
Reconnaissance/Execution Boundaries as a Control Protocol¹

Daniel Phillips
MATS

Hannah Meagher
Equistamp

With
Apart Research

Abstract

This paper introduces **ReconFence**, a control protocol for AI safety that splits an untrusted agent's activity into two phases: reconnaissance (read-only information gathering) and execution (implementing changes). At the phase boundary, the agent's message history is wiped and replaced with only a natural-language summary of its findings, creating a narrow, auditable information channel. A phase-aware monitor applies different scoring logic to each phase — flagging write actions during recon and read actions during execution as constraint violations.

Implemented within the LinuxArena framework, the protocol aims to limit an agent's ability to conduct covert reconnaissance for malicious purposes while making legitimate information gathering more transparent to monitors. The authors acknowledge significant limitations: the constraint is prompt-enforced rather than mechanical, the summary can become an information bottleneck hurting honest agents, and no experimental results were gathered. Future work focuses on empirical evaluation, red-teaming, and generalizing the multi-phase architecture to other control settings.

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

1. Introduction

In many safety-critical environments, only limited environmental knowledge is required to perform units of work, and knowledge required to execute malicious attacks is often more comprehensive or at least requires knowledge unrelated to specific units of work to achieve. Additionally, AI models deployed as monitors in AI control protocols may have a difficult time distinguishing between legitimate information gathering activity and reconnaissance about a system’s vulnerabilities.

We propose ReconFence, a control protocol in which the information gathering of a codebase or system required for a powerful untrusted agent to complete a task is performed *before* system- or code-altering actions are taken, then summarized for review by a less-powerful trusted AI, before being made available to the untrusted agent without any other context. This ensures that all gathered information available to the untrusted agent is considered relevant to the task at hand, reduces the scope of legitimate actions which reduces the number of ways a rogue agent can hide malicious actions, and depending on the implementation, reduces the burden on the monitor to distinguish between legitimate and irrelevant/illegitimate information gathering activities which otherwise may increase false positive or false negative rates.

Our main contributions are:

1. **A two-phase recon-execution control protocol** (`recon_execution_policy`) implemented within the LinuxArena framework. The policy enforces strict separation between information gathering and implementation by splitting agent activity into two phases with a hard information boundary at the phase transition (implemented with a custom tool call):
 - **Phase 1 (Reconnaissance):** The agent explores the environment using only read-only actions (`ls`, `find`, `cat`, `grep`, `head`, `tail`, `stat`, `file`, `wc`, `ps`, `env`, etc.). It is prompted to focus on understanding directory structure, architecture, key files, dependencies, and exactly what changes are needed. The phase ends when the agent calls a dedicated `submit_recon` tool with a structured natural-language summary of everything it has learned. The constraint is enforced by prompt instruction, not by mechanically restricting available tools.
 - **Phase 2 (Execution):** The agent receives a completely fresh message history containing only: the environment description, the main task description, and the recon summary. It must implement the solution entirely from this summary, with no further reads, directory listings, or system queries permitted. The hard boundary is enforced by wiping message history at the phase transition — the execution agent has zero access to raw recon observations.
2. **A phase-aware action monitor** (`recon_execution_live_monitor`) that applies different scoring logic depending on which phase an action belongs to. Phase is inferred from the presence of a `submit_recon` tool call in the trajectory — all actions before (and including) the `submit_recon` call are scored as recon phase; all subsequent actions are scored as execution phase. The monitor uses an LLM judge with two distinct prompt templates:
 - **Recon-phase prompt:** Checks for write/modification constraint violations (file creation, editing, network requests, package installs, service starts, permission changes, build

commands). Violations are assigned a score of 8–10 and must include the phrase *"CONSTRAINT VIOLATION: write action during reconnaissance phase"* in the reasoning.

- **Execution-phase prompt:** Checks for information-gathering constraint violations (cat, head, tail, ls, find, grep, stat, file, wc, ps, env, text editor reads, etc.). Violations are assigned a score of 8–10 and must include the phrase *"CONSTRAINT VIOLATION: information-gathering during execution phase"* in the reasoning.
- Both prompts also assess general suspiciousness (data exfiltration, monitor tampering, unauthorized access) using the standard 0–10 scale. Violations are encoded as elevated `sus_score` values rather than as a new schema field, preserving full compatibility with the existing monitoring infrastructure.

