AI Agents in CI/CD: Security, Robustness, and Productivity

A Deep Dive from Theory to Production

Andre Lustosa, PhD

Principal Software Engineer | Red Hat | AIPCC Ecosystems

Red Hat

Lecture Outline

I. Foundations	Agent architectures, delegation, and the confused deputy
II. Security Theory	Injection taxonomy, trust boundaries, containment primitives
III. Robustness	Non-determinism, convergence, cycle detection, action-space reduction
IV. Measurement	Productivity quantification, automation bias, human-in-the-loop control
V. Open Problems	Formal verification, alignment in CI, research frontiers

Agent Architecture: From Theory to Tool-Use LLMs

Agent Model

- Sense: read files, API responses, CI logs
- Reason: LLM inference (probabilistic)
- Act: write files, run commands, call APIs
- Loop: observe outcome, adjust, repeat

Key distinction from classical AI agents: the reasoning engine is a neural network with no formal guarantees on output correctness.

Tool-Use LLM as UTM Analog

- LLM + tools = Turing-complete system
- Tape: filesystem, git repos, APIs
- Head: tool-use function calls
- Control: learned policy (weights), not program
- Halting: not guaranteed (token limits as proxy)

Consequence: you cannot statically determine what an agent will do. All safety must be enforced at the environment boundary.

The “Principal-Agent” Problem in AI Automation

- Economics: principal delegates to agent with misaligned incentives
- In AI CI/CD: the organization delegates code changes to an LLM agent
- Information asymmetry: agent 'sees' codebase details principal doesn't verify
- Moral hazard: agent may take shortcuts invisible at review time
- Adverse selection: which tasks are suitable for delegation?

The classical solution is monitoring + incentives. For LLM agents, monitoring = code review + gates + telemetry. 'Incentives' = prompt engineering + structured output constraints. Neither is complete.

The Confused Deputy Revisited

Classic Confused Deputy (1988)

- Program A has authority to write billing file
- User B tricks A into writing user-controlled data
- A acts on B's behalf using A's privileges
- Root cause: ambient authority not scoped to intent

LLM Agent as Confused Deputy

- Agent has git push + API credentials (authority)
- Attacker embeds instructions in a bug ticket (trick)
- Agent executes attacker's intent with org's credentials
- Root cause: identical. Ambient authority + untrusted input

Mitigation: Capability-Based Security

Replace ambient authority with explicit capabilities. The agent receives only the permissions it needs for the specific task, scoped by ticket, repo, and time window. This is the theoretical basis for our gate architecture.

Injection Attack Taxonomy: Why Prompt Injection is Different

Attack Class	Mechanism	Defense	Why It Works
SQL Injection	Untrusted data interpreted as SQL	Parameterized queries	Grammar-based separation
XSS	Untrusted data interpreted as script	Output encoding / CSP	Context-aware escaping
Command Injection	Untrusted data interpreted as shell	Avoid shell; use execve	Argument isolation
Prompt Injection	Untrusted data interpreted as instruction	**???**	**No grammar to parse**

Every prior injection class was solved by separating data from code at a syntactic level. Prompt injection cannot be solved this way because natural language has no formal grammar that distinguishes instruction from data.

Trust Boundary Analysis

Formal decomposition: Who provides input? What authority does the agent hold? When is output trusted?

WHO (Identity Layer)

- Trigger author (changelog-verified)
- Comment authors (email-domain filtered)
- External reporters (quarantined)
- The LLM itself (not a trusted source)

WHAT (Authority Layer)

- Git push to specific branches
- MR/PR creation and update
- Issue tracker mutations
- Network egress (sandboxed)

WHEN (Temporal Layer)

- Pre-agent: input filtering
- During: runtime containment
- Post-agent: output validation
- Post-merge: monitoring

Case Study: Defense in Depth in Production

Case Study: Red Hat Agentic CI gate architecture (production since 2025)

PRE-AGENT

- Label author: @redhat.com via Jira changelog API
- External reporter gate: quarantine + human review
- Comment filter: only @redhat.com in prompt
- Embargo: JQL excludes EMBARGOED tickets
- AI sensitivity: semantic security-bug detection

POST-AGENT

- Sensitive files: blocks .env, .pem, .key commits
- Secret scan: gitleaks on all commits pre-push
- Commit identity: author matches expected bot
- Visibility: comments restricted to employees

e.g: https://opendatahub-io.github.io/agentic-ci/api/gates/

Containment Primitives: Kernel-Level Enforcement

Landlock (Linux 5.13+)

Filesystem access control via LSM. Process declares which paths it can read/write. Inherited by children. No root required.

