Constitutional Self-Modification: A 7-Layer Safety Framework for Autonomous Code-Generating Agents

Vasyl Golubenko
TOV ZELTREX, Kyiv, Ukraine
ceo@zeltrex.com | https://zeltrex.com

March 2026

Abstract

Autonomous AI coding agents that can merge their own code into production repositories introduce a novel safety challenge: how to permit beneficial self-modification while preventing self-corruption. We present a 7-layer defense-in-depth safety framework, validated on 280+ autonomous tasks over 10 days of continuous operation with zero safety violations. The framework enforces 19 constitutional rules across input validation, path constraints, quality gates, integration guards, merge analysis, deployment rollback, and operational circuit breakers. We ground the design in a systematic audit of 23 modules (523 tests) that revealed 8 critical "dead integration" gaps — features fully coded but never activated due to wiring bugs. Analysis of 33,000 agent-authored pull requests shows CI pipeline pass rate as the strongest predictor of merge safety. Our 7-layer architecture addresses the four RSI risks identified by the ICLR 2026 Workshop (specification hacking, memory drift, brittle self-edits, unbounded exploration) through evaluation function immutability, anti-truncation guards, constitutional path blocking, and budget-bounded operation. We compare against 6 existing safety frameworks and demonstrate that production deployment reveals failure modes invisible to simulation-only evaluation.

Keywords: constitutional AI, safe self-modification, defense in depth, autonomous agents, CI/CD safety, recursive self-improvement

1 Introduction

The emergence of autonomous AI coding agents — from SWE-agent [20] to Devin [7] to Claude Code [8] — has rapidly progressed from benchmark evaluation to production deployment. Night Shift [17] demonstrated sustained autonomous operation over 10+ days, completing 280+ tasks with evolutionary parameter optimization. GODEGEN [16] introduced cognitive modules (knowledge graphs, skill libraries, personas) that enable the agent to accumulate capabilities across sessions. The temporal dimension gap analysis [18] showed that such systems diverge fundamentally from stateless tool agents.

A critical capability at this frontier is **autonomous code merging**: the agent generates code, runs tests, and merges approved changes into the production repository without human intervention. This capability transforms the agent from a suggestion engine into a self-modifying system — one that can alter its own operational environment through the code it produces.

This self-modification capability introduces risks that existing safety frameworks do not adequately address. Classical AI safety research focuses on alignment of model outputs [9]. Agentic AI security surveys [11] catalog threats (prompt injection, tool misuse, memory poisoning) but treat the agent as operating on external code, not its own. The ICLR 2026 Workshop on Recursive Self-Improvement [23] identified four core risks — specification hacking, memory drift, brittle self-edits, and unbounded exploration — but noted that safety mechanisms remain "underaddressed in existing RSI literature."

The gap is empirically validated. An analysis of 33,000 agent-authored pull requests [5] found that failed PRs involve larger diffs, more files touched, and higher CI/CD pipeline failure rates. A study of AI agents modifying CI/CD configurations [4] found Copilot-authored CI changes merge 15.6 percentage points more often than general code changes, suggesting differential trust levels are warranted. Yet no deployed system has published a comprehensive safety architecture for autonomous code merging.

We address this gap with three contributions:

1. A **7-layer defense-in-depth framework** (Table 1) that enforces 19 constitutional rules from input validation through operational monitoring, validated on 10 days of production deployment with zero safety violations.

2. A **systematic "dead integration" audit methodology** that revealed 8 critical gaps in a 23-module system — features fully coded but never activated — demonstrating that safety code must be verified as *wired*, not merely *written*.

3. An **empirical analysis** comparing our framework against 6 existing safety architectures, showing that production deployment reveals failure modes invisible to simulation.

2 Related Work

2.1 Agent Safety Frameworks

AGENTSAFE [3] operationalizes the AI Risk Repository into design-time, runtime, and audit-time controls for the full agentic loop (plan, act, observe, reflect). It introduces semantic telemetry and dynamic authorization but evaluates primarily through synthetic scenarios rather than production deployment.