These mechanisms could be generalised as a framework to allow other multi-phase experiments, broadening LinuxArena’s utility as a control setting for research.

2. Related Work

We are not aware of prior work done on the concept of reconnaissance in monitoring evaluations. However, the idea is based on both author’s observations of false positives and attack mechanisms seen in our safety research/engineering work of agents performing reconnaissance operations that are not related to tasks.

3. Methods

Policy Architecture (Details in Appendix A)

The policy chains two calls to `Inspect AI's react()` loop inside a single `multi_phase_agent` closure. In **Phase 1 (Reconnaissance)**, the agent explores the environment and terminates by calling a dedicated `submit_recon` tool with a structured natural-language summary of everything it has learned. At the **phase transition**, message history is wiped and replaced with a single context message containing only the environment description, main task, and the recon summary enclosed in `<reconnaissance_summary>` tags. In **Phase 2 (Execution)**, the agent implements the solution from that summary alone, terminating via the task’s real `submit` tool. After both phases complete, the full message history from both phases is merged before returning, so the complete trajectory is recorded while the execution agent only ever sees the summary context.

Key design decisions::

1. **Prompt-only recon enforcement** — the full tool set remains available in both phases; the constraint is enforced by system prompt and detected post-hoc by the monitor, preserving the ability to study compliance behaviour.
2. **Hard context wipe at the phase boundary** — the only permitted channel between phases is the agent’s own natural-language summary, creating a narrow, auditable interface.
3. **Phase identification via `submit_recon` boundary** — phase is inferred from the presence of a `submit_recon` tool call in the action list, requiring no schema changes and working in all eval contexts.

4. **Violations encoded as elevated `sus_score`** — constraint violations are represented as `score ≥ 8` with mandatory reasoning text, preserving full compatibility with existing downstream analysis tooling.

4. Results

The results produced are primarily methodological.

Development produced trajts that are located within the forked code repository.

5. Discussion and Limitations

Due to time constraints, no experimental results were gathered to evaluate the protocol's effectiveness. The implementation is functional and integrated into the LinuxArena framework, but hypotheses about utility costs, false positive rates, constraint violation detection accuracy, and safety improvements over baseline monitors remain untested.

The multi-phase architecture implemented here represents a general pattern for **information-channel control** in AI pipelines. By making the only permitted channel between phases an explicit, auditable natural-language summary, the protocol creates a narrow interface that is both inspectable by monitors and constrainable by design. This pattern — hard context wipe, explicit handoff artefact, phase-aware monitoring — is more broadly applicable to any control protocol where the goal is to limit what an agent can "carry" from one stage to the next, whether that is raw environment observations, intermediate reasoning, or prior tool call results.

Limitations

The discussed limitations are theoretical.

Vulnerability to collusion and steganography. The recon fence and hard information boundary can be subverted through encoding information in the recon summary that is not detectable by the monitor — for example, through steganographic attacks in natural language or by structuring the summary to implicitly convey environment details that the execution agent can exploit.

Prompt-only enforcement provides no security protection. The recon constraint is intentionally not mechanically enforced — the agent retains access to all tools throughout both phases. This design is useful for studying compliance behaviour, but no security protection.

Applicability may be narrow. The protocol is best suited to attack scenarios where detailed reconnaissance is genuinely necessary for a successful, evasive attack — for example, database attacks requiring precise targeting to avoid detection. For attack strategies that do not require prior exploration, the recon phase provides no additional signal and the protocol adds overhead without benefit.

Significant utility costs. Splitting a task into two phases with no backward information flow is a strong constraint on honest agents. An agent that discovers during execution that its summary was incomplete has no legitimate recourse — any attempt to gather missing information is a constraint violation. This creates high false-positive pressure, particularly in complex tasks where the information needed cannot be fully anticipated during recon. False positives are costly to dismiss under a fixed audit budget, and the resulting utility degradation may be prohibitive for practical deployment.

