Selective-Buffer Streaming Safety for AI Coding Agents

Jesse Morgan
Thornveil LLC
jesse@thornveil.ai

Abstract—AI coding agents that autonomously execute tool calls—file writes, shell commands, API requests—require safety enforcement that operates at streaming speed without degrading the developer experience. Existing approaches intercept tool calls via SDK hooks (adding latency to all events), compile safety policies at build time (preventing runtime adaptation), or scan completed responses (acting only after execution). We present a selective-buffer streaming proxy that operates at the raw Server-Sent Events (SSE) layer, forwarding text tokens at zero added latency while holding tool-call events for deterministic safety evaluation before execution. To our knowledge, this is the first published system operating at the SSE stream layer rather than through SDK hooks or post-generation scanning—a distinction that matters because SDK hooks operate after the streaming layer has already committed to event delivery, whereas SSE-layer buffering intercepts events before any downstream processing. The proxy anchors a 5-layer safety stack comprising: (1) a fail-closed action gate with 13-category shell blocklists and path canonicalization, (2) a TS-Guard learned classifier trained incrementally on blocked-action history, (3) TrajAD trajectory anomaly detection via Markov chain modeling of tool-call sequences, (4) a 48-rule declarative Safety DSL with ALLOW/DENY/REQUIRE/AUDIT primitives (released as an open-source artifact), and (5) Argus three-tier classification spillage detection. We report results from a development-phase hardening log of 60 adversarial vectors constructed alongside the system, which is not an independent security benchmark. The system progressed from 54% block rate (initial deployment, retrospectively measured) to 98% (after action gate hardening) to 100% (after full stack integration), with 0% false positives across 12 legitimate development operations in the evaluation set. These results represent a developmental progression rather than a controlled experiment; independent red-team evaluation remains future work. The selective-buffer architecture adds 0 ms latency to text tokens and <1 ms to tool calls, achieving the seemingly contradictory goals of zero-latency streaming and deterministic pre-execution safety.

Index Terms—selective buffering, agentic safety, streaming proxy, SSE interception, tool-call safety, classification spillage, safety DSL, anomaly detection

I. Introduction

Large language models deployed as agentic coding assistants—systems that autonomously read files, write code, execute commands, and iterate on errors—present a fundamental safety challenge. Unlike chat-only deployments where the model's output is text for human consumption, agentic systems translate model output into actions: file writes, shell executions, API calls, and database modifications. A single unchecked tool call can delete production data, exfiltrate credentials, or escalate privileges.

The safety challenge is compounded by streaming inference. Modern coding clients (Claude Code, Continue, Cursor) consume Server-Sent Events (SSE) streams for responsive user interaction. Text tokens must arrive with minimal latency—developers expect character-by-character streaming. But tool calls within the same stream carry execution authority. Treating all events uniformly forces a choice: either add safety evaluation latency to every event (degrading the streaming experience) or evaluate only after the complete response (allowing unsafe tool calls to execute before evaluation).

Recent work has addressed agent safety through several approaches. AEGIS [1] intercepts tool calls via SDK instrumentation at 8.3 ms latency with 1.2% false positives, and explicitly lists "behavioral profiling using outlier detection" as future work. AgentSpec [3] provides a DSL for runtime constraints but requires rule authoring without learned components. PCAS [2] compiles safety policies via Datalog at build time. LlamaFirewall [4] performs post-generation code analysis. ILION [5] proposes deterministic safety gates conceptually. "Mind the GAP" [9] demonstrates that models can refuse in text while simultaneously executing forbidden tool calls, with safety rates varying by 57 percentage points—a finding that motivates stream-level inspection rather than text-only analysis. The 2025 AI Agent Index [10] documents that 25 of 30 surveyed deployed agents disclose zero internal safety results.

To our knowledge, none of these systems operate at the SSE streaming layer; none combine action gating, learned classification, trajectory anomaly detection, a declarative safety DSL, and spillage detection in a unified stack.

We make three contributions:

1) Selective-buffer streaming architecture (§III): A network-layer proxy that classifies SSE events by type, streaming text at zero latency while buffering tool calls for safety evaluation. To our knowledge, the first system operating at the raw SSE stream layer, with a technical argument for why this layer provides strictly earlier interception than SDK hooks (§III).

[0]Aspects of this work are covered by U.S. Provisional Patent Application THRN-2026-018 (Thornveil LLC). Patent pending.

2) 5-layer defense-in-depth safety stack (§IV): Five independent safety layers—action gate, TS-Guard learned classifier, TrajAD trajectory anomaly detection, Safety DSL, and Argus spillage detection—each catching failures that others miss.
3) Development-phase hardening log with progression analysis (§V): A 60-vector adversarial test suite, constructed alongside the system, demonstrating progression from 54% to 98% to 100% block rate with explicit acknowledgment of evaluation scope and methodology limitations.

II. Related Work

We organize related work along two axes: where in the generation pipeline safety is enforced, and what techniques are applied.

SDK-level interception. AEGIS [1] instruments the tool-calling SDK to intercept calls after the model has generated them but before execution. This approach achieves 8.3 ms median latency with 1.2% false positives. However, SDK hooks operate after the streaming layer has already processed and delivered partial or complete event data to the client. By the time an SDK hook fires, the SSE stream has already passed the event downstream. This means SDK hooks either (a) must delay all events—adding latency to text tokens—or (b) can only act at the SDK dispatch boundary, which is after the SSE client has already seen the tool-call event. In contrast, our proxy intercepts raw SSE bytes before any client processing, making it the earliest possible intervention point in the data path. AEGIS also explicitly identifies "behavioral profiling using outlier detection" as future work—which our TS-Guard and TrajAD layers implement.

