

The Architectural Evolution of Large Language Model Control Planes:

From Prompt Engineering to
Autonomous Loop Systems

 Research Paper

Abstract

We present a four-phase taxonomy for the control planes used to interface with LLMs on software-engineering tasks: **Prompt Engineering** (manual text steering), **Context Engineering** (curating what enters the context window), **Harness Engineering** (multiagent orchestration over rigid, hand-authored routing graphs), and **Loop Engineering** (a declarative goal paired with a recursive, self-managing control loop that terminates only on an independently verified condition). We formalize the fourth phase as a discrete-time control system and describe six structural mechanisms that distinguish a loop from a harness: a Maker-Checker split restricted to deterministic environmental signals; externalized, disk-persisted state; validation gates keyed to compiler/test exit codes rather than self-assessment; structural delta-rollback; iteration compression; and a meta-loop that rewrites the primary loop's own configuration. We also formalize two multi-agent loop-detection mechanisms: static cycle detection via strongly connected components, and a proposed dynamic detector for implicit deadlocks based on state-embedding similarity and stagnant error. **This is a synthesis and formalization position paper, not a claim of priority on every mechanism in isolation.** Independent work has converged on several of the same ideas — in one case (SCC-based loop detection) within weeks of this writing, validated across 6,549 repositories at a scale we do not attempt to match. We give, mechanism by mechanism, what is corroborated by existing systems, what differs and how, and what remains an unvalidated proposal.

1. Introduction

The interface architecture governing LLM-driven software engineering has moved, in roughly three years, from a human manually rephrasing strings to a system that manages its own state, verifies its output against deterministic signals, and decides for itself when to stop. We organize this into four phases and argue the bottleneck shifts correspondingly: from how to phrase the prompt, to what data enters the context window, to how to route between tools, to how to bound and verify a recursive loop with no human in it.

Several of the mechanisms we describe already exist, independently, in current literature and production systems — sometimes under different names, sometimes as a looser version of the same constraint, several published in the same months as this writing. We do not think this diminishes the contribution: no single treatment we are aware of assembles them into one architecture with one formal control-loop model. Section 2 names the closest prior art for each mechanism, including one case where independent work published within weeks of this one already validates the same detection technique at a scale we do not attempt to match. Our contribution is the taxonomy, the formal model (Section 3), and an honest accounting of what is corroborated, tightened, or proposed (Section 7).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA
© 2026 Copyright held by the owner/author(s).
Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

2. Relatedwork and Terminology

“Context engineering” is in wide informal use for curating the context window rather than phrasing prompts; we could not attribute a specific coinage and cite it as community usage. “Agent harness” has a direct primary source: Anthropic’s Claude Agent SDK is described as “a powerful, general-purpose agent harness” [1] — the same post documents agents marking work complete without proper testing, fixed by forcing tool-based verification, and persists state across fresh-context sessions via a progress-log file and a feature-status manifest.

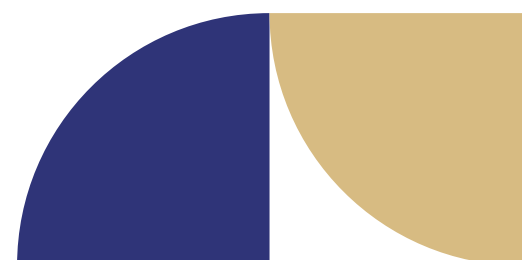
Reflexion [6] runs Actor/Evaluator/Self-Reflection roles but allows the Evaluator to be “a different instantiation of an LLM,” not a deterministic source. GA-Rollback [3] separates a generator from an assistant that triggers rollback on its own LLM judgment of “incorrect actions.” CodeAct [9] is a single-persona self-revising loop with no separate verifier. Our Maker-Checker rule (Section 4) is stricter than all three: the checker is barred from issuing a judgment call and restricted to compiler, linter, and test-suite exit codes. AlphaCodium [5] already runs a test-driven iterative flow, and its own title — From Prompt Engineering to Flow Engineering — prefigures our rhetorical move at a narrower, single-loop scale.