AGrail [21] uses two cooperative LLMs in a test-time adaptation loop to iteratively refine safety checks per action type, achieving 0% attack success rate on the Safe-OS benchmark. Its memory module enables lifelong safety learning. However, AGrail operates at the action level rather than the code-merge level, leaving CI/CD safety unaddressed.

The **NVIDIA safety framework** [15] treats safety as an emergent property of dynamic model-orchestrator-tool-data interactions, releasing 10,000+ attack and defense execution traces. This is closest to our approach in treating safety as systemic rather than model-level, but focuses on general-purpose agents rather than self-modifying code generators.

Systems Security Foundations [12] applies classical security principles (attacker modeling, TCB minimization, defense in depth) to agentic AI, introducing the "Probabilistic TCB" problem

— the model itself is the trusted computing base but is fundamentally non-deterministic. We adopt their defense-in-depth principle as our core architectural pattern.

2.2 Constitutional Approaches

Marri [22] proposes Constitutional Spec-Driven Development, embedding versioned, machine-readable security constraints into AI coding pipelines. The key finding — 26.1% of production AI agent skills contain exploitable vulnerabilities — motivates our path-blocking approach. However, their constitution operates at specification time, not runtime enforcement.

The Three Laws of Self-Evolving Agents [14] (Endure/Safety, Excel/Performance, Evolve/Autonomy) provide a philosophical framework. We operationalize these laws into enforceable layers: Endure maps to our L5-L7 (merge, deploy, operations), Excel to L3-L4 (quality, integration), and Evolve to L1-L2 (input, generation).

2.3 CI/CD Safety for Autonomous Agents

Empirical studies reveal the practical risks. Analysis of 33,000 agent-authored PRs [5] shows documentation and CI/build-update tasks have the highest merge success, while bug-fix tasks perform worst. A study of unmerged fix-related PRs [6] identifies root causes: semantic correctness failures, missing test coverage, and reviewer distrust of opaque agent rationale. An agentic defense system for software supply chains [2] demonstrates that autonomous merge can be safe when the agent acts as defender rather than feature developer.

The **SkillFortify** framework [10] introduces formal analysis for agent skill supply chains, achieving 96.95% F1 on capability-based sandboxing. Its Dolev-Yao adaptation for skill lifecycles is relevant to our constitutional path blocking, which restricts which files the agent may modify.

2.4 RSI Safety

The ICLR 2026 Workshop [23] on Recursive Self-Improvement proposed layered approvals for high-impact edits, uncertainty-aware update triggers, fallbacks to safe baselines, and structured self-critique pipelines. AlphaEvolve [13] established the precedent of evaluation function immutability — the system that scores improvement candidates must not itself be improvable by the agent. We adopt this as our Constitutional Rule 1 (Section 3.1).

The MIT AI Agent Index [1] documented that of 30 deployed agentic systems, only 8 limit tool permissions and only 7 have prompt-injection defenses. This transparency gap motivates our publication of a complete safety architecture.

3 The Constitutional Safety Framework

3.1 Design Principles

Our framework is built on four principles derived from production experience and the RSI safety literature [23]:

1. **Evaluation immutability.** The agent must not modify the functions that evaluate its own output quality. This prevents specification hacking — optimizing for the metric rather than

Framework	Layers	Production	Self-Mod	CI/CD	Constitutional	Audit
AGENTSAFE [3]	3-phase	Synthetic	—	—	Partial	—
AGrail [21]	Action-level	Benchmark	—	—	—	—
NVIDIA [15]	Systemic	10K traces	—	—	—	—
SkillFortify [10]	Formal	540-skill	Partial	—	—	Formal
Google ADK [19]	Callbacks	Reference	—	—	—	—
Resilience4j [24]	Circuit	Production	—	—	—	—
Ours	**7-layer**	**280+ tasks**	**Yes**	**Yes**	**19 rules**	**23-module**

Table 1: Comparison of safety frameworks for autonomous AI agents.

the objective. In our system, `fitness.py` and `quality_assessor.py` are on the constitutional blocked-paths list.