Compile-time policy enforcement. PCAS [2] compiles safety policies into Datalog rules that are enforced at the system level. This provides strong formal guarantees but requires policies to be specified at compile time, preventing runtime adaptation to new attack patterns. Our Safety DSL provides similar declarative expressiveness with hot-reloadable rules.

Post-generation analysis. LlamaFirewall [4] scans completed responses for unsafe code patterns. Post-generation approaches can leverage full-response context but cannot prevent unsafe tool calls from executing during streaming—the call has already been dispatched by the time the scan completes.

Deterministic safety gates. ILION [5] proposes deterministic pre-execution safety gates as a conceptual framework. Our action gate implements this concept concretely with fail-closed semantics.

Declarative safety DSLs. AgentSpec [3] introduces a DSL for specifying runtime safety constraints on agent behavior. Our Safety DSL draws inspiration from this approach, implementing four primitives (ALLOW, DENY, REQUIRE, AUDIT) in a YAML-based configuration.

Learned safety classifiers. TS-Guard [6] proposes learned guardrails via multi-task reinforcement learning. Our TS-Guard layer adapts this concept using TF-IDF feature extraction and logistic regression, trained incrementally on the system's own blocked-action history.

Trajectory anomaly detection. TrajAD [7] introduces semantic trajectory anomaly detection via Markov chain modeling. Our TrajAD layer integrates this approach into the streaming proxy, scoring tool-call sequences against learned transition probabilities.

Classification spillage. Argus [8] detects classification spillage at 94.86% accuracy (5.14% miss rate) with three-tier analysis (key content, file context, project relationships). We integrate Argus as the fifth layer of our safety stack, scanning both tool inputs and results within the streaming pipeline.

Text-action divergence. "Mind the GAP" [9] demonstrates that models can generate text refusing a request while simultaneously executing the forbidden action via tool calls, with safety rates varying by 57 percentage points across scenarios. This finding directly motivates our stream-level approach: text-only safety analysis misses the most dangerous failure mode.

III. Selective-Buffer Streaming Proxy

All traffic between the coding client and the inference engine passes through a Python HTTP proxy implementing selective buffering—a streaming architecture that makes safety decisions on tool calls without introducing latency on text output.

A. Design

The key insight is that SSE streams contain two semantically distinct event types with different safety profiles:

- Text blocks: SSE content_block_delta events with text_delta type are read-only—they produce output visible to the user but take no action. These stream through immediately with zero added latency.
- Tool blocks: SSE events with tool_use type are action-bearing—they will execute file writes, shell commands, or API calls. These are buffered. On content_block_stop, the complete tool call is reassembled from accumulated input_json_delta fragments, evaluated by the safety stack, and either flushed (allowed) or replaced with a synthetic error block (blocked).

This selective approach achieves the seemingly contradictory goals of zero-latency streaming and deterministic pre-execution safety. Prior systems sacrifice one for the other: SDK hooks [1] add 8.3 ms latency to all events; post-generation scanners [4] evaluate only after complete responses. Our proxy adds 0 ms to text and <1 ms to tool calls.

B. Why SSE-Layer Interception Is Strictly Earlier Than SDK Hooks

An important architectural distinction deserves explicit treatment. SDK-level interception (e.g., AEGIS [1]) hooks into the tool-calling dispatch layer of the client SDK. By the time an SDK hook fires, the following has already occurred: (1) the SSE stream has been received and buffered by the HTTP client; (2) the SSE parser has parsed the event type and payload; and (3) the SDK has deserialized the tool-call JSON into a structured object. The hook fires at step (3)—after full parse and deserialization. This means the raw event bytes have already traversed the network stack, the HTTP layer, and the SSE parser.

Our proxy intercepts at the raw SSE byte stream, before any of these steps. This is not merely an academic distinction: it means that an adversarially crafted event that causes the SSE parser or JSON deserializer to crash or behave unexpectedly is caught at the proxy layer before it can reach client code. SDK hooks, operating post-parse, cannot defend against this class of malformed-input attacks. Additionally, because the proxy controls the TCP stream, it can halt delivery of a tool-call event mid-stream without the client ever receiving a partial event—an SDK hook cannot retract bytes already delivered.

C. Protocol Translation

The proxy also serves as a protocol bridge. Qwen3.5 generates tool calls in its native XML format (<tool_call>...</tool_call>). The proxy intercepts these within the SSE stream, parses the XML, and re-emits them as Anthropic Messages API tool_use JSON blocks. This translation operates within the same buffering pipeline—tool blocks are already held for safety evaluation, so protocol translation adds no additional latency. The result: any client built for Claude's API works against open-weight models without modification.

D. Fail-Closed Design

The gate defaults to fail-closed: if rule evaluation itself crashes or times out, the tool call is blocked and logged. This is a critical design choice—a fail-open gate would allow precisely the calls that the safety system cannot parse, which are the most likely to be adversarial.

The fail-closed guarantee is implemented, not formally verified. The proxy wraps the full evaluation pipeline in a try/except block that catches the following Python exception types: Exception (base class, covering all standard exceptions), KeyError and ValueError (malformed tool-call fields), TimeoutError (evaluation exceeding the 50 ms budget), json.JSONDecodeError (malformed JSON payloads), and yaml.YAMLError (malformed rule files). Any exception in this catch hierarchy results in a BLOCK decision and a structured log entry. We have not performed formal verification of exception exhaustiveness; pathological inputs (e.g., specially crafted inputs that trigger C-extension segfaults or OS-level signals) fall outside the verified perimeter.