MemGPT [4] is the clearest prior art for our externalized-state and compression mechanisms: its virtual context management moves data between in-context and external tiers via an explicit OS-paging analogy. AGENTS.md, the manifest convention we use as a running example, is real and multi-vendor (OpenAI Codex, Amp, Google Jules, Cursor; stewarded by the Linux Foundation’s Agentic AI Foundation). A separately-named skill-configuration file (SKILL.md, rewritten by our Meta-Loop mechanism) is a convention we have direct operational familiarity with from Claude

Code’s Skill system, though we did not find it independently documented as a cross-vendor standard the way AGENTS.md is.

GA-Rollback and AgentDebug [11] establish rollback on localized failure via natural-language feedback at the trajectory-step level, not a computed structural diff tree — the source-artifact-level distinction our Adaptive Delta Rollback makes. MetaSkill-Evolve [10], essentially contemporaneous with this writing, runs task skills on a fast loop and the meta-skill on a slower loop “under the same pipeline applied to itself” — closing the same gap as our Evolutionary Meta-Loop, differing mainly in cadence (fixed-H-iteration vs. our failure-triggered trigger).

Most importantly: Hou et al. [2], submitted within weeks of this paper, names Infinite Agentic Loops and builds IAL-Scan, a static analyzer that recovers an agent dependence graph and computes strongly connected components over it, validated across 6,549 repositories at 91.9% precision — as far as we can establish, the same core technique as Section 6’s static detector. We report ours for taxonomic completeness and defer to their empirical results rather than claim priority. Tacheny [7] formalizes agentic loops as dynamical systems in embedding space with three regimes — a complementary lens on the same trajectory we formalize (Section 3), classifying a single loop’s regime rather than multi-agent deadlock. We found no prior art for Section 6’s dynamic detector — sliding-window cosine similarity combined with stagnant error, aimed at implicit deadlocks — and present it as unvalidated.



3. Taxonomy and Formalization

- **2022–2024, Prompt Engineering.** The LLM is a stateless mapping; the human is the control loop, manually steering via Chain-of-Thought and few-shot examples.
- **2025, Context Engineering.** Focus shifts to retrieval and curation of what enters the context window.
- **Early 2026, Harness Engineering.** Multi-agent orchestration over DAGs/state machines; rigid, hand-authored routing; prone to self-evaluation bias.
- **Mid-2026, Loop Engineering.** A declarative goal paired with a recursive loop that manages its own state and runs until an independent gate verifies success.

Phases I–II are a static mapping $Y = f(P; \theta)$, augmented in Phase II by a context vector C : $Y = f(P, C; \theta)$. Phase III chains fixed nodes, $Y_{n+1} = f(Y_n, P_{n+1}; \theta)$, so an error at Y_n propagates unchecked. Phase IV tracks environmental state X_t against goal G via an evaluation operator D producing an error vector $E_t = D(G, X_t)$; a Maker M inspects E_t and a skill manifest K to produce a mutation $A_t = M(X_t, E_t, K)$; an execution utility T transitions the state, $X_{t+1} = T(X_t, A_t)$; and the loop recurses until $D(G, X_{t+1}) = 0$.

Remark 1. This is a standard observe-act-transition control loop; we claim no novelty in the control-theoretic structure. Tacheny [7] studies the same trajectory X_t as a point in embedding space rather than gating it against a deterministic D — the two views are compatible, not competing.

4. Core Pillars of the Loop Architecture

4.1 The Maker-Checker Separation Rule

A single model instance must never evaluate its own output: doing so introduces confirmation bias, where the model rationalizes its own errors. We enforce a physical runtime separation — a Maker persona with write access to the environment, and a distinct, separately prompted Checker persona with read-only access to logs, build flags, and compilation outcomes. Unlike Reflexion’s Evaluator or GA-Rollback’s assistant (Section 2), the Checker here is barred from issuing its own judgment call; it may only report signals already computed by a compiler, linter, or test runner.

4.2 Externalized State Registries

To avoid the performance degradation associated with very large context windows, intermediate logs, reasoning trajectories, and historical failures are flushed to disk (e.g. an AGENTS.md-style manifest) at the end of every loop cycle. The context window is wiped and reinstantiated at iteration $t+1$ using only the high-level goal, the current file diffs, and the latest disk-persisted summary, bounding input token growth to linear rather than exponential. This generalizes Anthropic’s per-session state persistence [1] to a per-iteration wipe.