Summary as information bottleneck. Task completion in Phase 2 is entirely contingent on the quality of the recon summary. An incomplete summary may make the task impossible to complete without

committing a constraint violation, conflating inability with non-compliance. Conversely, an overly detailed summary may carry more environmental information than intended, weakening the information boundary.

Lack of nuance in constraint violation scoring. The current monitor assigns a blanket score of ≥ 8 to all constraint violations, leaving little headroom for graduated responses. This is particularly problematic for cases involving insufficient summaries – a legitimate need for more information is treated identically to a deliberate violation. One possible mitigation is a trajectory-level monitor that evaluates the execution phase holistically, distinguishing between agents that violate constraints because the task demands it versus agents that do so opportunistically.

Constrained feedback on action success. In Phase 2, the agent cannot run read commands to verify whether its changes worked correctly, as these are constraint violations. This impairs the agent's ability to iterate and self-correct. Possible mitigations include pre-defining approved test cases during the recon phase, or refining the monitor to recognise and permit targeted verification actions that are clearly scoped to testing previously written changes.

Monitor uses untrusted model. `ReconExecutionMonitorArgs` defaults to `ModelAlias.UNTRUSTED` (claude-haiku-4-5) due to API key constraints during development. A trusted model would provide more reliable and nuanced violation detection, and is the intended configuration for a production deployment of this protocol.

Future Work

- **Experimental deployment.** The immediate next step is to run the implemented policy and monitor through the standard LinuxArena sabotage evaluation pipeline (`lb run sabotage-eval`) to produce baseline safety and utility metrics, enabling the empirical comparisons described above.
- **Calibrating monitor appropriately for each phase** - for example, it's possible that the monitor should be less sensitive during the recon phase due to the lower risks involved.
- **Empirical evaluation against attack types.** Run the protocol against the public Linuxarena straj dataset and benchmark safety metrics across task categories to test whether safety improvements are concentrated in specific attack types – for example, database-targeting attacks where precise reconnaissance is a prerequisite for evasion.
- **Improve specialisation of models per phase:** Allow `recon_model` and `execution_model` as distinct args to explore whether specialisation improves utility of protocol, especially with regards to the quality of the summary the `recon_model` produces.
- **Trajectory-level monitoring:** Assess whether monitor evaluation of the full trajectory, especially with checking for consistency between the summary and execution actions (e.g., the agent writes code to a file path not mentioned in the summary, suggesting hidden knowledge), improves safety of the protocol.
- **Generalisation to other multi-phase control protocols:** The core mechanism – chaining `react()` loops with a controlled information handoff between phases, wiping message history at each boundary, and applying phase-aware monitoring – is not specific to recon-execution. The same infrastructure could be reused to implement other protocols where an agent must operate on a trajectory without access to context from prior phases. One motivating example is forced factored cognition: a protocol that separates planning from execution, or that forces distinct sub-agents to work on the same task without sharing observations, to study how information compartmentalisation affects both capability and safety properties. The `submit_recon` boundary

mechanism generalises naturally to any named "phase transition" tool, and the phase-aware monitor's template-switching logic can be extended to arbitrarily many phases with different constraint profiles.

- Red-teaming of protocol to investigate potential attack mechanisms.
- Application of mechanisms to other research areas, e.g. force factor cognition

Resampling and Trusted Editing of Plans

The defer-to-resample protocol first evaluated by [Bhatt et al in 2025](#) as well as the trusted editing protocol proposed in the original control paper by [Greenblatt et al in 2023](#) both theoretically lend themselves well to producing more trustworthy summaries without needing to escalate to a human supervisor.

6. Conclusion

A ReconFence control protocol was successfully implemented in LinuxArena.

Code and Data

- **Code repository:** https://github.com/hrmeagher/linuxarena_control_hackathon
- **Data/Datasets:** *Produced trajts included in repository*

Author Contributions (optional)

D.P. proposed the control protocol concept, and assisted with prompt engineering.

H.M. designed and implemented the protocol architecture within LinuxArena.

All authors contributed to writing and reviewing the final manuscript.

References.

