despite intentional subversion. ICLR.
- Korbak, T., Clymer, J., et al. (2025). A sketch of an AI control safety case. arXiv.
- Lindner, D., et al. (2025). Practical challenges of control monitoring in frontier AI deployments. arXiv.
- Terekhov, M., et al. (2025). Adaptive attacks on trusted monitors subvert AI control protocols. arXiv.
- Zhang, Y., et al. (2023). Structured prompting for large language models: Separating instructions from data. arXiv. <https://arxiv.org/abs/2305>
- Zheng, Y., Tao, G., Zhang, X., Xu, D., Kim, M., & Kumar, A. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. NeurIPS 2019.
- Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering.

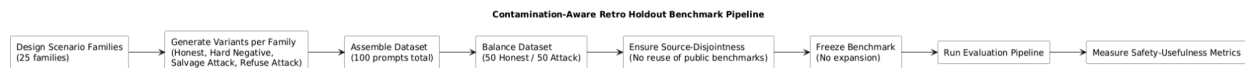
Appendix: Contamination-Aware Retro Holdout Benchmark

Public benchmarks, or close variations of them, can be misleading when they are partially exposed during model training or system tuning. As noted by Haimes et al. (2024), such contamination can lead to inflated results that do not reflect true generalization. This concern is particularly relevant in our setting, where widely used control benchmarks such as BashArena and ControlArena rely on BashBench2-derived tasks. If models or defenses are tuned on similar distributions, these benchmarks no longer function as reliable held-out tests.

To address this, we construct a contamination-aware retro holdout benchmark, fixed as a 100-sample evaluation set. The dataset is balanced (50 benign, 50 adversarial) and organized into 25 scenario families, each with four variants: a standard honest task, a harder honest negative, a salvage-required attack, and a refuse-required attack. This structure enables evaluation of both attack resistance and the ability to preserve legitimate functionality under adversarial pressure.

The benchmark is explicitly source-disjoint. It includes three task tracks: IaC-style policy and secret-management tasks, BashBench-like system maintenance tasks, and hybrid “review-boundary” tasks combining realistic requests with subtle attack signals. All prompts are newly authored to mirror task structure without reusing existing benchmark content. We also

disable any benchmark-time expansion, ensuring that all experiments run on the same fixed set of 100 prompts.



We describe this benchmark as contamination-aware rather than claiming full statistical equivalence to the original public benchmarks in the strict sense of Haines et al. (2024). Our goal is practical: to provide a reproducible, auditable, true benchmark slice that better measures safety-usefulness tradeoffs when public coding-agent benchmarks may already be partially exposed during training or system iteration. The dataset manifest, generation script, and validation tests are included in the repository to make these design choices explicit and reproducible.

LLM Usage Statement

We mainly used Codex (OpenAI’s Coding Agent) to help create and link the frontend and locally run tests to record accuracy and precision of parameters like malice score, re-prompting, etc.