4.3 Deterministic Environmental Validation Gates

While the loop’s internal reasoning is probabilistic, its termination criterion must be absolute: the loop cannot file a pull request, commit code, or declare an assignment complete based on its own assertion of success. It terminates only when the environment returns a clean exit code from a compiler, linter, or test suite — the same discipline AlphaCodium applies to a single test-driven loop [5] and Anthropic’s harness enforces after observing that self-assessed completion was unreliable [1].



5. Advanced Execution Techniques

5.1 Adaptive Delta Rollbacks

When a loop writes many lines across multiple files and a compiler fails on one localized fault, a full reset to the prior state discards all successful intermediate work. We instead compute a structural diff tree, let the Checker isolate the precise fault block, and revert only that block (Δ), feeding just the reverted context back to the Maker. GA-Rollback [3] and AgentDebug [11] establish the same failure-and-recovery pattern at the level of an agent’s trajectory step, via natural-language feedback rather than a computed diff tree; whether the structural-diff version offers a measurable advantage over step-level natural-language rollback is untested here.

5.2 Context-Swapping State Caches

Long-running loops risk forgetting early constraints even with externalized state. We compress iteration logs $t-1$ through $t-5$ into a compact summary via an automated summarization pass, dropping raw history and retaining only target files, compiler errors, and the summary — a direct application of MemGPT’s virtual context management [4] to loop-iteration history rather than conversational turns.

5.3 Evolutionary Meta-Loops

Base configuration (e.g. a skill manifest) may be sub-optimal for a specific codebase. A slow-cadence loop runs above the primary Maker-Checker loop: if the primary loop exceeds a failure-count threshold on a given task, the meta-loop inspects the execution trace, rewrites the underlying configuration, and restarts the primary loop. MetaSkill-Evolve [10] runs the same fast/slow split on a fixed iteration cadence rather than a failure trigger; which cadence policy converges faster is, again, untested here.

Table 1: Structural dimensions across the four phases.

	I: Prompt	II: Context	III: Harness	IV: Loop
Control	Manual, humanin-loop	Programmatic retrieval	Static graph routing	Autonomous recursive loop
State	Volatile context	Ephemeral perquery	Volatile, crossnode	Externalized, on disk
Gate	Human operator	Vector similarity	Self-grading LLM	Maker-Checker, exit codes
Failure	Hallucination	Retrieval drift	Error compounding	Localized rollback

6. Algorithmic Loop Detection In Multi-Agent Graphs

Static graph detection (explicit loops). Model the communication layout as adjacency matrix A , $A_{ij} = 1$ if agent i can delegate to agent j . Running Tarjan’s strongly-connected-components algorithm [8] over this graph identifies explicit loops before a token is spent: an SCC with more than one agent vertex is a design-level cycle. This is essentially Hou et al. [2]’s independently-validated technique (Section 2); we present it for taxonomic completeness and defer to their results.

Dynamic entropy and state-similarity detection (implicit loops). Implicit deadlocks are harder: the design looks linear, but two agents stall passing an unresolved error back and forth. Let X_t be the environment-state embedding at iteration t and E_t the error-vector magnitude. Over a sliding window of size k :

$$\text{Sim}(X_t, X_{t+k}) = \frac{X_t \cdot X_{t+k}}{\|X_t\| \|X_{t+k}\|} \tag{1}$$

We flag a stalled implicit loop when

$$(\text{Sim}(X_t, X_{t+k}) > \tau) \wedge (\|E_{t+k}\| \geq \|E_t\|) \wedge (\|E_t\| > 0), \tag{2}$$

with $\tau \approx 0.95$ suggested as a starting point, not a measured optimum. Intuitively: the environment has stopped changing in any meaningful way, yet the error it is supposed to be resolving has not shrunk. On this condition, the system can kill the run or trigger an Adaptive Delta Rollback (Section 5). We found no prior art for this specific combination (Section 2); it is an unvalidated proposal, not a corroborated mechanism.

7. Discussion: Corroborated vs. Proposed

Corroborated. Externalized state, context compression, localized rollback, deterministic test-driven termination, and meta-level rewriting all have direct real analogs (Section 2). What we tighten — deterministic-only checking, structural-diff rollback, failure-triggered meta-evolution — is not demonstrated to be an improvement here.

Independently converged upon, validated elsewhere at greater scale. Static SCC-based explicit loop detection is the same core technique Hou et al. [2] validated across 6,549 repositories at 91.9% precision. We defer to their results.