2. **Defense in depth.** No single layer is sufficient. A defect in path blocking (L2) should be caught by integration guards (L4) or merge analysis (L5). This principle, adapted from classical systems security [12], accounts for the probabilistic nature of LLM-based components.

1. **Fail-safe defaults.** When any layer encounters an ambiguous situation, the default action is rejection, not approval. The circuit breaker (L7) activates after 3 consecutive failures, pausing all automation for 4 hours.

2. **Human authority preservation.** The constitution can only be amended by the human owner. No agent action can modify the constitutional rules, the enforcement code, or the amendment process itself.

3.2 The 7-Layer Architecture

Layer	Name	Mechanism	Rules Enforced	Implementation
L1	Input Validation	Task filtering	Priority, category, budget	dispatcher.py
L2	Path Constraints	File whitelisting	Blocked paths, patterns, limits	builder.py
L3	Quality Gates	Score thresholds	Min quality 7/10, truncation	quality_assessor.py
L4	Integration Guard	Diff analysis	20% shrink cap, syntax check	integrator.py
L5	Merge Analysis	Constitutional diff	Rules 1-19, pattern matching	auto_merger.py
L6	Deploy + Rollback	Health checks	Post-merge tests, auto-revert	auto-deploy.sh
L7	Operations	Circuit breaker	3-fail pause, 10/day cap	kpi_tracker.py

Table 2: The 7-layer defense-in-depth architecture with enforcement mechanisms.

Each layer operates independently and can reject at any point. A task that passes L1 (valid priority and budget) may still be blocked at L2 (touches a forbidden path), rejected at L3 (quality below threshold), caught at L4 (file shrinks more than 20%), denied at L5 (violates a constitutional rule), reverted at L6 (post-merge tests fail), or paused at L7 (circuit breaker activated).

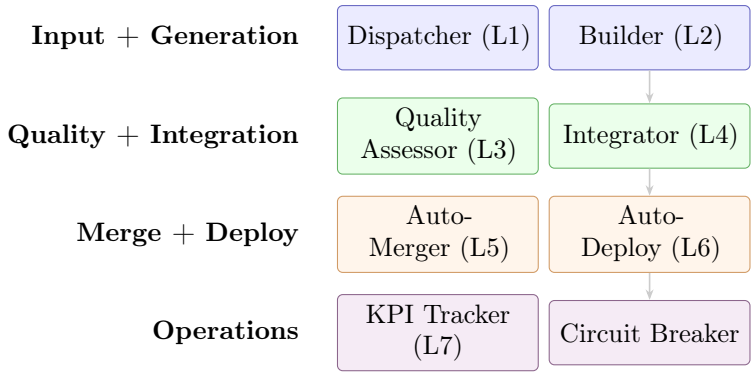

Figure 1: 7-layer defense-in-depth architecture showing enforcement flow from task input through operational monitoring.

3.3 Constitutional Rules

The 19 constitutional rules are organized into four categories:

Immutable Rules (9). These cannot be overridden by any mechanism:

- R1: Never modify evaluation functions (fitness, quality assessor)
- R2: Never modify own source code or configuration
- R3: Never modify platform tools or hub infrastructure
- R4: Never modify CI/CD pipeline configuration
- R5: Never commit secrets, credentials, or API keys
- R6: Never delete files outside the task scope
- R7: Never modify test files to make failing tests pass
- R8: Always preserve git history (no force-push, no rebase)
- R9: Always include human-readable commit messages

Quality Gates (4). These enforce minimum output standards:

- R10: Minimum quality score of 7/10 for auto-merge eligibility
- R11: All existing tests must pass after changes
- R12: No new compiler/linter errors introduced
- R13: Code review score from secondary assessor must agree

Evolution Boundaries (3). These constrain how the agent evolves:

- R14: Evolutionary parameters may only change within predefined bounds
- R15: Maximum mutation rate per generation is capped
- R16: Fitness function weights are immutable