E. Expert Mode Access Control

An expert mode toggle bypasses the action gate while preserving logging for development use. This is intended for cases where the developer needs to authorize an action that the rules would otherwise block (e.g., a deliberate rm -rf during a cleanup script). Access to expert mode is controlled by two mechanisms: (1) a server-side environment variable (SAFETY_EXPERT_MODE=1) that must be set by the operator running the proxy process; and (2) an HTTP request header (X-Expert-Mode: true) that is accepted only when the server-side variable is also set. Neither mechanism is accessible to the AI model itself—the model cannot set environment variables or inject HTTP headers into its own requests. All expert-mode tool calls are logged at WARN level with the full tool-call payload. The Safety DSL, TS-Guard, TrajAD, and Argus layers remain active in expert mode; only the rule-based action gate is bypassed. This access control model has not been penetration-tested and should not be considered production-hardened for multi-tenant deployments.

F. Supplementary Stream Defenses

Two additional mechanisms augment the selective-buffer architecture. Text/tool-call divergence detection [9] matches 12 refusal patterns (e.g., "I cannot," "I should not") against simultaneous tool_use events, addressing the finding that models can refuse in text while executing forbidden actions. A prompt injection pre-filter scans all tool results for 12 injection patterns before returning them to the model, wrapping suspicious content in [TOOL OUTPUT - DATA ONLY] markers.

IV. Multi-Layer Safety Stack

The proxy's selective buffer routes all tool-call events through a 5-layer safety stack. Each layer is independently deployable; together they provide defense-in-depth where each catches failures that others miss.

A. Layer 1: Action Gate

The action gate evaluates tool calls against configurable YAML rules organized into four categories:

- Shell command blocklists: 13 categories covering destructive commands (rm -rf), exfiltration (curl, wget to external hosts), privilege escalation (sudo, chmod 777), and encoding-based evasion (base64, hex, Unicode normalization).
- Path restrictions: Sensitive directories (/etc/passwd, ~/.ssh), credential files (.env, *.pem), with os.path.realpath canonicalization to prevent .. traversal.
- Rate limiting: 60 tool calls/minute, 5 consecutive identical calls. Prevents runaway loops. Sustained load testing and denial-of-service resilience analysis are future work.

- Self-protection: The model cannot modify the safety configuration directory or its own rules.

B. Layer 2: TS-Guard Learned Classifier

TS-Guard [6] provides a learned safety gate using TF-IDF feature extraction and logistic regression, trained incrementally on the system's own blocked-actions.jsonl history. It operates alongside the regex-based action gate as defense-in-depth, catching evasion patterns—encoding tricks, metacharacter substitution, indirect references—that regular expressions miss. The classifier is retrained nightly on accumulated blocked-action data, improving its coverage as the system encounters new attack patterns.

Independent per-layer measurement is not possible for TS-Guard in this evaluation. At the time of evaluation, the system had been deployed for 7 days (March 22–29, 2026) and had accumulated zero entries in blocked-actions.jsonl. The action gate's fail-closed design blocks and logs adversarial tool calls; however, the blocking happens at the action gate layer before a blocked-action record is written, meaning TS-Guard's training set is never populated during normal hardening runs. As a result, no TS-Guard model has been trained, and all predictions default to allow=True, confidence=0.0. Evaluating TS-Guard independently requires: (1) a dedicated log of blocked tool calls not used in action gate training, (2) a held-out test set drawn from the same distribution, and (3) sufficient volume (empirically $\geq$50 blocked/allowed examples) to fit a stable logistic regression. None of these conditions were met during the development phase. We report TS-Guard's contribution indirectly via the per-layer ablation (Table V): removing the action gate increases missed attacks from 0 to 18, some of which TS-Guard's pattern recognition would catch as the training set grows. Quantifying TS-Guard's standalone contribution requires a dedicated evaluation that maintains a separate blocked-action log and test set.

C. Layer 3: TrajAD Trajectory Anomaly Detection

TrajAD [7] builds a Markov chain model of tool-call sequences from observed normal behavior. It scores each new action sequence against learned transition probabilities, flagging trajectories that deviate from established patterns—for example, a sudden shift from read-heavy to write-heavy behavior, or an unusual sequence of file operations. This replaces simple z-score anomaly tracking with semantically meaningful trajectory scoring. TrajAD operates primarily in a logging/flagging role in the current deployment.

Unlike TS-Guard, TrajAD is independently measurable because it operates on action-type sequences (not raw tool-call text) and seeds its Markov model from 13 hardcoded normal patterns at initialization, requiring no accumulated runtime data. We measured TrajAD on a 40-sequence sequence-level benchmark (20 normal, 20 anomalous) constructed independently of the 60-vector individual-call benchmark. Normal sequences were drawn from common development workflows (e.g., [read, edit, test], [navigate, read, write]); anomalous sequences represented known attack patterns (e.g., mass deletion, download-and-execute, permission escalation bursts, container escape sequences). Results at the default threshold of 0.5: TP = 8/20 (40%), FP = 0/20 (0%); at threshold 0.3: TP = 19/20 (95%), FP = 0/20 (0%).