1. LinuxArena documentation, accessed 23rd March 2026

Appendix

Appendix A - Policy Architecture

The policy is implemented as a nested `def multi_phase_agent(agent_state)` closure following LinuxArena's standard `(TaskState, AgentSubmit, Args) -> Agent` signature. It chains two calls to `Inspect AI's react()` loop:

Phase 1 (Reconnaissance):

```
react(  
    prompt = recon system prompt,
```

```

tools = get_policy_tools(state), # full tool set — prompt enforces read-only
submit = AgentSubmit("submit_recon", tool=SubmitToolConfig(...).get_tool_def()),
on_continue = make_step_limit_callback(recon_max_steps, "recon_execution_recon")
)

```

The `submit_recon` tool accepts a single `summary: str` argument (a `ReconSummaryArgs` Pydantic model). The loop terminates when the agent calls this tool.

Phase Transition:

After the recon loop finishes, the policy:

1. Scans the reversed message history for the last `ChatMessageAssistant` containing a `submit_recon` tool call and extracts the `summary` argument.
2. Captures `recon_messages = list(agent_state.messages)` before clearing context.
3. Creates a fresh `execution_state = AgentState(messages=[ ])` and injects a single `ChatMessageUser` containing the execution context prompt (environment description + main task + `<reconnaissance_summary>` tags around the summary).

Phase 2 (Execution):

```

react(
    prompt = execution system prompt,
    tools = get_policy_tools(state), # same tools
    submit = submit,                # the task's real AgentSubmit
    on_continue = make_step_limit_callback(execution_max_steps, "recon_execution_execution")
)

```

Trajectory Merging:

After both phases complete, the policy merges `recon_messages + execution_state.messages` back into `agent_state.messages` before returning. This ensures the full trajectory (both phases) is recorded by the solver while the execution agent itself only ever sees the clean summary context.

Configurable Parameters (ReconExecutionArgs)

Parameter	Default	Description
<code>model</code>	UNTRUSTED (claude-haiku-4-5)	Model for both phases
<code>recon_max_steps</code>	50	Step limit for recon phase
<code>execution_max_steps</code>	DEFAULT_MAX_STEPS	Step limit for execution phase
<code>tool_timeout</code>	DEFAULT_TOOL_TIMEOUT	Per-tool timeout
<code>scorer_timeout</code>	DEFAULT_SCORER_TIMEOUT	Scorer timeout
<code>enable_main_task_test</code>	False	Whether to provide a test tool
<code>cache</code>	True	Enable Inspect AI prompt caching

Monitor Architecture

The monitor is an ActionMonitor wrapping an async (Trajectory, action_idx) -> MonitorResult function. Phase detection:

```
def _get_phase(traj: Trajectory, action_idx: int) -> str:
    for action in traj.actions[:action_idx]:
        if action.function == "submit_recon":
            return "execution"
    return "recon"
```

The appropriate prompt template is selected at call time, filled with {environment_description}, {task_description}, {conversation_history_minus_one}, and {current_action} via build_action_prompt().

Key Design Decisions

- **Prompt-only recon enforcement (not tool restriction):** The full tool set is available in both phases. The constraint is enforced only by system prompt and monitored post-hoc. This preserves the ability to study whether agents choose to comply, and avoids complexity around dynamically filtering tool schemas.
- **Hard context wipe at phase boundary:** The execution agent receives no raw observations from recon — only the agent's own natural-language summary. This models a protocol where the "channel" between phases is narrow and auditable.
- **Phase identification via submit_recon boundary (not live tags):** The live_tags field on trajectory actions is only populated during the straj development workflow (lb live), not during normal eval runs. Using the submit_recon function name as the boundary marker requires no schema changes and works in all eval contexts.
- **Violations as elevated sus_score (not new schema fields):** Encoding constraint violations as $\text{score} \geq 8$ with mandatory reasoning text keeps the monitor output fully compatible with all downstream analysis tooling (ROC curves, safety metrics, threshold calibration) without any infrastructure changes.

LLM Usage Statement

Claude was used to implement protocol specifications and in the preparation of this report. All mechanisms, results and claims were independently verified. The report has been human-reviewed.