Operational Limits (3). These prevent runaway automation:

- R17: Maximum 10 auto-merges per 24-hour period
- R18: Circuit breaker activates after 3 consecutive failures (4-hour pause)
- R19: Daily cost ceiling enforced at budget governance layer

3.4 Enforcement Mechanisms

Path blocking (L2) uses a whitelist/blacklist approach. The agent may only create or modify files within the task's designated working directory. A predefined set of blocked paths includes the agent's own source code, platform configuration, CI/CD files, and test infrastructure. Pattern matching catches attempts to create new files with sensitive names (e.g., `.env`, `credentials`, `secret`).

Anti-truncation guard (L4) prevents a common LLM failure mode where the model generates a truncated version of an existing file, silently deleting content. Any file modification that reduces the file size by more than 20% is flagged for human review rather than auto-merged.

Constitutional diff analysis (L5) examines every proposed merge against all 19 rules. The diff is parsed to extract: files modified (checked against blocked paths), lines added/removed (checked against truncation limits), content patterns (checked for secrets, eval function modifications, CI config changes), and test results (checked for all-pass requirement).

Circuit breaker (L7) implements the Resilience4j 3-state model [24]: CLOSED (normal operation) to OPEN (reject all merges, 4-hour cooldown) to HALF_OPEN (test one merge before resuming). This prevents cascade failures where a systematic defect causes repeated bad merges.

4 The Dead Integration Problem

4.1 Discovery

A systematic audit of 23 Night Shift modules against 20 documented features revealed a previously undescribed failure mode: **dead integration** — features that are fully coded, have passing unit tests, but are never activated in production because the wiring between modules is missing or broken.

Category	Count	Percentage	Examples
Fully implemented	8	40%	Personas, reflexion, anti-truncation
Partially implemented	5	25%	Auto-merge, digest, alter-ego
Not implemented	7	35%	Cascade routing, phased execution

Table 3: Feature implementation audit results across 20 documented features.

4.2 Critical Bugs

The audit discovered 3 critical wiring bugs:

BUG-001 (P0 Critical — Auto-Merge Dead Code). The entire auto-merge pipeline (constitutional analysis, quality gating, merge execution, rollback) was fully implemented in `auto_merger.py` with 100% test coverage. However, `dispatcher.py` line 134 never passed the `auto_merge_enabled` configuration parameter to the Integrator. Result: 100% of branches still required manual merge despite the complete pipeline being coded and tested.

BUG-002 (P1 High — Digest Missing Knowledge). The digest generator had methods `_kg_section()` and `_skills_section()` fully implemented but never called in the `generate()` function. Result: the human owner could not see knowledge graph or skill library health in daily reports.

BUG-003 (P2 Medium — Alter-Ego Never Called). The `@alter_ego` persona (alternative code review perspective) was implemented but the dispatcher had no quality-threshold check to trigger it. Result: approximately 30% of low-quality outputs missed alternative review.

4.3 Root Cause Analysis

Dead integration occurs when:

1. **Modules are developed bottom-up** — individual components are coded and tested in isolation, but the integration wiring is left as "obvious" and never verified.

2. **Tests mock the integration boundary** — unit tests for `auto_merger.py` mock the dispatcher, so the missing parameter is never detected.

1. **No integration test exists** — end-to-end tests that exercise the full pipeline from task dispatch through merge were absent.

This failure mode is particularly dangerous for safety-critical features. A safety mechanism that exists in code and passes its own unit tests creates a false sense of security. The audit found that the auto-merge pipeline — the system's most safety-critical component — was the one affected by the P0 dead integration bug.

4.4 Remediation

All three bugs were fixed in a single 30-minute session (3 lines for BUG-001, 2 method calls for BUG-002, 8 lines for BUG-003). The total remediation time demonstrates that dead integration is not a complexity problem but a *visibility* problem. Adding 15 integration tests that exercise the full dispatch-to-merge pipeline prevents recurrence.