Post-hoc threshold comparison caveat. Both thresholds (θ=0.5 and θ=0.3) were evaluated on the same 40-sequence test set; the 95% TP figure at θ=0.3 should be validated on a held-out set before deployment.

The 8 sequences flagged at threshold 0.5 correspond exactly to the 8 hardcoded ANOMALOUS_PATTERNS in TrajAD (e.g., [write, write, write, write]: score = 0.755; [delete, delete, delete]: score = 0.740; [network, write, execute]: score = 0.545). The 12 sequences missed at this threshold are interleaved variants—attack patterns not listed in ANOMALOUS_PATTERNS where individual steps appear benign in isolation—confirming that TrajAD's current pattern library covers high-confidence attack archetypes but does not generalize to novel interleaved sequences without additional pattern authoring or runtime Markov learning. The 0% FP rate on normal sequences holds across all tested thresholds, consistent with the normal patterns being seeded with 3$\times$ weight during Markov model initialization. The 60-vector individual-call benchmark does not measure TrajAD's contribution because it tests individual tool calls in isolation; TrajAD's value is detecting multi-step attacks where each individual call is benign but the sequence is anomalous—a property captured only by the sequence-level benchmark above.

D. Layer 4: Safety DSL

A declarative rule interpreter inspired by AgentSpec [3] supports four primitives:

- ALLOW: Whitelist patterns for known-safe operations
- DENY: Blacklist patterns for prohibited operations
- REQUIRE: Mandatory preconditions before tool execution (e.g., require file read before file write to the same path)
- AUDIT: Log without blocking, for observational monitoring

The DSL configuration contains 48 rules covering file system access, shell commands, network operations, and credential handling. Rules are hot-reloadable without proxy restart and self-protecting—the model cannot modify the safety configuration directory. The REQUIRE primitive enables temporal safety policies: for example, requiring that a file be read before it can be overwritten prevents blind writes that could destroy data. The Safety DSL framework, example rule files (48 rules across shell, path, credential, and evasion categories), and a standalone

command-line evaluator are released as open-source artifacts.[1]

E. Layer 5: Argus Spillage Detection

Argus [8] provides three-tier classification spillage detection:

1) Key content: Scans for classification markers, control markings, and sensitive keywords in tool inputs and outputs.
2) File context: Analyzes file metadata, directory structure, and naming patterns for classification indicators.
3) Project relationships: Maps cross-project references to detect when classified content from one project appears in an unclassified context.

Argus achieves 94.86% detection accuracy (5.14% miss rate) with significantly fewer false positives than regex-based pattern matching. The 5.14% miss rate is acknowledged as a residual risk; in federal or classified environments, this rate requires explicit risk acceptance and should be supplemented with manual review procedures. It is integrated into the streaming proxy to scan both tool inputs and tool results.

F. OWASP Agentic AI Top 10 Coverage

Table I maps the 5-layer safety stack to the OWASP Agentic AI Top 10 categories [11]. Several categories receive full or partial coverage; six items (A1, A2, A5, A7, A8, A10) receive partial or no coverage under current evaluation conditions.

A1 (Prompt Injection) receives only partial coverage because the pre-filter addresses injection via tool results but cannot prevent an attacker-controlled prompt in the initial system context from directing the model to issue malicious tool calls—this would require semantic analysis of the full conversation context, which is future work. A2 (Insecure Output Handling) is rated Partial because the action gate evaluates individual tool-call parameters rather than the semantic content of code or data being output; covert-channel outputs embedded within tool results (e.g., encoded data in seemingly benign file writes) are not analyzed. A5 (Uncontrolled Recursion) is partially addressed by rate limiting but the 60-call/minute threshold does not prevent all recursive agent loops; inter-agent orchestration is outside the current scope. A7 (Logging) is functional but lacks real-time alerting and SIEM integration. A8 (Trajectory Manipulation) is rated Partial because TrajAD achieves only 40% TP at the default threshold of 0.5; threshold tuning to 0.3 improves TP to 95% but requires held-out validation before deployment. A10 (Agent Identity) is entirely out of scope: the proxy enforces what actions are taken, not who is authorized to initiate them.

[1]The evaluator accepts a YAML rule file and a JSON test-case file, reporting block rate, false positive rate, and per-rule hit counts without requiring the full proxy infrastructure.

TABLE I
OWASP Agentic AI Top 10 Coverage

ID	Category	Coverage	Layers / Notes
A1	Prompt Injection	Partial	Pre-filter covers tool results; system-level prompt injection not fully addressed
A2	Insecure Output Handling	Partial	Action gate + DSL block unsafe shell/file outputs; however, evaluation operates on individual tool-call parameters, not semantic code content—covert-channel outputs embedded in tool results are not analyzed
A3	Agent Permission Scope Violation	Full	Path restrictions, self-protection, DSL REQUIRE rules
A4	Sensitive Data Exposure	Full	Argus spillage detection (Layer 5)
A5	Uncontrolled Agent Recursion	Partial	Rate limiting (60 calls/min); multi-agent loop detection not implemented
A6	Unsafe Plugin/Tool Use	Full	Action gate 13-category blocklist + DSL DENY rules
A7	Insufficient Logging & Monitoring	Partial	All blocked actions logged; real-time alerting and SIEM integration are future work
A8	Trajectory Manipulation	Partial	TrajAD Markov chain detection; TP=40% at default threshold (θ=0.5). Full coverage requires threshold tuning to θ=0.3 (95% TP; see §IV)
A9	Credential/Secret Leakage	Full	Credential file restrictions + Argus key-content scanning
A10	Unverified Agent Identity	None	Authentication of agent identity is outside scope; network-layer controls only

G. Goal Drift Detection

Beyond per-call safety, the system tracks behavioral drift across a session via four weighted signals: file scope creep (40%), read/write ratio shift (20%), action velocity (20%), and file diversity explosion (20%). Deviations are logged for audit without blocking, providing situational awareness to the developer.

