Loop Engineering: Building Blocks, Adoption, and Impact

Jai Lal Lulla
Singapore Management University
Singapore, Singapore
jailal.l.2025@phdcs.smu.edu.sg

Vahram Nersesyan
Heidelberg University
Heidelberg, Germany
vahram.nersesyan@stud.uni-heidelberg.de

Seyedmoein Mohsenimofidi
Heidelberg University
Heidelberg, Germany
s.mohsenimofidi@uni-heidelberg.de

Christoph Treude
Singapore Management University
Singapore, Singapore
ctreude@smu.edu.sg

Sebastian Baltes
Heidelberg University
Heidelberg, Germany
sebastian.baltes@uni-heidelberg.de

Abstract

Over the past months, the way developers direct agentic AI coding tools has moved up several levels of abstraction, from phrasing prompts to engineering context to configuring the harness around the model. In June 2026, practitioners began to describe a further level called *loop engineering*: Instead of prompting an agent interactively, developers design systems that prompt agents for them. These systems start agent runs on a schedule or on repository events and stop them when a machine-checkable condition holds. The term spread rapidly, accompanied by bold claims and vocal skepticism, but its adoption in software projects has not been measured. We present an exploratory review of the emerging gray literature, which largely agrees on what a well-engineered loop contains: triggered agent runs bounded by machine-checkable stop conditions, persistent state files, verifier sub-agents, token budgets, and defined points of escalation to humans. From this review, we derive a research agenda for the empirical study of loop engineering in open-source projects, analyze which of its aspects are traceable from repository data, and report an exploratory mining study of 36,710 software repositories. We confirmed the operation of autonomous agent loops in 217 of the 256 repositories our heuristics matched. The repositories commit the configuration around these loops, but almost none commits the state files the discourse prescribes, and the loops' runtime state remains outside version control. We conclude by outlining a planned controlled study of agent autonomy levels and their effect on effort and outcomes.

CCS Concepts

• **Software and its engineering**;

Keywords

Loop Engineering, Context Engineering, Harness Engineering, AI Agents, Agentic Coding Tools, Generative AI, Open Source

ACM Reference Format:
Jai Lal Lulla, Vahram Nersesyan, Seyedmoein Mohsenimofidi, Christoph Treude, and Sebastian Baltes. 2026. Loop Engineering: Building Blocks, Adoption, and Impact. In *2nd Journal Ahead Workshop (JAWs@ASE 2026), October 12–16, 2026, Munich, Germany. Under review*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

JAWs@ASE 2026 (under review), Munich, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Agentic AI coding tools such as Claude Code or OpenAI Codex interpret a goal, decompose it into steps, invoke tools, and iterate until they consider the goal achieved [12]. For most developers, working with these tools still means typing a request, reading the result, and typing the next one, but some prominent practitioners claim to have abandoned this interactive mode altogether. In an interview published on 2 June 2026, Boris Cherny, the creator of Claude Code [7], said: "*I don't prompt Claude anymore. I have loops that are running. They're the ones that are prompting Claude and kind of figuring out what to do. My job is to write loops*" [55]. On 7 June 2026, Peter Steinberger, the author of OpenClaw, posted a similar message: "*you shouldn't be prompting coding agents anymore. You should be designing loops that prompt your agents*" [48], a post that had reached more than eight million views by mid-July [37]. The same day, Addy Osmani published a blog post naming the emerging practice *loop engineering* and defining it as "*replacing yourself as the person who prompts the agent. You design the system that does it instead*" [40].

The term extends a trajectory along which the unit of concern has moved from individual prompts (prompt engineering) to everything a model sees in its context window (context engineering) and on to the mechanisms configured around the model (harness engineering), which Section 2.1 describes. *Loop engineering*, as characterized in the practitioner discourse, sits one abstraction level above the harness. Where a harness equips a single agent run, a loop governs many such runs over time. It decides what starts each run (e.g., a schedule or an event such as a failing build or an opened issue), how its results are verified and persisted, and when to escalate to a human [20, 40].

We argue that this development deserves the attention of empirical software engineering researchers, for three reasons. First, the practice is no longer hypothetical: Recurring and goal-driven agent runs became first-class commands (`/loop`, `/goal`) in Claude Code, Codex, and Hermes between March and May 2026, with community plugins for OpenCode and Pi [37]. Second, the claims attached to loop engineering are substantial but rest almost entirely on anecdotes and self-reported productivity numbers, while experienced engineers voice equally strong skepticism, calling loops renamed cron jobs and warning about token costs and reviewer fatigue [37, 38].

Third, adjacent discourse-driven terms such as *vibe coding* attracted surveys and empirical studies within months [14, 43], showing how quickly practitioner vocabulary shapes research agendas. Our goal is to move loop engineering from a practitioner term to an empirically grounded object of study, through three research questions:

RQ1 *How is loop engineering defined in the gray literature, and how does it relate to prompt, context, and harness engineering?*

RQ2 *Which loop engineering practices are adopted in open-source projects, and which aspects of loop engineering are traceable from repository data?*

RQ3 *How does the degree of loop autonomy affect development outcomes and developer effort?*

We answer **RQ1** by situating loop engineering in the trajectory from prompt via context and harness engineering (Section 2) and through an exploratory review of the gray literature that consolidates it into a working definition (Section 3). We address **RQ2** with a traceability analysis and an exploratory mining study of 36,710 engineered software repositories from an existing AI configuration dataset [10] (Section 4.1). For **RQ3**, we describe a controlled study of agent autonomy levels (Section 4.2) that we will execute for the planned journal extension.

2 Background

This section describes the practices loop engineering extends (Section 2.1), how the term emerged (Section 2.2), and adjacent academic work (Section 2.3).

2.1 From Prompts to Context to Harnesses

Prompt engineering was one of the first named practices for directing large language models (LLMs). Catalogs of reusable prompt patterns [53] and a broad survey of prompting techniques [44] systematized how to phrase prompts, and in software projects prompts soon became engineering artifacts in their own right, committed to repositories and evolving like code [49, 51]. *Context engineering* generalized this practice from single instructions to the systematic design of everything a model sees in its context window [19, 30]. For agentic coding tools, its most visible instrument is a repository-versioned context file (e.g., `AGENTS.md` or `CLAUDE.md`), a "*README for agents*" [1] whose structure and evolution mining studies have examined [6, 31]. Early evidence on their effect is mixed: One study associated `AGENTS.md` presence with lower runtime and token consumption [27], and another found that context files generally do not improve task success while raising inference cost by more than 20% [16].

Harness engineering moves the focus from the context to the harness, "*the software layers around the model(s) that drive the agent loop*" [12, 34]. The *agent loop* is the model's repeated cycle of calling tools and acting on the results within a single run. *Loop engineering* governs such runs through structures that start, verify, and restart them over time. In an exploratory study of 2,853 open-source repositories, Galster et al. mapped eight configuration mechanisms (context files, settings, skills, subagents, commands, hooks, rules, and Model Context Protocol (MCP) servers) across five agentic AI coding tools. They found that developers overwhelmingly adopt static context files while rarely using executable mechanisms, concluding that "*harness engineering in open source today is … mostly context engineering*" [11, 12]. Their taxonomy contains no mechanism that governs *when and how often* an agent runs (hooks come closest but fire at fixed lifecycle points within a session), so recurrence, termination, cross-run state, budgets, and graduated human oversight all lie outside that analysis. This is the gap this paper begins to address.

