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 guide agentic AI coding tools has moved up several levels of abstraction, from phrasing prompts to engineering context to configuring the harness around the underlying model. In June 2026, practitioners began to describe a new level called *loop engineering*: Instead of prompting an agent interactively, developers design systems that prompt agents for them. The term spread rapidly, accompanied by bold claims and vocal skepticism, but its adoption in software projects has not yet been measured. We present (1) an exploratory review of the emerging gray literature on loop engineering, (2) a traceability analysis and an exploratory mining study of 36,710 software repositories, and (3) the design of a controlled experiment to study how loop goals, static analysis checks, and decision policies affect task success, code quality, resource usage, and token consumption of agent loops. Our literature review found broad agreement on what a well-engineered loop contains: agent loops started on a schedule or by repository events and bounded by machine-checkable stop conditions, persistent state files, verifier subagents, token budgets, and defined points of escalation to humans. These sources provide no systematic evidence of the claimed benefits. In the mining study, we found autonomous agent loops in 217 open-source GitHub repositories using heuristics and behavioral signals. Nearly all of these repositories committed the configuration that starts their loops, but almost none of them committed the state files that the gray literature suggests. Our proposed experiment is a first step toward closing the evidence gap identified in our literature review. We call for further empirical studies of the benefits and costs of loop engineering for software engineering tasks.

CCS Concepts

• **Software and its engineering**;

Keywords

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

This work is licensed under a Creative Commons Attribution 4.0 International License.
JAWs@ASE 2026, Munich, Germany
© 2026 Copyright held by the owner/author(s).

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*. ACM, New York, NY, USA, 11 pages.

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 [15]. For most developers, working with these tools still entails typing a request, reading the result, and then typing the next request. However, 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 [10], 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*" [58]. 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*" [51], a post that had reached more than eight million views by mid-July [40]. 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*" [43].

The term *loop engineering* extends the trajectory from prompt engineering via context engineering to harness engineering (see Section 2.1). In fact, the practitioner discourse places loop engineering one abstraction level above harness engineering. Where a harness configures an agent loop, a loop controls repeated agent loops over time. It specifies what starts each agent loop (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 [24, 43].

We argue that this development deserves the attention of empirical software engineering researchers. First, the practice is no longer hypothetical: Between March and May 2026, tool vendors added commands such as `/loop` and `/goal` that developers can use to trigger recurring and goal-driven agent loops [40]. Second, the claims attached to loop engineering rest almost entirely on anecdotes and self-reported productivity numbers, while experienced engineers call loop engineering "*a renamed cron job*" and warn about token costs and reviewer fatigue [40, 41]. Third, the past years have shown that practitioner vocabulary can influence research agendas quickly.

For example, the discourse-driven term *vibe coding* attracted surveys and empirical studies within months [17, 47]. 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 gray literature, and how does it relate to prompt, context, and harness engineering?*

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

RQ3 *How do loop goals, static analysis checks, and decision policies affect task success, code quality, resource usage, and token consumption of agent loops?*

We answer **RQ1** by situating loop engineering in the trajectory from prompt via context and harness engineering (Section 2) and introducing a working definition (Section 2.2), then synthesizing selected gray literature into six observations (Section 3). We address **RQ2** with a traceability analysis and an exploratory mining study of 36,710 engineered software repositories (Section 4.1). To answer **RQ3**, we describe a controlled study of loop goals, static analysis checks, and decision policies (Section 4.2) that we will execute for the planned journal extension.

2 Background

2.1 From Prompts to Context to Harnesses

Prompt engineering was one of the first named practices for guiding large language models (LLMs). Catalogs of reusable prompt patterns [56] and a broad survey of prompting techniques [48] 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 [53, 54]. *Context engineering* generalized this practice from single instructions to the systematic design of everything a model sees in its context window [23, 33]. 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], with mining studies examining how such files are structured and how they evolve [9, 34]. Early evidence on their effect is mixed: One study associated `AGENTS.md` presence with lower runtime and fewer output tokens [30]; another found no gain in task success at over 20% higher inference cost [20].

Harness engineering moves the focus from the context to the harness, i.e., "*the software layers around the model(s) that drive the agent loop*" [15, 37]. The *agent loop* is the model's repeated cycle of calling tools and acting on the results until a stop condition ends it, such as a final response, an achieved goal, or a time limit. *Loop engineering* adds structures that start an agent loop, verify its results, and start it again over time. For example, an agent can run tests and revise a change within one session, while a loop can start new sessions whenever a build fails. 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 coding tools. They found that developers overwhelmingly adopt static context files and rarely use executable mechanisms, concluding that "*harness engineering in open source today is . . . mostly context engineering*" [14, 15]. Their taxonomy contains no mechanism that determines *when and how often* an agent runs. Hooks are a related mechanism but fire at fixed lifecycle points within a session. Therefore, loop recurrence, termination, state across agent loops, budgets, and graduated human oversight are all beyond the scope of that analysis. With our study, we intend to close that gap.