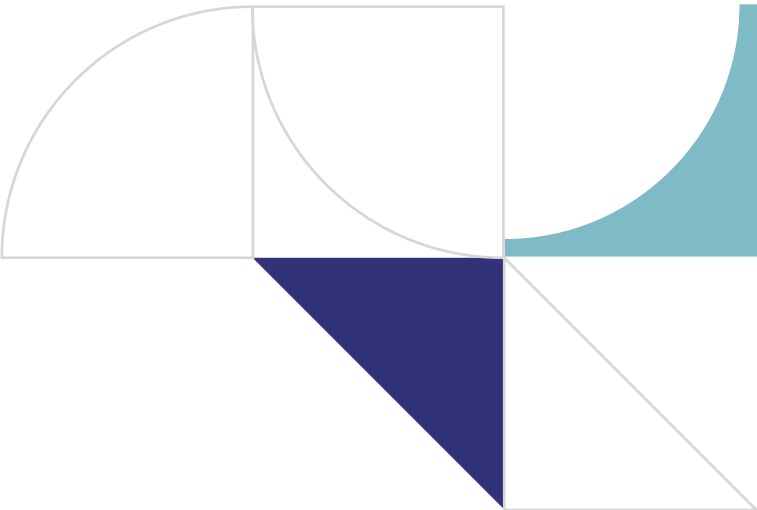
Proposed, unvalidated. The dynamic cosine-similarity detector, its threshold $\tau = 0.95$, and the taxonomy’s naming convention itself are, as far as our research could establish, not present in the existing literature in this form and have not been measured against a real deployment. Minimally needed: a labeled implicit-deadlock corpus to measure precision/recall against τ ; a controlled comparison of structural-diff vs. natural-language-feedback rollback; and a comparison of failure-triggered vs. fixed-cadence meta-loops on convergence speed. None of these experiments have been run.

Table 2: Taxonomy of loop diagnostics.

Classification	Detection	Remediation
Explicit design loop	Static SCC over agent adjacency	Hardcode a deterministic exit validator into the graph path
Implicit hallucination loop	Dynamic cosine similarity, stagnant error	Break the run; trigger a meta-loop to alter configuration
Token-bleed cascade	Running tokenvolume auditing	Flush logs to external state registry

8. Conclusion

Loop engineering is a maturation of the AI engineering stack, from a linguistic art form to a structural systems discipline. Its individual mechanisms are mostly not new in isolation — they are scattered across work still being written as this paper is, in one case within the same weeks. Our contribution is the taxonomy connecting all four phases, a formal control-loop model for the fourth, and an explicit account of what is corroborated, tightened, or open. The frontier this leaves is empirical: measuring whether the constraints we tighten actually improve convergence and reliability over the looser variants already in production.