2.2 The Emergence of Loop Engineering

Developers ran agents in loops before the term existed, at both levels that Section 2.1 distinguishes. The first level is iteration inside a single run, the agent loop: In March 2024, Andrew Ng reported that GPT-3.5 "*wrapped in an agent loop*" reached up to 95.1% on HumanEval, against 67.0% for zero-shot GPT-4 [32]. The second level adds recurrence and cross-run state, and it arrived as a shell loop. In July 2025, Geoffrey Huntley published the "*Ralph Wiggum*" technique, "*in its purest form, […] a Bash loop […]* `while :; do cat PROMPT.md | claude-code ; done`" [21]. It works around the finite context window by restarting the agent with a fresh context in every iteration while persisting progress to the file system [37]. Tool vendors then absorbed the technique: Claude Code added a recurring `/loop` command in March 2026, Codex introduced goal objects with machine-checkable completion conditions in April, and Claude Code, Hermes, and other harnesses followed with `/goal` commands in May [37].

The term *loop engineering* was coined in early June 2026, when Steinberger's post, Cherny's remark, and Osmani's blog post appeared within days of each other [37, 40, 48]. Later that month Ng called it "*a hot buzzphrase*" [33]. Osmani described a loop as "*a recursive goal where you define a purpose and the AI iterates until complete*" and listed five capabilities held together by state that persists beyond any single conversation ("*the agent forgets, the repo doesn't*") [40]. A month later, he reported a four-part classification of loops ordered by how much the developer hands off: *turn-based* loops keep each turn under the developer's review, *goal-based* loops hand off the stop condition, *time-based* loops hand off the trigger, and *proactive* loops hand off the prompt itself [41].

Within weeks, an ecosystem formed around these ideas. Cobus Greyling's community repository [18] distilled the discourse into seven loop patterns (e.g., 'daily triage', 'CI sweeper'), four "*readiness levels*" from L0 (documented intent) via L1 (report-only) and L2 (assisted fixes with verifier) to L3 (unattended), catalogs of failure modes and anti-patterns, and tooling for scaffolding, auditing, and cost estimation. André Lindenberg framed loop engineering as "*a different solution architecture, not a different way to use the agent*" and argued that the verifier must be designed first [26]. A self-published synthesis decomposed one loop turn into five "*moves*" (discovery, handoff, verification, persistence, scheduling) [20], and Ng distinguished the agentic coding loop (minutes) from the developer and external feedback loops around it (hours to weeks) [33].

Table 1 summarizes the four layers of directing AI coding agents and their corresponding artifacts. Within the harness layer, Böckeler distinguishes what is built into a coding agent (e.g., the system prompt) from what its users assemble on top [5]. We call these the *inner* and *outer harness*; the table lists outer-harness artifacts, since those are the ones developers assemble.

We consolidate the discourse into the following working definition, which Section 3 grounds element by element in the

Table 1: Four layers of directing AI coding agents [12, 20, 40]. Each layer subsumes the previous one; the harness layer refers to the outer harness.

Layer	Unit of Concern	Example Artifacts
Prompt eng.	One instruction	Prompts, prompt templates
Context eng.	One model call	`AGENTS.md`, `CLAUDE.md`, rules
Harness eng.	One agent run	Skills, subagents, hooks, MCP config.
Loop eng.	Recurring runs	Schedules, stop conditions, state files, run logs, budgets, verifier agents

practitioner sources: ***Loop engineering*** *is the practice of designing automated control structures that repeatedly invoke coding agents, triggered on a schedule or by events, with each run bounded by a machine-checkable stop condition. A well-engineered loop persists state across runs, verifies results independently of the implementation agents, bounds inference costs, restricts what an unattended run may change without approval, and defines when humans should intervene.*

2.3 Adjacent Academic Work

In a preprint, Macedo defines the "*loop specification*" and hand-coded the fifty entries of a public loop catalog for trigger, goal type, verification level, architecture, and terminal states [29]. His corpus contains only loops, so it carries no denominator over software projects: It can show how published loop designs are composed, but not how many projects run one. Our mining study asks that question (Section 4.1). Closest to loop engineering, Geng and Neubig studied *asynchronous* software engineering agents that operate without a human in the loop and reported that dependency-aware plans, isolated workspaces, and test-based verification gates improve long-horizon outcomes [15]. Such systems are evaluated on bounded benchmark tasks [23]. We study the developer practice of designing recurring control structures around them. Autonomy taxonomies grade how much a developer delegates to the agent: Feng et al. define five levels by the user's role, from operator to observer [9], and Cihon et al. measure autonomy from the code that orchestrates the model (in our terminology, the loop) without running the agent [8]. A randomized controlled trial found experienced open-source developers measurably *slowed down* by early-2025 tools, despite believing the opposite [4]. Whether recurring loops change that result is an open question.

3 Exploratory Literature Review

Because the term *loop engineering* was coined by practitioners only weeks before we wrote this article, the relevant knowledge resides almost exclusively in gray literature, a constellation in which including such sources is explicitly recommended [13].

Having followed the discourse on social media since the seed posts of June 2026, the last author compiled a first wave of seven sources published between 7 June and 15 July 2026: Osmani's blog post [40] and one-month retrospective [41], Orosz's newsletter investigation [37] and its 136-comment discussion thread [38], Lindenberg's newsletter article [26], a self-published synthesis note [20], and a snapshot of Greyling's community repository [18]. On 18 July, we added three sources the collected material referenced: Huntley's post introducing the Ralph technique [21], Greyling's essay accompanying his repository [17], and the podcast interview in which Stripe engineer Steve Kaliski describes the company's internal agent system [52]. On 25 July, after consolidating the observations below, we added one further source, Osmani's post on agent autonomy with its slide deck [39]. We included sources that treat loop engineering substantively (define the term, describe building blocks or patterns, or report experiences) and excluded passing mentions, reactions, and reposts.

In parallel, on 15 July 2026, we searched arXiv, DBLP, and Semantic Scholar for uses of the term *loop engineering* and for adjacent academic work (Section 2.3). An LLM-based agent (Claude Code with Fable 5) assisted in this search, in locating the sources the collected material referenced, and in extracting candidate quotes, which the last author checked against the sources. We disclose this in line with the community guidelines for empirical studies involving LLMs [2]. From the collected sources, we extracted proposed definitions, building blocks, claimed benefits, risks, and explicitly skeptical positions, which the last author consolidated into the six observations below (O1–O6) and all authors reviewed.

O1: The definitions converge on handing off the next prompt. Across sources, the definitional core of loop engineering is stable: The developer stops deciding what the agent does next and designs the system that decides instead [20, 26, 40]. The four-part taxonomy Osmani reported (Section 2.2) operationalizes this as a sequence of hand-offs, from the check to the stop condition, the trigger, and finally the prompt itself [41]. In Orosz's thread, one commenter argued that a scheduled or event-triggered agent run is "*still largely a workflow*": It becomes a loop only when an explicit feedback signal carries one iteration's outcome into the next [38].