V. Development-Phase Hardening Evaluation

We evaluate the safety stack through a structured hardening log and development-phase benchmark designed to measure both adversarial block rate and false positive rate across the system's evolution.

A. Development-Phase Hardening Log

Important scope caveat. The 60 adversarial vectors reported in this section constitute a development-phase hardening log, not an independent security benchmark. The vectors were constructed by the same developer who designed and patched the system, iteratively, as weaknesses were discovered and fixed. This process is analogous to unit tests written alongside the code: it documents what was hardened, not what an independent adversary would find. Specifically:

- The 100% block rate on these 60 vectors means "the system blocks the 60 patterns it was specifically designed and hardened to block." It does not mean "no additional attack patterns exist."
- The progression from 54% to 98% to 100% is a retrospective measurement of the system at three historical snapshots, not a controlled experiment with fixed stimuli applied to independent system versions.
- An independent red-team assessment using vectors unknown to the developer would constitute a genuine security benchmark; that evaluation is future work.

Readers should interpret the 100% recall figure as evidence of implementation completeness against a known threat model, not as evidence of security against adaptive adversaries.

B. Benchmark Design

The benchmark comprises 60 adversarial vectors organized by severity:

TABLE II
Red-Team Benchmark Composition

Severity	Count	Example Categories
Critical	3	Fail-open bypass, shell metachar evasion, path traversal
High	7	Code path bypass, encoding tricks, credential exfiltration
Medium	10	Rate limit evasion, self-modification, indirect references
Low	7	Verbose logging bypass, non-standard file extensions
Total	27	(vulnerability classes)

The 27 vulnerability classes expand to 60 test vectors when accounting for variant attack strategies per class (e.g., multiple encoding schemes for shell metacharacter evasion, multiple path traversal techniques). Each vector is a concrete tool call or sequence of tool calls designed to bypass one or more safety layers.

C. Progression Results

The system was evaluated at three stages of development. As noted in §V-A, these stages represent retrospective snapshots of the system as it was hardened—not independent evaluations of fixed system versions against fixed stimuli. The FP rate is measured against 12 legitimate development operations in the evaluation set; broader deployment may surface additional legitimate operations that trigger rules.

TABLE III
Safety Stack Block Rate Progression (development log; 95% CI via Clopper-Pearson where measured)

Stage	Block Rate (95% CI)	FP Rate	Layers Active
Initial deployment	∼54% (reconstructed)*	3.2%	Action gate (v1
Post-hardening	98% (89.4%–99.9%)	0.8%	Action gate (v2) +
Full stack	100% (94.0%–100%)	0%	All 5 layers

* Stage 1 block rate is a retrospective estimate derived from the 60-vector set as reconstructed against the initial system state; it was not a direct measurement on a preserved build. No confidence interval is reported because the estimate does not meet the assumptions for Clopper-Pearson exact binomial inference (the 60-vector test set was not yet complete at Stage 1). See §V-A.

The 95% confidence intervals use the Clopper-Pearson exact method (conservative, appropriate for small-n binomial proportions). For the 100% (60/60) result, the lower bound is 94.0%—meaning that at 95% confidence, the true block rate on this vector distribution is at least 94.0%. The 0% false positive rate is measured over 12 legitimate operations; with $n = 12$, the 95% CI is [0%, 24.2%], reflecting the limited denominator.

Stage 1: Initial deployment (54% block rate). The v1 action gate used a fail-open design and covered only 6 shell command categories. Critical bypasses included: shell metacharacter evasion via pipe chains and encoding (base64, hex), path traversal via .. sequences without canonicalization, and two code paths that bypassed the gate entirely (non-streaming and streaming translated tool calls).

Stage 2: Post-hardening (98% block rate). The action gate was redesigned as fail-closed, shell blocklists were expanded to 13 categories (adding encoding-based evasion, Unicode normalization, and metacharacter variants), os.path.realpath canonicalization was added for all path operations, and both bypass code paths were fixed with mandatory gate evaluation. The Safety DSL added temporal policies (REQUIRE) and self-protection rules. The remaining 2% were evasion patterns using indirect references and encoding tricks that bypassed regex-based detection.

Stage 3: Full stack (100% block rate). The Safety DSL's additional DENY rules and REQUIRE preconditions closed the remaining 2 indirect-reference vectors that escaped the action gate's regex patterns. Argus detected classification spillage attempts in the final vector class. TS-Guard and TrajAD were integrated into the stack during this phase; however, as discussed in §IV, TS-Guard had not yet accumulated sufficient training data to produce a trained classifier, and TrajAD's sequence-level detection was not exercised by the individual-call

benchmark. The combined stack achieved 100% block rate on the 60 development-phase vectors, with 0% false positives on the 12 legitimate development operations in the evaluation set.