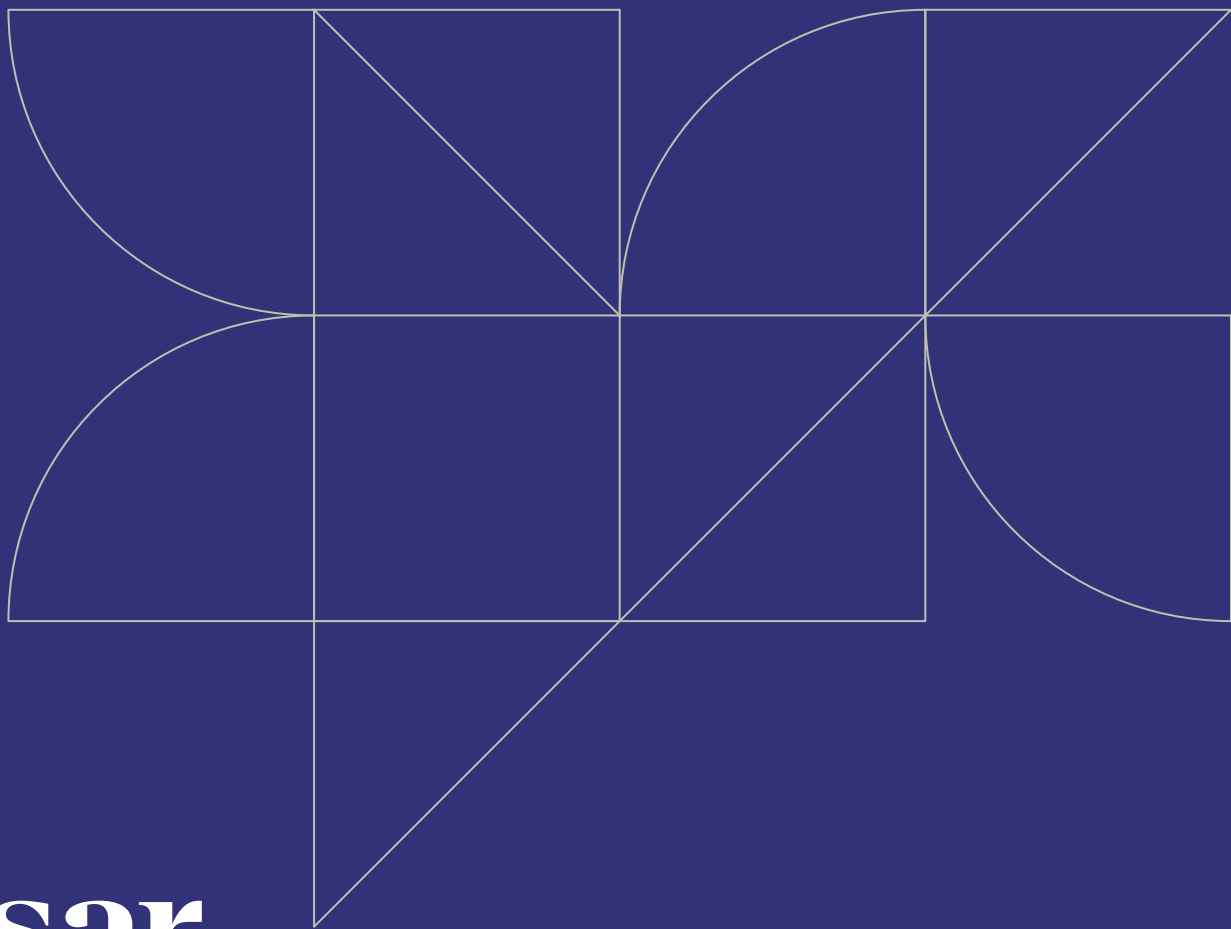
References

1. Anthropic. 2025. Effective harnesses for long-running agents. Anthropic Engineering Blog.
2. Xinyi Hou, Shenao Wang, Yanjie Zhao, and Haoyu Wang. 2026. When Agents Do Not Stop: Uncovering Infinite Agentic Loops in LLM Agents. arXiv preprint arXiv:2607.01641 (2026).
3. Xingzuo Li, Kehai Chen, Yunfei Long, Xuefeng Bai, Yong Xu, and Min Zhang. 2025. Generator-Assistant Stepwise Rollback Framework for Large Language Model Agent. In Proceedings of EMNLP 2025.
4. Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. arXiv preprint arXiv:2310.08560 (2023).
5. Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. arXiv preprint arXiv:2401.08500 (2024).
6. Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv preprint arXiv:2303.11366 (2023).
7. Nicolas Tacheny. 2025. Geometric Dynamics of Agentic Loops in Large Language Models. arXiv preprint arXiv:2512.10350 (2025).
8. Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1, 2 (1972), 146–160.
9. Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In Proceedings of ICML 2024.
10. Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. 2026. MetaSkill-Evolve: Recursive Self-Improvement of LLM Agents via Two-Timescale Meta-Skill Evolution. arXiv preprint arXiv:2607.05297 (2026).
11. Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where LLM Agents Fail and How They Can Learn From Failures. arXiv preprint arXiv:2509.25370 (2025).

Authors:

 **Saipriyadarshan Cindula**
Zensar

 **Nikhil Gajanan Bajaj**
Zensar



zensar
An  **RPG** Company

At Zensar, we're 'experience-led everything.' We are committed to conceptualizing, designing, engineering, marketing, and managing digital solutions and experiences for over 145 leading enterprises. Using our 3Es of experience, engineering, and engagement, we harness the power of technology, creativity, and insight to deliver impact.

Part of the \$4.8 billion RPG Group, we are headquartered in Pune, India. Our 10,000+ employees work across 30+ locations worldwide, including Milpitas, Seattle, Princeton, Cape Town, London, Zurich, Singapore, and Mexico City.

For more information, please contact: info@zensar.com | www.zensar.com