O2: The novelty of the practice is contested. Many reactions in the collected sources reject the framing of loop engineering as novel: Loops are "*a renamed cron job*", "*automation was a thing before LLMs*", and the term repackages event-driven architecture with, as one commenter put it, "*a fuzzy worker*" [37, 38]. That commenter does not dismiss the practice, however: What is new, on his account, is the non-deterministic worker, which puts the engineering into "*the guardrails around it rather than the trigger*" [38]. A further objection suspects the AI labs' interests behind the practice, since loops consume many tokens, a debate Orosz reports as "*token-maxxing*" [37]. Orosz's own investigation, drawing on roughly 210 practitioner replies, concluded that most concrete examples fall into two familiar buckets, cron jobs and event-based triggers, now executed by more capable workers [37, 38]. Max Kanat-Alexander argues that the practice will disappear into the tools: Anything developers must know about a tool "*would eventually become baked into its default workflow*", so explicit loops around AI agents (e.g., the Ralph loop) were a temporary workaround that features such as `/goal` are already absorbing [37]. Orosz agrees: After `/loop` and `/goal` became built-in commands, loop engineering seemed "*as good as obsolete*" for engineers building regular software, while "*designing loops will remain an important task at the AI infra level*" [37], which by our definition is itself loop engineering. The discourse has not settled whether loop engineering is a new discipline, a transitional technique, or a marketing label.

O3: The proposed building blocks are consistent across sources. Despite the definitional dispute, the sources largely agree on what a well-engineered loop contains: a trigger (a schedule or an event) and a stop condition that decides when a run is done, durable state outside the context window, encoded project knowledge (skills), isolation for parallel work (worktrees), an independent verifier separated from the implementer, connectors to external systems, budgets with a documented way to pause a running loop (a "*kill switch*" [18]), binding constraints on what an unattended run may change without human approval, and defined escalation points to humans [18, 20, 26, 40]. However, the sources are not independent. Greyling's community repository and HuaShu's synthesis note explicitly build on Osmani's posts, and Lindenberg cites the same seed posts, so much of the agreement traces back to one origin. The most emphasized building block of loop engineering is the "*maker/checker*" separation. Osmani calls the existence of a verifier the reason one "*can walk away*" [40], Greyling's verifier skill makes rejection the default, so the agent's output must 'earn' approval [18], and Lindenberg warns that a loop "*satisfies the gate you wrote*", not the goal behind it [26]. Staged adoption is another consistent recommendation: Greyling's readiness levels start report-only (L1) and reach unattended operation (L3) only once the verifier has proven reliable [18]. Osmani's 25 July post revises the eight stages of developer evolution Yegge had published in January 2026 (from near-zero AI use to building one's own orchestrator), treats autonomy as a choice made per task rather than a level a developer reaches and keeps, and separates two dimensions a single ordering conflates: *agency*, how far one agent proceeds without human intervention, and *orchestration*, how many agents run at once and who coordinates them [39, 56]. The highest agency level on his scale is goal-driven autonomy, in which the agent iterates until a measurable stopping condition holds (the two levels above it vary orchestration), and a team should raise autonomy only as far as it can check the result at acceptable cost, making verification cost the bound on delegation [39].

O4: Claimed benefits are large but are based on self-reports and anecdotes. The reported use cases are plausible and concrete: flaky-test stabilization, nightly end-to-end test repair, and incremental migrations run from cron jobs [37]. The boldest claims, however, are self-reported: In a podcast interview, Stripe engineer Steve Kaliski describes the company's internal "*Minions*" system as producing about 1,300 agent-written pull requests per week [52], and Osmani reports that a loop pointed at Firefox produced 423 security fixes in one month [41]. No published study or dataset we know of supports these numbers, and HuaShu's synthesis note states that other numbers circulating in the discourse "*are mostly secondhand summaries and should be treated as rough reference*" [20].

O5: The discourse comprehensively documents failure modes. Reported and cataloged failures include runaway costs (one experience report describes roughly eight million tokens spent in 48 hours by an over-eager CI-fixing loop), infinite fix loops, verifiers that approve without real checks ("*verifier theater*"), notification fatigue, and drift between a loop's committed instructions and its state [18], as well as plain disappointment from developers who tried loops and found they did not help [37]. Two risks were described under the labels *comprehension debt* (the gap between what exists in the repository and what the developer understands grows with loop velocity) and *cognitive surrender* (using loops to avoid thinking rather than to move faster on understood work) [17, 40]. Review capacity is the practical bottleneck. Commenters in Orosz's thread observe that nightly agent runs can open more pull requests than teams can review, so the volume itself defeats the "*reviewed by devs*" guardrail the loop depends on. Review fatigue turns the human gate into a "*rubber stamp*" while "*the pipeline still reports green*", and "*the loops that stick are the ones where somebody was already paid to read the output*" [38].

O6: Systematic evidence on the value claims is missing. The discourse offers a stable design vocabulary and diverging assessments of value, but systematic evidence for either side is missing. This is the gap our studies target: measuring actual adoption in open-source projects (**RQ2**) and testing outcome claims under controlled conditions (**RQ3**). Together, O1–O6 answer **RQ1**: The discourse defines loop engineering by handing the next prompt to a designed system one layer above the harness, agrees on its building blocks, and disputes its novelty and value.

4 Empirically Studying Loop Engineering

This section reports the mining study that addresses **RQ2** (Section 4.1) and the design of the planned controlled experiment to answer **RQ3** (Section 4.2).

4.1 Mining Loop Engineering Traces (RQ2)

RQ2 asks whether software projects adopt loop engineering at all. Our mining study extends the method of our previous harness configuration study [11]. On 19 August 2026, we scanned the complete sample: the 36,710 repositories classified as engineered software projects [3, 10]. The sample is defined without reference to loop engineering, so it can serve as a denominator for adoption.

We scanned every repository in this frame for committed loop-related *artifacts* using file- and content-based heuristics, and extracted loop *activity* from the Git history of the candidate files. All rules are collected in one catalog, and every result file records the catalog's hash. Artifact detection is deterministic and involves no LLM. Because the dataset's March 2026 snapshot predates the June 2026 discourse, we scanned each repository at its current head and dated each detected artifact by its introducing commit. Between the dataset's March 2026 snapshot and our August 2026 scan, 65 of the 36,710 repositories (0.18%) had been removed from GitHub. We excluded 158 forks (Section 6). We focused on eight tools: the dataset's five (Claude Code, GitHub Copilot, Cursor, OpenAI Codex, and Gemini CLI, selected by developer-survey frequency [10, 46]) plus three open-source tools with native loop primitives (OpenCode, Hermes, and OpenClaw).