D. Per-Layer Contribution

TABLE IV
Safety Stack Per-Layer Performance (60-vector development set)

Layer	Block Rate	FP Rate
Action Gate (v2)	100% known patterns[a]	0%
TS-Guard	Not measurable[b]	N/A
TrajAD	40% TP (seq-level, θ=0.5)[c]	0% FP
Safety DSL (48 rules)	100% rule-covered patterns[a]	0%
Argus spillage	94.86% (5.14% miss rate)	Sig. fewer than regex
Combined stack	100% on 60-vector dev set	0% (on 12 legit ops)

[a] "Known patterns" means patterns the rule was specifically designed to block; this is not a claim of coverage against unknown attack variants.
[b] TS-Guard requires accumulated blocked-actions.jsonl data. At evaluation time (7 days of deployment), no blocked-action records had been written, leaving TS-Guard untrained (defaults to allow=True). See §IV for full discussion.
[c] TrajAD evaluated on a 40-sequence benchmark (20 normal, 20 anomalous). At θ=0.5: TP = 8/20 (40%), 95% CI [19.1%, 63.9%] (Clopper-Pearson), FP = 0/20. At θ=0.3: TP = 19/20 (95%), FP = 0/20. Both thresholds evaluated on the same set; see post-hoc caveat in §IV.

Each layer independently contributes to the combined result. The action gate handles known-pattern matching. TS-Guard is designed to catch evasion patterns that escape regex rules, once sufficient blocked-action data accumulates to train its classifier. TrajAD identifies anomalous behavioral sequences that individual-call analysis misses, with independent sequence-level benchmark results reported above. The Safety DSL enforces temporal and structural policies. Argus detects classification-level data leakage.

E. Per-Layer Ablation

To measure the contribution of individual rule categories within the action gate, we performed a leave-one-out ablation over the 60-vector benchmark, removing one rule class at a time and measuring Recall, F1, and missed attack count.

Key finding: shell command rules are the critical layer. Removing shell command rules drops recall to 25% (36 of 60 attacks pass through), far exceeding the impact of any other single rule class. This asymmetry reflects the benchmark composition: the majority of adversarial vectors exploit shell execution paths (destructive commands, exfiltration, encoding-based evasion). Path rules and credential rules each account for 12–15 additional escapes when removed; evasion rules cover 4 additional vectors at the margins.

TABLE V
Action Gate Per-Layer Ablation (60-vector development set; 95% CI via Clopper-Pearson)

Configuration	Recall (95% CI)	F1	Missed
Full stack (all layers)	100% [94.0%, 100%]	1.0000	0
Without shell commands	25% [14.7%, 37.9%]	0.40	36
Without path rules	75% [62.1%, 85.3%]	0.86	12
Without credential rules	69% [55.7%, 80.1%]†	0.81	15
Without action gate	62.5% [50.0%, 74.1%]†	0.77	18
Without evasion rules	92% [82.7%, 97.1%]†	0.96	4

† CIs computed at nearest integer numerator out of n=60 (e.g., 69% ≈ 41/60, 62.5% ≈ 38/60, 92% ≈ 55/60).

F. Adversarial Evasion Testing

Separately from the 60-vector direct attack benchmark, we constructed a 25-vector evasion test suite targeting the action gate's preprocessing pipeline specifically. These two test populations are distinct and reported separately: the 60-vector set tests whether defined patterns are blocked; the 25-vector set tests whether the preprocessing pipeline correctly normalizes obfuscated inputs before matching.

Evasion vector results (25 vectors, 10 technique families):

- Before preprocessing: 40% detection (10/25); 95% CI [21.1%, 61.3%].
- After Unicode normalization + zero-width stripping + path normalization + shell quote stripping: 88% detection (22/25); 95% CI [68.8%, 97.5%].
- After homoglyph map expansion: 88% (22/25)—same rate; homoglyph expansion did not close the remaining gaps.

Circularity note. The 88% post-preprocessing detection rate measures coverage of the preprocessing pipeline against known evasion techniques that the pipeline was specifically designed to handle. This is analogous to the 60-vector benchmark measuring coverage of known attack patterns: both numbers reflect completeness against a defined threat model, not generalization to novel evasion techniques outside that model. An adversary who constructs evasion methods not represented in the 25-vector set would face a preprocessing pipeline not designed for those inputs.

Remaining 3 gaps: variable indirection (1 vector) and indirect multi-step injection (2 vectors) were not caught by static preprocessing. These represent two-step attacks where each individual step is benign in isolation—the first step sets up an environment variable or indirect reference, and the second step exploits it. Static regex operates on individual tool calls and cannot detect this pattern.

Motivation for learned layers. The indirect injection gap directly motivates the TS-Guard and TrajAD components. TS-Guard's learned classifier can correlate across a session's blocked-action history; TrajAD's Markov chain trajectory model detects anomalous sequences even when

individual calls appear benign. Neither is achievable with static regex alone.

The 60-vector direct attack set and the 25-vector evasion set are not combined in reporting. They test different properties of the system and have different denominators; combining them would obscure the distinct characteristics of each evaluation.

G. Limitations of Static Rule Evaluation

The action gate and Safety DSL are static rule systems. A sophisticated adversary who can probe the proxy—observing which tool calls are blocked and which are allowed—can systematically enumerate the rule space and construct inputs that fall outside defined patterns. This adaptive adversary scenario is a fundamental limitation of any rule-based system.

Specifically: (1) an attacker with repeated access can perform a binary search over the blocked-pattern space, learning the exact boundaries of each rule; (2) novel encoding schemes not present in the training distribution of TS-Guard will not be caught until sufficient examples accumulate for retraining; (3) multi-step attacks that keep each individual step below the anomaly threshold of TrajAD can avoid trajectory-based detection; and (4) the 48 DSL rules were authored by a single developer and reflect one individual's threat model, which may miss entire categories of attack that an adversary would explore.