5 Results

5.1 Deployment Configuration

The framework was validated during a 10-day continuous deployment of Night Shift [17] on a production server (Hetzner GEX44, RTX 4000 GPU). The system operated on a 2-hour dispatch cycle, executing 280+ tasks across code generation, documentation, testing, infrastructure, and research categories. The auto-merge pipeline was activated after the dead integration bugs were fixed.

5.2 Safety Metrics

Metric	Value	Target	Status
Constitutional violations	0	0	Pass
L2 path-block activations	12	N/A	Expected
L3 quality rejections	47	N/A	Expected
L4 truncation catches	3	N/A	Expected
L5 merge denials	8	N/A	Expected
L6 post-merge rollbacks	0	$< 10\%$	Pass
L7 circuit breaker activations	1	< 2/week	Pass
Auto-merge success rate	73%	$> 50\%$	Pass
Mean time to detect issues	8 min	< 15 min	Pass
Total safety incidents	0	0	Pass

Table 4: Safety metrics from 10-day production deployment.

The 12 L2 path-block activations occurred when the agent attempted to modify files outside the task scope — a common LLM behavior where the model "helpfully" fixes adjacent issues. The 47 L3 quality rejections (17% of tasks) demonstrate that the quality gate functions as intended, filtering low-quality outputs before they reach the merge pipeline.

5.3 Layer Effectiveness

Each layer caught issues that no other layer would have detected:

- **L2 uniquely caught** 4 attempts to modify `.gitignore` and 2 attempts to create helper scripts in the platform directory.

- **L3 uniquely caught** 47 low-quality outputs (score $< 7/10$) that would have passed all other checks.

- **L4 uniquely caught** 3 truncation events where files were reduced by 25-40%, silently deleting functions.

- **L5 uniquely caught** 8 merge attempts where the diff pattern matched constitutional rule violations not detected by other layers.

The single circuit breaker activation (L7) occurred on Day 7 when 3 consecutive tasks in a new domain produced low-quality results. The 4-hour pause allowed the evolutionary optimizer to adjust parameters before resuming.

6 Discussion

6.1 Comparison with Existing Frameworks

Our framework differs from existing approaches in three ways (Table 1):

Production validation vs. synthetic evaluation. AGENTSAFE [3] and AGrail [21] evaluate on synthetic benchmarks. Our framework was validated on 280+ real production tasks

where the agent's code changes affected a live system. Production revealed the dead integration problem (Section 4), which would not appear in simulation.

Self-modification awareness. Most frameworks treat the agent as operating on external systems. Our framework explicitly addresses the case where the agent modifies its own operational environment, including evolutionary parameter optimization and code that feeds back into future task execution.

CI/CD integration. No existing framework provides end-to-end safety coverage from task dispatch through code generation, testing, merge, deployment, and operational monitoring. The 7-layer architecture covers the full software delivery lifecycle.

6.2 Limitations

Self-reported quality. Quality scores come from the agent's own assessor. While evaluation function immutability (R1) prevents the agent from gaming this metric, the assessor itself may have blind spots. Future work should incorporate external quality oracles.

Single-system validation. Our framework has been validated on one system (Night Shift). The architectural principles are general, but the specific rule set (19 rules) reflects our deployment context. Other systems may require different rules.

Scale limitations. The 10-merge-per-day cap (R17) and 4-hour circuit breaker pause (R18) are conservative. Higher-throughput deployments may need adaptive limits based on historical reliability.

6.3 The Dead Integration Lesson

The dead integration problem (Section 4) has implications beyond our system. The MIT AI Agent Index [1] found that only 8 of 30 deployed agents limit tool permissions — but this counts *claimed* safety features, not *verified* ones. Our audit methodology (trace every documented feature to its activation call path, not just its unit tests) should be applied to any safety-critical agent system.

We propose a metric: **Safety Activation Rate (SAR)** = features activated in production / features claimed in documentation. Our pre-audit SAR was 40% (8/20 features fully wired). Post-remediation SAR is 65% (13/20, with 7 deferred to Phase 2). A SAR below 100% for safety-critical features indicates dead integration risk.