Because most loop building blocks can also occur as ordinary outer-harness configuration, each heuristic applies a loop-specific inclusion criterion: A trigger artifact (a GitHub Actions workflow, or a committed cron entry, systemd unit, or shell script) counts only when a schedule or event starts an agent from it, a subagent only when a loop artifact references it as a verifier, a skill only when its name or content marks it as part of a loop, and a state file only when its content carries loop structure, such as a run-log entry or a last-run stamp. We also exclude GitHub Actions workflows gated on a human mention or label (the `@claude` assistant template), since

Table 2: Loop engineering mechanisms and their traceability from GitHub data. The first eight rows are the building blocks named in the practitioner sources (Section 3); the last two are traces of a loop's operation. 'Detected': number of repositories with matches (n = 36,645); 'TP' (true positives): number of repositories confirmed by manual inspection.

Mechanism	Role in the Loop	Example Repository Artifacts	Traceability from GitHub Data	Detected	TP
Triggering & scheduling	Starts runs on a cadence or event	`.github/workflows/*.yml` with `cron` schedules invoking agent CLIs	CI schedules visible; tool-native schedulers (`/loop`) and local cron jobs leave no trace	253	242
Goal & stop conditions	Specifies when a run is complete, in a form a machine can check	Goal definitions in skills, commands, or workflow scripts	Visible only when encoded in committed prompts or scripts	0	0
State & memory	Persists findings and progress across runs	`STATE.md`, `LOOP.md`, plan/progress files, run logs	File presence, plus Git history revealing cadence, authorship, and append-only updates	2	0
Skills & intent	Encodes durable project knowledge read on every run	`.claude/skills/`, `SKILL.md`	Presence detectable; runtime usage invisible [12]	1	1
Parallel isolation	Lets concurrent agents work without collisions	Git worktrees (local); branch naming conventions	Worktrees are not pushed; agent branches indicate general agent use, not loop isolation	—	—
Verification ("*maker/checker*")	Checks the stop condition independently of the implementer; can reject	Verifier subagent definitions (`.claude/agents/`), CI gates, protected branches	Definitions and CI results visible; agent-side verdicts invisible unless logged	0	0
Human oversight & escalation	Gates risky actions; receives escalations	Pull-request reviews, `CODEOWNERS`, escalation issues, gate rules in loop docs	Review traces and documented gates visible; interactive supervision invisible	15	13
Budgets & pause controls	Caps spend per run; restricts what an unattended run may change	`loop-budget.md`, `loop-constraints.md`, pause labels	Declarations visible when committed; enforcement invisible	0	0
Outcomes & provenance	Identifies what the loop produced	Bot-authored commits/PRs, `Co-Authored-By` trailers	Attribution conventions inconsistent across tools and teams [42, 45]	30	—
Cost	Tokens and money actually consumed	Run logs with token estimates	Rarely committed; observed logs hold template values, not measurements [18]	0	0

these are interactive sessions rather than loops. Three triggers override the gate, a schedule, a completed workflow run, and a repository dispatch, because none of them is a person asking. Table 2 maps which mechanisms can leave evidence that agent runs recur.

Because dedicated loop artifacts are rare, we complement the artifact scan with behavioral signals derived from the Git histories of context files (e.g., `AGENTS.md`, `CLAUDE.md`) and candidate state files (e.g., `STATE.md`, `LOOP.md`, plan and progress files), since loops that persist state must revise such files regularly. Over each file's recent commits we tested for three signals: *cadence* (a regular inter-commit interval between ten minutes and seven days), *authorship* (a strict majority of commits attributable to a qualifying tool), and *append-only* (append-shaped edits adding dated run-log entries). The rules and their thresholds are heuristics. We drafted them from the community reference repository, verified the invocation patterns against a 100-file GitHub code-search sample of real agent-invoking GitHub Actions workflows and scripts, and revised rules whose recorded matches an audit showed were not the mechanism the rule claimed. Estimating false-positive rates in advance would have required repositories known *not* to run loops, and no such set exists. We therefore inspected every match. Our supplementary material records the reason for each threshold's value and the per-file measurements the signals are derived from. Because maintenance bots produce similar commit patterns and humans also edit the context and state files [31], we exclude twelve maintenance bots and classify a repository as loop-adopting only when the *authorship* signal co-occurs with one of the other two signals on the same file and is linked to a detected loop artifact.

Before reporting prevalence, we manually verified every repository our heuristics matched: 256 candidates, each repository a rule matched plus one qualifying on the behavioral signal alone. We archived each candidate's evidence, including its GitHub Actions run history, so the labels are based on a fixed snapshot rather than on the platform's mutable state. For each of the 256, Claude Code (Fable 5) annotated this evidence against a committed codebook, labeling the repository *yes*, *no*, or *unclear* on whether it runs an autonomous agent loop. An adversarial agent on a second model (Claude Opus 5) then attempted to refute each label, with disagreement downgrading it to *unclear*, and the last author adjudicated every label as the final authority, an LLM-as-annotator design we disclose per the community guidelines [2]. Where that snapshot, which holds each workflow's 30 most recent runs, could not decide a label, the adjudication drew on deeper evidence fetched one day after the sweep and archived with it (complete run lists, per-run job and step records, and targeted probes). This evidence overturned some labels the snapshot supported, and we clarified the codebook: A successful GitHub Actions workflow run counts as an agent execution only together with a step duration long enough for real work or with produced output. We retained 196 of the 256 annotator labels. Most of the 60 changes flip *unclear* to *yes* on the deeper evidence. Because the last author saw the annotator's labels rather than coding blind, this is a label-retention rate, not inter-rater reliability, so we report no agreement statistic such as Cohen's κ. During adjudication, the last author also judged every individual detection: 325 of the 341 detections we recorded were the mechanism their rule claims.

The scan results, summarized in the rightmost column of Table 2, divide along a clear line: A loop's configuration leaves committed traces, but its runtime state does not. Committed trigger artifacts are the common case. In 253 of the 36,645 scanned repositories a schedule or event trigger invokes a qualifying agent (35 scheduled only, 205 event-triggered only, 13 both), and all but one of the 323 trigger occurrences we recorded are GitHub Actions workflows, the exception being a committed shell script. Parallel isolation leaves no reliable trace: Worktrees are local to a developer's machine, and the 3,507 repositories with two or more branches under agent name prefixes (e.g., `claude/`) reflect ordinary coding-agent use rather than a loop isolating parallel work [42]. The committed state the discourse prescribes is absent. Across all 36,645 repositories, the scan found only two state files meeting the state rule's content criterion, one skill marked as part of a loop, no stop condition, no

budget file, no verifier subagent referenced by a loop artifact, and no cost log with measured values. Inspection rejected both state files as homonyms of unrelated "*loop*" vocabulary. The skill detection is genuine (an issue-triage skill written for a loop), but no loop in its repository runs it. The empty goal row reflects the rule's fixed phrase list: None of the stop-condition phrases it accepts occurs in the committed artifacts, yet the annotation recorded a goal or stop-condition mechanism somewhere in 55 of the 256 repositories we inspected, several as terminal states or retry caps written into committed prompts. The behavioral signals show no committed loop state either. The candidate file names matched 1,334 files with the required history of at least five commits, in 637 repositories. No file shows the append-only signal, and the cadence and authorship signals co-occur on only one file, an agent-maintained context file in a repository with no detected loop artifact to link to (the verification nonetheless confirmed it as running a loop from its platform records), so the signals alone classify no repository as loop-adopting. Since loops demonstrably run in these repositories, their state is either absent from version control or kept in files our candidate file names do not cover. The signals cannot separate the two. Of the artifact occurrences, 150 were first committed after the dataset's March 2026 snapshot, all but two of them GitHub Actions workflow files.