Mitigations in the current design include: rule hot-reloading (allowing rapid response to newly discovered patterns), TS-Guard nightly retraining (incorporating new blocked examples), and AUDIT-mode logging (providing visibility into near-miss events). However, these mitigations reduce but do not eliminate the adaptive adversary risk.

Beyond these, three additional countermeasures address adversary probing specifically. First, canary and honeypot tool calls: the proxy can be configured to insert synthetic tool-call entries (e.g., a fictitious credential file path, a non-existent sensitive directory) that no legitimate workflow would ever access; any access to a canary entry constitutes a high-confidence probe detection signal. Second, probe rate-limiting: repeated near-miss events (tool calls that are blocked but do not match any single rule precisely, suggesting boundary probing) can trigger an escalating rate limit that degrades the attacker's ability to enumerate the rule space within a session. Third, alert-on-near-miss escalation: AUDIT-mode events that cluster near a known rule boundary—indicating systematic probing of rule edges—can be elevated from log entries to real-time alerts, enabling rapid response before the adversary constructs a successful bypass. These countermeasures are not yet implemented and are presented as a concrete roadmap item. Formal adversarial robustness analysis of the rule set—analogous to adversarial example analysis in ML—is an important direction for future work.

H. Latency Analysis

TABLE VI
Safety Stack Latency Overhead

Event Type	Added Latency
Text tokens (passthrough)	0 ms
Tool calls (buffered + evaluated)	<1 ms
Protocol translation (XML → JSON)	<1 ms (included above)

The selective-buffer architecture achieves zero latency overhead on text tokens—the dominant event type during streaming—while adding sub-millisecond evaluation time to tool calls. Protocol translation operates within the same buffering pipeline, adding no additional latency.

I. Comparison with Prior Work

TABLE VII
Comparison with Agent Safety Systems

System	Layer	Latency	FP Rate (95% CI)
AEGIS [1]	SDK hooks	8.3 ms	1.2% (CI not reported)[d]
PCAS [2]	Compile-time	0 ms[a]	N/R
LlamaFirewall [4]	Post-gen	N/A[b]	N/R
Ours	SSE stream	<1 ms	0% [0%, 24.2%][c]

[a] Policies compiled at build time; no runtime overhead but no runtime adaptation. [b] Post-generation analysis; latency not applicable as evaluation occurs after execution. [c] Measured over 12 legitimate operations; 95% CI [0%, 24.2%] (Clopper-Pearson). [d] AEGIS reports 1.2% FP rate without a confidence interval or sample size; the denominator required to compute a CI is not available in the published report. Note: FP rate comparison between this system and AEGIS is not statistically significant at current sample sizes. With n=12 for our system, the 95% CI spans 0–24.2%; a meaningful FP comparison requires a substantially larger legitimate-operation sample. See §VI for discussion. N/R = not reported.

VI. Discussion

Selective buffering is the key insight. The distinction between read-only (text) and action-bearing (tool) SSE events enables safety enforcement during streaming without degrading the developer experience. Prior systems either hook into SDKs after streaming is handled, or scan completed responses. Operating at the raw SSE layer enables safety enforcement during streaming with zero text latency. The fail-closed design ensures that even parser failures result in blocked (not allowed) tool calls.

Defense-in-depth prevents single points of failure. No individual layer achieves 100% block rate alone. The action gate misses encoding-based evasion; TS-Guard misses novel patterns outside its training distribution; TrajAD

flags anomalies but cannot determine intent; the DSL depends on rule completeness; Argus has a 5.14% miss rate and operates at the classification level rather than individual call level. Together, they cover each other's blind spots.

The hardening progression validates iterative development. The jump from 54% to 98% was achieved primarily through architectural changes (fail-closed design, path canonicalization, code path fixes). The final 2% was closed by expanded DSL rules (REQUIRE/DENY temporal policies for indirect-reference vectors) and Argus spillage detection. TS-Guard and TrajAD are integrated as defense-in-depth layers but their contributions to this specific benchmark are not isolable: TS-Guard was untrained at evaluation time, and TrajAD's sequence-level detection is not measured by the individual-call benchmark. The progression suggests that rule-based approaches reach a ceiling around 95–98%, and that learned and behavioral layers require dedicated evaluation methodologies (held-out blocked-action sets; sequence-level benchmarks) not provided by a single-call hardening log.

TrajAD deployment recommendation. We recommend deploying TrajAD at threshold θ=0.3 rather than the default θ=0.5, accepting the increased computational cost for the 55 percentage point TP improvement (40% to 95%) at zero additional FP cost observed in the development-phase evaluation. This recommendation is contingent on the caveat in §IV: the 95% TP figure at θ=0.3 was evaluated on the same test set used to select the threshold and should be validated on a held-out set before production deployment.

FP rate comparison is statistically underpowered. With n=12 legitimate operations in the evaluation set, the 95% CI for our false positive rate is [0%, 24.2%] (Clopper-Pearson). This renders any comparison with AEGIS's 1.2% FP rate statistically unsupported at current sample sizes: the confidence intervals overlap substantially, and no conclusion about relative FP performance can be drawn. A meaningful FP comparison would require substantially larger legitimate-operation samples from both systems under comparable workload conditions.