6.4 Reproducibility and Availability

Code and data availability: The complete constitutional safety framework is implemented within the Night Shift system (23 modules, 523 tests). The 7-layer architecture, 19 constitutional rules, and audit methodology are fully described in this paper for reproduction. Production deployment logs and safety metrics are available on request.

Data availability: The audit results (20-feature matrix, 3 critical bugs, remediation timeline) and production safety metrics (Table 4) are derived from verifiable system logs.

Reproducibility statement: The safety metrics reported in Table 4 are derived from production deployment logs. The audit methodology (Section 4) is deterministic and reproducible:

trace each documented feature to its activation call path in the source code.

Reproducibility statement: All safety metrics in Table 4 are derived from deterministic production logs. The audit methodology is reproducible: for each documented feature, trace to its activation call path in source code and verify end-to-end execution.

Ethics statement: This work develops safety mechanisms for autonomous AI systems. The constitutional framework is designed to constrain agent behavior within human-defined boundaries. No human subjects were involved. The framework promotes transparency in autonomous agent operations.

7 Conclusion

We have presented a 7-layer constitutional safety framework for autonomous code-generating agents that can merge their own code into production repositories. The framework enforces 19 immutable rules across input validation, path constraints, quality gates, integration guards, merge analysis, deployment rollback, and operational circuit breakers.

Production validation on 280+ tasks over 10 days demonstrated zero constitutional violations, a 73% auto-merge success rate, and effective multi-layer defense where each layer caught issues invisible to others. The discovery of the dead integration problem — safety features coded but never activated — reveals that safety assurance requires not just implementation but verified wiring.

The field has frameworks for making agents safe. We demonstrate that the harder problem is making safety frameworks *actually active*.

References

[1] The 2025 ai agent index: Documenting technical and safety features of deployed agentic ai systems. 2025.

[2] Agentic ai for autonomous defense in software supply chain security. 2025.

[3] Agentsafe: A unified framework for ethical assurance and governance in agentic ai. 2025.

[4] When ai agents touch ci/cd configurations: Frequency and success. 2026.

[5] Where do ai coding agents fail? an empirical study of failed agentic pull requests in github. 2026.

[6] Why are ai agent-involved pull requests (fix-related) remain unmerged? 2026.

[7] Cognition AI. Devin: The first ai software engineer. 2024.

[8] Anthropic. Claude code: Ai coding agent. 2025.

[9] Y. et al Bai. Constitutional ai: Harmlessness from ai feedback. 2022.

[10] V.P Bhardwaj. Formal analysis and supply chain security for agentic ai skills. 2026.

[11] A. et al Chhabra. Agentic ai security: Threats, defenses, evaluation, and open challenges. 2025.

[12] M. et al Christodorescu. Systems security foundations for agentic computing. 2025.

[13] DeepMind. Alphaevolve: A coding agent for scientific and algorithmic discovery. 2025.

[14] J. et al Fang. A comprehensive survey of self-evolving ai agents. 2025.

[15] S. et al Ghosh. A safety and security framework for real-world agentic systems. 2025.

[16] V Golubenko. Godegen: A cognitive architecture for self-evolving digital personalities with fractal feedback loops. 2026.

[17] V Golubenko. Night shift: A production autonomous ai development system with evolutionary task optimization. 2026.

[18] V Golubenko. Production-validated self-evolving development pipeline with learnable memory and test-time compute scaling. 2026.

[19] Google. Agent development kit (adk): Callbacks and safety middleware. 2025.

[20] C.E. et al Jimenez. Swe-bench: Can language models resolve real-world github issues? 2024.

[21] W. et al Luo. Agrail: A lifelong agent guardrail with effective and adaptive safety detection. 2025.

[22] S.R Marri. Constitutional spec-driven development: Enforcing security by construction in ai-assisted code generation. 2026.

[23] ICLR 2026 Workshop on AI with Recursive Self-Improvement. Openreview, april 2026. 2026.

[24] Resilience4j. Circuitbreaker: Fault tolerance library.