Manual verification confirmed autonomous agent processes, evidenced by archived execution histories or the changes they committed, in 217 of the 256 repositories we inspected (0.59% of the scanned repositories), against 33 not confirmed and six left unclear. Excluding the six unclear labels, the heuristics' precision is 0.868 (95% Clopper–Pearson CI [0.820, 0.907]). Among the 217 confirmed loops, 21 run on a schedule only (e.g., `dimagi/commcare-android`'s daily workflow, with 276 scheduled runs in the archived history), 180 run on repository events only (mostly automatic pull-request review), 15 run on both, and one operated until its GitHub Actions workflow broke. In 12 of the 217, every trigger workflow we detected is marked manually disabled in the archived snapshot. These and the broken one count as confirmed because they demonstrably ran, even if no longer at inspection time. Successful runs alone are weak evidence of agent execution: `AI-Hypercomputer/xpk`'s hourly Gemini triage logged 6,290 runs without a single one, because its gating issue search matches nothing. Two observations are reported descriptively because no committed artifact carries them. In 95 of the 159 repositories where we measured review timing, GitHub Copilot reviews arrive within minutes of a pull request opening, clustered on individual authors' pull requests, the pattern of a personal standing auto-request (an account-level setting invisible to the repository, which never counts toward adoption here). And no inspected repository declares an autonomy level for its loops.

Claude Code dominates the confirmed tool attributions (189 of 217), followed by OpenAI Codex (11), OpenCode (6), Gemini CLI (5), Cursor (4), and Copilot (2). Agents do co-maintain their own configuration: The outcomes and provenance row counts the 30 repositories where an agent-majority context file co-occurs with a committed loop artifact (without that conjunction the measure returns 384 files in 217 repositories), but co-maintained configuration alone is not evidence that a loop produced anything. Because our pipeline is deterministic and versioned, re-running it on later snapshots can track how adoption evolves.

4.2 Effects of Loop Autonomy (RQ3)

The discourse claims that handing off more of the loop increases developer output. Skeptics claim that it mostly increases token spend and the number of unreviewed changes. We propose to test these claims experimentally.

We plan to conduct a controlled experiment in which the same set of maintenance tasks (e.g., bug fixes, dependency updates, test augmentation) is executed on selected repositories, using an agentic AI coding tool. We will compare three conditions that map onto the hand-off sequence Osmani described [41] and onto academic autonomy taxonomies [9]: (C1) *interactive* agent use, where a developer prompts each step; (C2) *goal-driven* runs, where the developer starts each run but hands off the stop condition, defined in a form a machine can check, and reviews only the result; (C3) *scheduled* loops, where runs recur without per-run initiation and the developer only handles escalations and reviews. Only C3 is a loop under our working definition. Comparing C1 with C2 measures the effect of handing off the stop condition; comparing C2 with C3, the effect of recurrence. We exclude the proactive step, in which the agent generates its own tasks, because it makes it hard to establish a shared task set on which to compare conditions. All three conditions run the same agent on the same task set, so the autonomy level is the only independent variable. In the terms of the two dimensions Osmani separates (see O3), orchestration is the same in every condition: one agent works on one task. Agency increases from C1 to C2, where the agent decides when a run is done; C3 adds recurrence rather than more agency [39]. Our effort measures capture the cost of checking a run in each condition, a cost Osmani treats as the bound on autonomy but does not quantify.

To limit the number of human participants, we divide the experiment into three parts: an *agent-only benchmark*, a *simulated interactive condition*, and *trace collection* from real developers. The *agent-only benchmark* compares C2 and C3. Their outcomes are machine-measurable, so the unit of analysis is one agent run on a task, and repeated stochastic runs across many repositories provide statistical power. It uses tasks whose success is decided by automated tests, as in SWE-bench [23]. Because recent audits found substantial defects in SWE-bench Verified and SWE-Bench Pro [35, 36], we will read each task's statement, tests, and reference solution before adopting it and discard any showing one of the four flaws those audits identified. Condition C1 requires a human who corrects the agent, approves its work, and answers its escalations. Rather than recruit a developer for every run, the *simulated interactive condition* simulates this role, seeded from existing datasets of real agent sessions that record such interventions at scale [50]: Deterministic rules extracted from the traces decide when the simulated developer intervenes, and an LLM generates the content of each intervention.

Trace collection covers what those datasets do not record: complete interaction traces—prompts, corrections, rejections, and escalation responses, each with a timestamp and a snapshot of the repository—from a small group of professional developers. We will recruit them from our industry network, from students with substantial professional experience as developers, from contributors to repositories our mining study identified, and from crowdsourcing platforms screening for such experience. These sessions work on open backlog items from the selected repositories; the items

stay outside the three-condition comparison. Before running C1 at scale, we will validate the simulated developer on held-out human sessions (it must produce the same outcome distribution, within agreement bounds fixed in advance). If it fails, a sample of developers will operate the agent directly at smaller scale. The benchmark comparison of C2 and C3 does not use the simulated developer either way. The simulated developer only estimates what a loop produces under human-like steering; conclusions about developers themselves come from those who provide traces.

We choose outcome measures to test the claims and counter-claims of Section 3: task throughput and correctness (e.g., tests passed, regressions, review rejections), cost (tokens and wall-clock time), and human effort (active interaction time from input timestamps, escalations handled, review rounds, and the fraction of agent-produced diffs developers modify before merging). We will follow established guidelines for experimentation [54] and for empirical studies involving LLMs [2], pin the model and harness versions, and release all configurations.

5 Discussion

New discipline or new label? Our review supports parts of both positions. Skeptics are right that schedulers, event triggers, and maintenance bots predate the term *loop engineering*, and that autonomic computing described self-managing control loops decades ago [24]. The new element is not the loop mechanics but their adoption as a developer practice: Developers place a non-deterministic worker behind natural-language goal conditions and verification-governed termination in their own repositories, which moves the engineering effort from implementing the automation to bounding it with guardrails, budgets, verifiers, and escalation policies [26, 34]. Our mining study adds evidence on both sides: Loops run in practice, but without committing the state files that the community reference repository prescribes. Of the 217 confirmed loops, 180 ran on repository events only, most of them reviewing each newly opened pull request. Such a run reads the pull request, posts its review, and finishes. The next run starts from the next pull request—there is no state to commit. The scheduled loops we confirmed were mostly issue triage. A triage run starts from whatever the issue tracker holds at that moment, so the tracker, not a committed file, carries the backlog between runs. The practitioner sources we reviewed prescribe committed state files to persist findings and progress across runs. In most of the loops we confirmed, there was either nothing to persist or the issue tracker already held the state.