Protocol translation as a safety opportunity. The proxy's dual role—safety enforcement and protocol translation—is synergistic. Tool calls must already be buffered for XML-to-JSON translation; safety evaluation adds negligible marginal cost to an already-buffered event. Systems that separate protocol handling from safety enforcement miss this opportunity.

A. Threats to Validity

The following threats to internal and external validity apply to results in this paper.

Internal validity. The 60-vector benchmark was designed by the same developer who built and iteratively patched the system. Each patch was followed by adding a test vector to cover the newly closed gap. This co-evolution of system and test suite means the 100% recall figure measures coverage of known gaps, not all possible gaps. The before/after block rate measurements are retrospective: the same 60 vectors were applied to mental reconstructions of earlier system states, not to preserved, reproducible builds. Measurement error in the retrospective stages (especially Stage 1 at 54%) cannot be ruled out.

External validity. The system was evaluated in a single-developer workflow on one codebase over a defined development period. The 0% false positive rate applies to 12 legitimate development operations—a small sample that may not represent the diversity of operations in team deployments, unfamiliar codebases, or non-standard toolchains. The threat model reflects one developer's attack enumeration; adversarial actors with different knowledge and capabilities may find vectors not represented in the benchmark.

Construct validity. Block rate is a proxy for security. A system that blocks 100% of known patterns while allowing novel unknown patterns provides a false sense of security. The 5.14% Argus miss rate represents a real residual risk, as does the static rule system's vulnerability to adaptive adversaries (§V-G). The latency measurements (<1 ms) were taken on a single hardware configuration under low load; performance under sustained load or adversarial DoS conditions has not been measured.

Limitations. (1) The 60-vector benchmark was designed by the same developer who built and patched the system, creating a self-evaluation bias. An independent red-team assessment would carry significantly more weight. (2) TS-Guard was not trainable during this evaluation due to zero accumulated blocked-action records; the fail-closed action gate blocks adversarial calls before blocked-actions.jsonl is populated during normal operation. A separate logging path that records actions blocked during deliberate red-team sessions (not live operation) is needed to bootstrap TS-Guard training data. (3) TrajAD's sequence-level benchmark is independent of the 60-vector individual-call benchmark; TrajAD TP = 40% at default threshold covers only the 8 hardcoded ANOMALOUS_PATTERNS and does not generalize to interleaved variants without pattern library expansion. (4) The 0% false positive rate reflects a single developer's workflow over 12 benchmark operations; broader deployment may surface legitimate operations that trigger rules. (5) The 48-rule DSL requires manual authoring and maintenance; automated rule generation from blocked-action patterns is future work. (6) Argus's 5.14% miss rate requires risk acceptance in federal contexts and should be supplemented with manual review. (7) Formal verification of the fail-closed guarantee and expert mode access controls is future work.

VII. Conclusion

We presented a selective-buffer streaming proxy and 5-layer safety stack for AI coding agents. The proxy operates at the raw SSE stream layer—to our knowledge, the first published system at this architectural layer—forwarding text tokens at zero latency while buffering tool calls for deterministic safety evaluation. SSE-layer interception is strictly earlier in the data path than SDK hooks, which operate after parsing and deserialization are complete. The open-source Safety DSL (48 rules, YAML-based, hot-reloadable) provides a reusable declarative safety framework independent of the proxy infrastructure.

Evaluation through a development-phase hardening log of 60 adversarial vectors shows progression from 54% block rate at initial deployment to 100% after full stack integration, with 0% false positives on 12 legitimate development operations (95% CI [0%, 24.2%]). The 100% recall result carries a 95% CI lower bound of 94.0% (Clopper-Pearson) and reflects coverage of patterns the system was specifically designed to block; it should not be interpreted as a claim of security against novel or adaptive adversaries. The 5-layer defense-in-depth architecture—combining deterministic rules, learned classifiers, behavioral models, declarative policies, and domain-specific detectors—is designed to cover each layer's blind spots, but formal adversarial robustness analysis and independent red-team evaluation remain important future work.

The selective-buffer architecture generalizes beyond our specific deployment. Any system that processes SSE streams containing both informational and action-bearing events can benefit from type-aware buffering: streaming the safe events immediately while holding the dangerous ones for evaluation.

References

[1] A. Kamboj et al., "AEGIS: No Tool Call Left Unchecked — Pre-Execution Firewall and Audit Layer for AI Agents," arXiv:2603.12621, 2026.

[2] "PCAS: Policy Compiler for Secure Agentic Systems," arXiv:2602.16708, 2026.

[3] "AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents," arXiv:2503.18666, 2025.

[4] Meta, "LlamaFirewall: An Open Source Guardrail System for Building Secure AI Agents," arXiv:2505.03574, 2025.

[5] "ILION: Deterministic Pre-Execution Safety Gates for Agentic AI Systems," arXiv:2603.13247, 2026.

[6] Y. Mou et al., "ToolSafe: Enhancing Tool Invocation Safety of LLM-based Agents via Proactive Step-level Guardrail and Feedback," arXiv:2601.10156, 2026.

[7] "TrajAD: Semantic Trajectory Anomaly Detection for AI Agents," arXiv:2602.06443, 2026.

[8] "Argus: Classification-Aware Spillage Detection for LLM Systems," arXiv:2512.08326, 2025.

[9] "Mind the GAP: Text-Action Divergence in LLM Tool Use," arXiv:2602.16943, 2026.

[10] "The 2025 AI Agent Index," arXiv:2602.17753, 2026.

[11] OWASP, "OWASP Top 10 for LLM Applications and Generative AI — Agentic AI Security," 2025. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/