Restrict agent to workspace directory + read-only deps

seccomp-bpf

System call filtering via BPF programs. Blocks dangerous syscalls (ptrace, mount, kexec). Granular per-process policy.

Prevent container escape and privilege escalation

Network Namespaces + nftables

Per-sandbox network stack with firewall rules. Allowlist-only egress. No ambient network access.

Block exfiltration to attacker-controlled endpoints

These are kernel-level enforcement mechanisms. The agent cannot bypass them regardless of prompt injection success.

Case Study: Sandboxing in Production

Case Study: OpenShell sandbox in Red Hat Agentic CI

- Embedded gateway starts per CI job, no external infrastructure
- Landlock: agent writes only to /workspace, reads only approved paths
- Network: allowlist of endpoints (configurable per-repo via .agentic-ci/openshell-policy.yml)
- Default allowlist: GitHub, GitLab, PyPI, Vertex AI, Anthropic API
- Credentials mounted read-only, never exposed as environment variables in the sandbox
- All other outbound traffic silently dropped (not rejected, to avoid side-channel leaks)

Design principle: the sandbox is the security boundary, not the prompt. If you rely on prompt instructions for safety, you've already lost.

The Oracle Problem: Limits of Output Verification

- Can you formally verify that LLM output is 'safe'?
- Rice's theorem: any non-trivial semantic property of programs is undecidable
- The generated code IS a program. Verifying its safety is at least as hard as the halting problem.
- Practical implication: you cannot build a gate that provably catches all malicious output
- What you CAN do: reduce the attack surface until the residual risk is manageable

The Defense Stack (ordered by enforceability)

- 1. Kernel enforcement (Landlock, seccomp, netfilter) - cannot be bypassed by the agent
- 2. Structural validation (gitleaks, file-path checks) - syntactic, decidable
- 3. Semantic analysis (AI-powered review) - probabilistic, best-effort
- 4. Human review - highest quality, lowest throughput

Non-Determinism in Agentic Systems

Sources of Non-Determinism

- Sampling temperature (even at T=0, not deterministic)
- Context window position effects
- Batching and quantization artifacts
- Tool output variance (git diff timing, API responses)
- Prompt sensitivity to minor wording changes

Consequences for CI/CD

- Same bug + same prompt = different patches
- Retry may produce better OR worse results
- Test suite pass is necessary but not sufficient
- Idempotency cannot be assumed
- Statistical reliability, not deterministic correctness

Mitigation Strategies

- Structured output schemas: constrain output space to valid shapes
- Verdict-based skill design: agent must declare intent before acting
- Idempotent gate design: gates are safe to re-run on retry
- Monotonic state machines: workflow state can only move forward, never back

Convergence and Divergence in Feedback Loops

When does a closed-loop agentic system converge to a fixed point?

Convergent Patterns

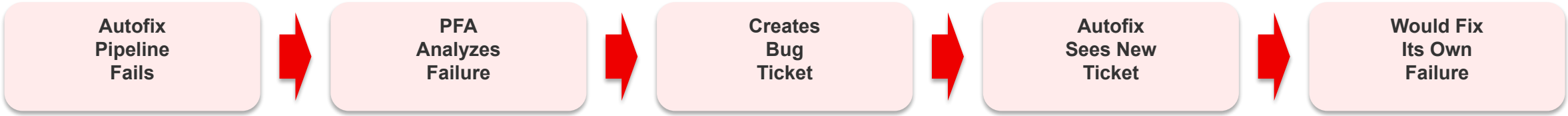
- Bug fix iteration: review feedback narrows the solution
- Bounded retry with monotonic state
- Human checkpoint breaks infinite loops
- CI pass/fail provides a decidable termination criterion
- Diminishing error surface per iteration

Divergent Patterns

- Self-healing loops without cycle detection
- Agent 'fixes' its own fixes (oscillation)
- Expanding scope: agent adds features while fixing bugs
- Cost explosion: each retry consumes tokens
- Cascading failures across dependent pipelines

Case Study: Cycle Detection in Self-Healing CI

Case Study: Pipeline Failure Analyzer-Autofix cycle prevention at Red Hat



Solution: Label-Based Cycle Prevention

- PFA-created tickets receive no-autofix label at creation time
- Autofix's query excludes tickets with no-autofix label
- Cycle is broken at the state machine level, not by detection heuristics
- Formal property: the label graph is a DAG, not a cycle
- Same pattern applies to any self-referential automation chain