Macedo's corpus of published loop designs reaches a complementary conclusion from the opposite sampling logic: Even among exemplary loops, automated triggering and durable memory are the least developed elements [29]. On stop conditions the numbers diverge: 74% of the published designs name terminal states, while our fixed phrase list matched nothing. The divergence is not a contradiction. Published exemplars spell their stop conditions out; ordinary projects that commit one phrase it in their own words (Section 4.1). Explicit loop design may be transitional if tools absorb today's explicit commands into harness features, as the discourse itself suggests [37], but the design problem it names, bounding a non-deterministic worker with verification, budgets, and escalation, does not disappear when they do.

Implications for research. If loops become a common operating mode, the unit for evaluating coding agents shifts from a single run on a bounded task, as in current benchmarks [23], to a loop operating over time, where convergence, cumulative cost, drift, and the quality of escalations matter. This extends the open problem of harness-level evaluation [34] and makes loop observability a research topic in its own right. Our scan makes the observability gap concrete: The loops we confirmed were visible only through committed triggers and platform execution records, their state was not version-controlled, and some agent activity has no committed artifact at all (GitHub's dynamic Copilot workflows, the personal review auto-request of Section 4.1, agents driven through `/loop` or `/goal`, private repositories, and platform-native schedulers). Repository mining therefore observes the configuration side of loop engineering while under-estimating its runtime side, as the traceability column of Table 2 records. Our planned autonomy experiment (**RQ3**) complements it by observing that runtime side directly.

Implications for practice. The most consistent advice across sources is for developers to design an independent verification step for the agent's work before granting autonomy and to adopt a loop in stages from report-only to unattended operation. That advice is intuitively plausible but, to our knowledge, untested. Our planned autonomy experiment (Section 4.2) measures task outcomes and developer effort as autonomy increases from interactive use to scheduled loops. If loops become common, the developer's work shifts to setting goal conditions, designing that verification step, and defining budgets and escalation; software engineering education would then need to teach these skills, as it teaches testing and code review. Ng argues that developers' "*context advantage*" over the agent keeps the developer feedback loop from being automated [33]. Judging what a loop should be pointed at is then itself a skill to teach.

GitHub's run history records each run's outcome, success or failure. Only the job and step records show whether the agent executed. Teams running a loop can make idle runs visible in the run history itself: report "*nothing to do*" instead of success when a run finds no work, log how long the agent step ran, and log which artifact each run created (e.g., a pull request, comment, or commit).

6 Threats to Validity

Construct validity. Sources disagree on what counts as a loop, and the term collides with unrelated uses, so our working definition may not match that of others. Our mining study operationalizes loop engineering through specific artifacts and traces. The inclusion criteria and manual verification (Section 4.1) reduce but do not remove the risk that these indicate ordinary automation. The line between an agent-invoking maintenance workflow and an engineered loop remains a judgment call; our codebook states the criteria we applied.

Our heuristics started from a single community reference repository and were iteratively refined against real workflows (Section 4.1); still, a loop that follows conventions they do not cover is not detected. In the other direction, 16 of the 341 detections were false positives; the 'TP' column of Table 2 reports for each mechanism how many detected repositories the inspection confirmed.

To keep the captured data at a manageable size, our pipeline saved each repository's matched files under per-repository and per-file size limits. These limits dropped files in 26 repositories before

they could be checked against our rules, so detections there may be missing. The supplementary material lists every dropped file.

Our authorship heuristic attributes a commit made by GitHub's CI account (`github-actions[bot]`, the identity under which GitHub Actions workflows commit) to a qualifying tool if the repository carries a detected agent-invoking workflow. This link is repository-level, not per workflow, so commits from unrelated workflows can be misattributed.

Internal validity. The 256 repository labels (Section 4.1) were annotated by an LLM and adjudicated by one author, not independently double-coded. The labels are based on each repository's archived GitHub Actions run history and on file content at the pinned head commit. During adjudication, the last author overturned labels that the run outcomes alone had supported. The archived job- and step-level records showed that in several runs marked successful, some ending within seconds, the agent step never executed. A successful run alone is therefore weak evidence of agent execution.

Because we inspected every matched repository, our heuristics' reported precision is an exact proportion; the 95% Clopper–Pearson interval reported alongside (Section 4.1) estimates the precision of a re-run on later and larger snapshots.

Our exploratory literature review (Section 3) is interpretive, not exhaustive: The last author selected the sources by following the discourse, so relevant material may be missing and another reviewer might weight the sources differently.

External validity. The prevalence we report describes our repository sample, not open-source software at large. The sample inherits the original dataset's composition—ten languages, activity filters, and an LLM-based engineered-project classification [10]—and its repository list was fixed in March 2026. We scanned the repositories about ten weeks after loop engineering was coined, so the practice had little time to spread. We excluded 158 forks, identified by a head commit shared with an earlier-created repository, from all detection and behavioral counts; diverged forks escape this criterion, so some duplicates may remain. Our counts cover the eight qualifying tools only: Loops calling an LLM API directly are not counted. Commits made by three of the eight tools carry no distinctive author name, email, or trailer, so the behavioral signals cover only the remaining five tools.

Our scan observed each repository once on 19 August 2026. The Git histories and archived GitHub Actions runs we read cover the time up to that date. A loop whose committed artifacts were removed before the scan is therefore invisible, while a disabled loop whose artifacts remain is still detected—one confirmed loop ran at least 27 times before its maintainers disabled it. Our behavioral heuristics require at least five commits per candidate file, so loops adopted in the weeks before the scan cannot meet that threshold. We also observe only what GitHub exposes: Platform-native and off-platform schedulers leave no committed execution record, and dynamically created workflows (e.g., Copilot's coding agent runs) leave execution records but no committed workflow file, so our scan does not count them.

The reviewed discourse is weeks old, dominated by a few highly visible voices, and possibly subject to survivorship effects. Practitioners who tried loops and then abandoned them may not post about it. It is also unsettled—ten days after the retrospective we cite, Osmani proposed a different account of agent autonomy [39]—and parts of it may be machine-generated: Orosz reports that his practitioner survey attracted hype replies from apparent bot accounts [37]. Our mining evidence comes from public open-source repositories only. The high-volume internal deployments described in O4 leave no public trace, so the prevalence we report understates adoption overall and covers industry practice only where it happens in open source.

Data availability. Our supplementary material is openly available on Zenodo [28]. It contains the collected source copies and our extraction and consolidation notes for the review, and for the mining study the scan scripts, heuristic catalog, result data, annotation protocol and codebook with adjudication record, per-repository evidence files, and archived GitHub Actions run histories.

7 Conclusion

Within weeks, loop engineering went from being coined in a blog post to a contested term for a practice that tool vendors had already begun to support. Our review of the gray literature found a stable vocabulary paired with an almost complete absence of evidence on the practice's claims of value. Our research agenda starts where Galster et al.'s harness configuration study ended [12]: measuring which parts of the loop leave traces in open-source repositories, and testing under controlled conditions whether handing off more of the loop improves outcomes. Our mining study gives that agenda a first data point: Autonomous agent loops already run in open-source projects, mostly for pull request review and scheduled issue triage. However, their state is not captured in committed files. For the journal extension, we will re-run the mining study on later and larger snapshots (**RQ2**) and run the autonomy experiment (**RQ3**). Loop observability and the effect of loops on program comprehension remain open directions.

The discourse has already named a layer above the loop: On 17 July 2026, Steinberger asked whether the discussion was still about loops or had shifted to graphs [47]; within hours, Hamel Husain published "*Loop Engineering Is Dead. Enter Graph Engineering*" [22], and Lindenberg reads the new term as loops wired into a structure, agents as nodes, with work, state, and routing decisions along the edges [25]. We treat graph engineering as an outlook on what to study once individual loops are understood, not as an extension of our agenda: Our mining study and our planned autonomy experiment address the single loop, and a result measured on a graph cannot be attributed to the individual loops or to the connections between them before **RQ2** and **RQ3** are answered. Only forty days separate the coining of loop engineering from Steinberger's question—less time than either of our studies takes to run. Waiting for the terminology to settle would mean not studying the practice while it is forming. Our heuristics detect scheduled, verified, state-carrying agent runs whatever practitioners call them, so our agenda holds whether the next label is graph engineering or something else.

References

[1] Agentic AI Foundation. 2026. AGENTS.md. Website. https://agents.md/ Stewarded by the Agentic AI Foundation under the Linux Foundation, accessed 2026-07-18.

[2] Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. 2026. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. arXiv:2508.15503 doi:10.48550/arXiv.2508.15503 Accepted for publication in Empirical Software Engineering.

[3] Sebastian Baltes, Seyedmoein Mohsenimofidi, Levi Böhme, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Matthias Galster. 2026. A Dataset of Agentic AI Coding Tool Configurations. doi:10.5281/zenodo.19375880

[4] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. arXiv:2507.09089 doi:10.48550/arXiv.2507.09089

[5] Birgitta Böckeler. 2026. Harness Engineering for Coding Agent Users. Blog post. https://martinfowler.com/articles/harness-engineering.html Published 2026-04-02, accessed 2026-07-17.

[6] Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida. 2025. Agent READMEs: An Empirical Study of Context Files for Agentic Coding. arXiv:2511.12884 [cs.SE] doi:10.48550/arXiv.2511.12884

[7] Boris Cherny. 2026. I'm Boris and I created Claude Code. X post. https://x.com/bcherny/status/2007179832300581177 Published 2026-01-02, accessed 2026-07-18.

[8] Peter Cihon, Merlin Stein, Gagan Bansal, Sam Manning, and Kevin Xu. 2025. Measuring AI Agent Autonomy: Towards a Scalable Approach with Code Inspection. arXiv:2502.15212 doi:10.48550/arXiv.2502.15212

[9] K. J. Kevin Feng, David W. McDonald, and Amy X. Zhang. 2025. Levels of Autonomy for AI Agents. arXiv:2506.12469 doi:10.48550/arXiv.2506.12469

[10] Matthias Galster, Seyedmoein Mohsenimofidi, Levi Böhme, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. 2026. A Dataset of Agentic AI Coding Tool Configurations. arXiv:2605.08435 [cs.SE] doi:10.48550/arXiv.2605.08435 Dataset deposit archived on Zenodo, DOI 10.5281/zenodo.19375880.

[11] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. 2026. Configuring Agentic AI Coding Tools: An Exploratory Study. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26)*. ACM, Montreal, QC, Canada, 11–20. doi:10.1145/3805760.3814887

[12] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. 2026. Harness Engineering for Agentic AI Coding Tools: An Exploratory Study. arXiv:2602.14690 [cs.SE] doi:10.48550/arXiv.2602.14690 Extended version of the AIware '26 paper "Configuring Agentic AI Coding Tools: An Exploratory Study".

[13] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121. doi:10.1016/j.infsof.2018.09.006

[14] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, et al. 2025. A Survey of Vibe Coding with Large Language Models. arXiv:2510.12399 doi:10.48550/arXiv.2510.12399

[15] Jiayi Geng and Graham Neubig. 2026. Effective Strategies for Asynchronous Software Engineering Agents. arXiv:2603.21489 doi:10.48550/arXiv.2603.21489

[16] Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. 2026. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? arXiv:2602.11988 [cs.SE] doi:10.48550/arXiv.2602.11988

[17] Cobus Greyling. 2026. Loop Engineering. Substack essay. https://cobusgreyling.substack.com/p/loop-engineering Published 2026-06-09, accessed 2026-07-18.

[18] Cobus Greyling. 2026. loop-engineering: Design systems that prompt your agents. GitHub repository. https://github.com/cobusgreyling/loop-engineering Accessed 2026-07-15.

[19] Qishuo Hua, Lyumanshan Ye, Dayuan Fu, Yang Xiao, Xiaojie Cai, Yunze Wu, Jifan Lin, Junfei Wang, and Pengfei Liu. 2025. Context Engineering 2.0: The Context of Context Engineering. arXiv:2510.26493 doi:10.48550/arXiv.2510.26493

[20] HuaShu. 2026. Loop Engineering: The Anthropic Playbook for Designing Systems That Prompt Your Agents. Self-published working note (Orange Books, v260615). https://huasheng.ai/orange-books Independent conference-style reformatting of the open guide "Loop Engineering: Stop Asking Me What It Is"; not peer-reviewed. Accessed 2026-07-15.

[21] Geoffrey Huntley. 2025. Ralph Wiggum as a "software engineer". Blog post. https://ghuntley.com/ralph/ Published 2025-07-14, accessed 2026-07-18.

[22] Hamel Husain. 2026. Loop Engineering Is Dead. Enter Graph Engineering. X article. https://x.com/HamelHusain/article/2078346425621237935 Published 2026-07-18, accessed 2026-08-21.

[23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024*. OpenReview.net, Vienna, Austria, 51 pages. https://openreview.net/forum?id=VTF8yNQM66 OpenReview proceedings are unpaginated; numpages from the camera-ready PDF.

[24] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50. doi:10.1109/MC.2003.1160055

[25] André Lindenberg. 2026. From Loops to Graphs. LinkedIn newsletter article, Artificial Engineering #65. https://www.linkedin.com/pulse/from-loops-graphs-andr%C3%A9-lindenberg-m8jae Published 2026-07-25, accessed 2026-07-28.

[26] André Lindenberg. 2026. From Prompts to Loops: The Next Step of Agent Engineering. LinkedIn newsletter article, Artificial Engineering #61. https://www.linkedin.com/pulse/rom-prompts-loops-next-step-agent-engineering-andr%C3%A9-lindenberg-wnqre/ Published 2026-06-28, accessed 2026-07-15.

[27] Jai Lal Lulla, Seyedmoein Mohsenimofidi, Matthias Galster, Jie M. Zhang, Sebastian Baltes, and Christoph Treude. 2026. On the Impact of AGENTS.md Files on the Efficiency of AI Coding Agents. arXiv:2601.20404 [cs.SE] doi:10.48550/arXiv.2601.20404 To appear at the 1st Journal Ahead Workshop (JAWs@ICSE 2026).