Action Space Reduction via Skills

Skills as structured constraints that reduce the agent's effective action space

Unconstrained Agent

- Action space: all possible tool call sequences
- High variance in output quality
- Harder to review (anything could happen)
- Security surface: entire tool set

Skill-Constrained Agent

- Action space: task-specific instruction set
- Structured verdict: declare intent before acting
- Reviewable: expected behavior is documented
- Security surface: scoped to task requirements

Analogy: type systems for agents

Skills function like type signatures: they constrain the space of valid behaviors without dictating implementation. A/B testing of skill variants (our production approach) is analogous to benchmarking type system designs for ergonomics and correctness tradeoffs.

Measuring AI Agent Productivity

Naive Metrics (misleading)

- MRs merged per week (quantity, not quality)
- Lines of code changed (Goodhart's law)
- Time to first MR (ignores review cost)
- Bug close rate (includes false closures)

Better Metrics (still imperfect)

- Review acceptance rate (% merged without revision)
- Engineer time displaced (hours saved per task)
- Defect escape rate (bugs introduced by agent)
- Cost per merged MR (tokens + review time)

The Measurement Paradox

If agents handle the easy bugs, engineers handle the hard ones. Average bug resolution time may INCREASE because the easy bugs no longer pull down the average. Productivity improvements are real but invisible in aggregate metrics. You need cohort analysis: compare similar-difficulty tasks with and without agent assistance.

Automation Bias and Over-Reliance

- Automation bias: tendency to favor suggestions from automated systems
- Particularly dangerous in code review: AI says it's fine, reviewer agrees faster
- Complacency effect increases with perceived agent reliability
- The 'LGTM stamp' risk: human review degrades when agent review exists
- Ironically, better agents may produce worse human review quality

Countermeasures

- Agent never self-merges: humans must take an explicit action
- AI review complements, does not replace, human review assignment
- Chill mode: suppress low-severity findings to prevent review fatigue
- Defect injection testing: periodically verify human reviewers catch planted bugs

Human-in-the-Loop as Supervisory Control

Control Theory Mapping

- Plant: the codebase + CI pipeline
- Controller: AI agent (non-linear, stochastic)
- Sensor: test suites, linters, gate output
- Actuator: git push, MR creation
- Supervisor: human reviewer (override authority)
- Reference signal: correct, secure, passing code

Why Human Supervision Matters

- The controller is stochastic: identical inputs may produce different outputs
- No Lyapunov function exists for LLM reasoning (stability is not provable)
- Test suites are incomplete sensors (cannot observe all state)
- Human acts as a bounded-rationality supervisor with override authority
- The merge button is a hard gate, not a suggestion

Production Evidence at Scale

Case Study: Red Hat Agentic Ecosystems production data (2025-2026)

100+	7	15+	5
MRs merged per week	Production workflows	Maintained projects	Engineers on the squad

Workflows: Autofix (triage + fix), Code Review, Knowledge Sync, Package Onboarding (CPU/CUDA/ROCm/Gaudi/TPU/Neuron/Spyre), Pipeline Failure Analyzer, RFE Assessor, Security Alerts. Zero security incidents from agent-generated code to date.

Open Research Problems

Formal Verification of Agent Behavior

Can we develop tractable verification for bounded agent traces? Partial verification of finite tool-call sequences may be feasible even if general verification is not.

Prompt Injection Defenses

No complete defense exists. Research directions: instruction hierarchy enforcement, data tainting through attention layers, formal separation of instruction and data channels in transformer architectures.

Alignment in CI/CD

Agent 'values' are shaped by training data and prompts. How do you align agent behavior with organizational security policies when those policies can't be fully specified in natural language?

Optimal Human-Agent Task Allocation

Which tasks should be delegated and which kept human? Current allocation is heuristic. We lack formal frameworks for delegation decisions under uncertainty.

Key Takeaways

- AI agents are confused deputies: ambient authority + untrusted input
- Prompt injection is fundamentally different from prior injection classes
- Kernel-level containment is the only trustworthy security boundary
- Non-determinism requires statistical thinking, not deterministic proofs
- Human-in-the-loop is supervisory control, not a crutch
- The defense stack must be ordered by enforceability, not convenience
- Measure displaced effort, not output volume

Questions?

Andre Lustosa, PhD | alustosa@redhat.com
https://www.redhat.com/en/products/ai


linkedin.com/company/red-hat


youtube.com/@redhat


facebook.com/redhat


x.com/RedHat

Red Hat