[28] Jai Lal Lulla, Vahram Nersesyan, Seyedmoein Mohsenimofidi, Christoph Treude, and Sebastian Baltes. 2026. *Loop Engineering: Building Blocks, Adoption, and Impact (Supplementary Material)*. Zenodo. doi:10.5281/zenodo.21540692

[29] Sandeco Macedo. 2026. Stop Hand-Holding Your Coding Agent: Engineering the Loops that Replace Step-by-Step Prompting. arXiv:2607.00038 [cs.SE] doi:10.48550/arXiv.2607.00038 Preprint, submitted 28 June 2026.

[30] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. 2025. A Survey of Context Engineering for Large Language Models. arXiv:2507.13334 [cs.CL] doi:10.48550/arXiv.2507.13334

[31] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. 2025. Context Engineering for AI Agents in Open-Source Software. arXiv:2510.21413 [cs.SE] doi:10.48550/arXiv.2510.21413 To appear at the 23rd IEEE/ACM International Conference on Mining Software Repositories (MSR 2026), Rio de Janeiro, Brazil.

[32] Andrew Ng. 2024. Robots Talk Back, AI Security Risks, Political Deepfakes, and more. The Batch, Issue 241, DeepLearning.AI. https://www.deeplearning.ai/the-batch/issue-241 Published 2024-03-20, accessed 2026-07-28.

[33] Andrew Ng. 2026. Loop Engineering: Three Key Loops for Building Great Software. The Batch, DeepLearning.AI. https://www.deeplearning.ai/the-batch/three-key-loops-for-building-great-software Published 2026-06-26, accessed 2026-07-28.

[34] Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, et al. 2026. Code as Agent Harness: Toward Executable, Verifiable, and Stateful Agent Systems. arXiv:2605.18747 [cs.CL] doi:10.48550/arXiv.2605.18747

[35] OpenAI. 2026. Separating signal from noise in coding evaluations. Blog post. https://openai.com/index/separating-signal-from-noise-coding-evaluations/ Published 2026-07-08, accessed 2026-07-20.

[36] OpenAI. 2026. Why SWE-bench Verified no longer measures frontier coding capabilities. Blog post. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/ Published 2026-02-23, accessed 2026-07-20.

[37] Gergely Orosz. 2026. What is "loop engineering?". The Pragmatic Engineer newsletter. https://newsletter.pragmaticengineer.com/p/what-is-loop-engineering Published 2026-07-14, accessed 2026-07-18.

[38] Gergely Orosz. 2026. What is "loop engineering" to you, anyway? LinkedIn post. https://www.linkedin.com/posts/gergelyorosz_what-is-loop-engineering-to-you-anyway-share-7482371640924737536-tmm8/ Published 2026-07-13, accessed 2026-07-15. The archived capture shows no absolute date; the post ID decodes to 2026-07-13 09:53 UTC. The same decoding reproduces the recorded dates of both Osmani LinkedIn posts exactly, and the post solicits the replies the 14 July newsletter reports on.

[39] Addy Osmani. 2026. AI coding? The autonomy you grant an agent is the level you can cheaply verify. LinkedIn post with slide deck. https://www.linkedin.com/posts/addyosmani_agentic-autonomy-levels-ugcPost-7486670405957505024-ZjLa/ Published 2026-07-25, accessed 2026-07-26.

[40] Addy Osmani. 2026. Loop Engineering. Blog post. https://addyosmani.com/blog/loop-engineering/ Published 2026-06-07, accessed 2026-07-15.

[41] Addy Osmani. 2026. Loop engineering, one month on. LinkedIn post. https://www.linkedin.com/posts/addyosmani_loop-engineering-a-month-on-ugcPost-7483004445094506496-R62H/ Published 2026-07-15, accessed 2026-07-15.

[42] Romain Robbes, Théo Matricon, Thomas Degueule, André C. Hora, and Stefano Zacchiroli. 2026. Agentic Much? Adoption of Coding Agents on GitHub. arXiv:2601.18341 [cs.SE] doi:10.48550/arXiv.2601.18341

[43] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2025. Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI. arXiv:2505.19443 doi:10.48550/arXiv.2505.19443

[44] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, et al. 2024. The Prompt Report: A Systematic Survey of Prompt Engineering Techniques. arXiv:2406.06608 doi:10.48550/arXiv.2406.06608

[45] Mohammed Latif Siddiq, Xinye Zhao, Vinicius Carvalho Lopes, Beatrice Casey, and Joanna C. S. Santos. 2026. Security in the Age of AI Teammates: An Empirical Study of Agentic Pull Requests on GitHub. arXiv:2601.00477 doi:10.48550/arXiv.2601.00477

[46] Stack Exchange Inc. 2025. Stack Overflow Developer Survey 2025: AI Agent out-of-the-box tools. https://survey.stackoverflow.co/2025/ai/#3-ai-agent-out-of-the-box-tools. Accessed 2026-07-21.

[47] Peter Steinberger. 2026. Are we still talking loops or did we shift to graphs yet? X post. https://x.com/steipete/status/2078277297791189132 Published 2026-07-17, accessed 2026-07-28.

[48] Peter Steinberger. 2026. Here's your monthly reminder that you shouldn't be prompting coding agents anymore. You should be designing loops that prompt your agents. X post. https://x.com/steipete/status/2063697162748260627 Published 2026-06-07, accessed 2026-07-15.

[49] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Santana de Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2025. Prompting in the Wild: An Empirical Study of Prompt Evolution in Software Repositories. In *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR 2025*. IEEE, Ottawa, ON, Canada, 686–698. doi:10.1109/MSR66628.2025.00106

[50] Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. 2026. How Coding Agents Fail Their Users: A Large-Scale Analysis of Developer-Agent Misalignment in 20,574 Real-World Sessions. arXiv:2605.29442 [cs.SE] doi:10.48550/arXiv.2605.29442

[51] Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Méndez. 2025. Prompts as Software Engineering Artifacts: A Research Agenda and Preliminary Findings. In *Product-Focused Software Process Improvement - 26th International Conference, PROFES 2025, Salerno, Italy, December 1-3, 2025, Proceedings (Lecture Notes in Computer Science)*. Springer, Cham, 470–478. doi:10.1007/978-3-032-12089-2_32

[52] Claire Vo. 2026. How Stripe built "minions"—AI coding agents that ship 1,300 PRs weekly from Slack reactions. Podcast episode, How I AI. https://podcasts.apple.com/us/podcast/how-stripe-built-minions-ai-coding-agents-that-ship/id1809663079?i=1000757255000 Interview with Steve Kaliski (Stripe). Published 2026-03-25, accessed 2026-07-18.

[53] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE] doi:10.48550/arXiv.2302.11382

[54] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-29044-2

[55] WorkOS. 2026. Boris Cherny: Claude Code & the Future of Engineering. Video interview, Acquired Unplugged, WorkOS YouTube channel. https://www.youtube.com/watch?v=RkQQ7WEor7w Interview with Boris Cherny (Anthropic). Published 2026-06-02, accessed 2026-08-21.

[56] Steve Yegge. 2026. Welcome to Gas Town. Blog post on Medium. https://steve-yegge.medium.com/welcome-to-gas-town-4f25ee16dd04 Published 2026-01-01, accessed 2026-07-27.