2.2 The Emergence of Loop Engineering

Developers ran agents in loops before the term *loop engineering* existed, at two different levels. The first level is a single execution of an 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 [35]. The second level adds recurrence and state across agent loops, 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`" [25]. 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 [40]. Tool vendors later adopted this 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 2026 [40].

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 [40, 43, 51]. Later that month Ng called it "*a hot buzzphrase*" [36]. Osmani described a loop as "*a recursive goal where you define a purpose and the AI iterates until complete*", with state persisted across agent loops ("*the agent forgets, the repo doesn't*") [43]. 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 execution trigger, and *proactive* loops hand off the prompt itself [44].

Within weeks, an ecosystem formed around these ideas. In a community repository, Cobus Greyling distilled the discourse into seven loop patterns (e.g., 'daily triage' and 'CI sweeper' for continuous integration), four "*readiness levels*" from documented intent (L0) to unattended operation (L3), catalogs of failure modes and anti-patterns, and tooling for scaffolding, auditing, and cost estimation [22]. 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 [29]. In a self-published playbook, HuaShu decomposed one loop turn into five "*moves*" (discovery, handoff, verification, persistence, scheduling) [24]. Ng distinguished the agentic coding loop (minutes) from the developer and external feedback loops around it (hours to weeks) [36].

For each of the four layers of guiding AI coding agents, Table 1 names the unit of concern and example artifacts. Within the harness layer, Böckeler distinguishes what is built into a coding agent (e.g., the system prompt) from what developers configure on top [7]. We call these the *inner* and *outer harness*, and only artifacts of the outer harness can appear in the repository of a project adopting loop engineering.

We have consolidated the practitioner discourse that is outlined in Section 3 into a unifying definition: ***Loop engineering*** *is the practice of designing automated control structures that repeatedly invoke coding agents, triggered on a schedule or by events, with*

Table 1: Four layers of guiding AI coding agents [15, 24, 43]. Each layer subsumes the previous one; the harness engineering 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 loop	Skills, subagents, hooks, MCP config.
Loop eng.	Repeated agent loops	Schedules, stop conditions, state files, run logs, budgets, verifier agents

each agent loop bounded by a machine-checkable stop condition. A well-engineered loop persists state across agent loops, verifies results independently of the implementation agents, bounds inference costs, restricts what an unattended agent loop may change without approval, and defines when humans should intervene.

2.3 Adjacent Academic Work

In a preprint, Macedo defined a "*loop specification*" as a reusable artifact made of a trigger, a goal, a verification step, a stopping rule, and a memory. They hand-coded 50 entries of a public loop catalog along five categories: trigger, goal type, verification level, architecture, and terminal states [32]. Such a study based on a loop catalog can show how published loop designs are composed, but not how many projects run certain loops. Therefore, our mining study complements such studies by estimating how many engineered open-source software projects run loops (Section 4.1). Related to loop engineering, Geng and Neubig studied *asynchronous* software engineering agents that operate without a human in the loop and reported that a design combining dependency-aware plans, isolated workspaces, and test-based verification improved task accuracy [18]. Benchmarks such as SWE-bench evaluate agents on individual software engineering tasks [27]. We, however, study the developer practice of designing recurring control structures in the software projects they maintain. Existing autonomy taxonomies grade how much a developer delegates to an agent: Feng et al. define five autonomy levels based on the user's role, from operator to observer [12]. Cihon et al. measure autonomy from the code that orchestrates the model, which, in our terminology, would be the loop [11]. A randomized controlled trial found experienced open-source developers measurably *slowed down* by early-2025 tools, despite believing the opposite [6]. Whether recurring loops change such results 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 situation for which established review guidelines recommend including such sources [16]. 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 [43] and one-month retrospective [44], Orosz's newsletter investigation [40] and its 136-comment discussion thread [41], Lindenberg's newsletter article [29], a self-published playbook [24], and a snapshot of Greyling's community repository [22]. On 18 July, we added three sources that the collected material referenced: Huntley's post introducing the Ralph technique [25], Greyling's essay accompanying his repository [21], and the podcast interview in which Stripe engineer Steve Kaliski describes the company's internal agent system [55]. On 25 July, after consolidating the observations below, we added one further source, Osmani's post on agent autonomy and a related slide deck [42]. 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 agentic tool (Claude Code with Fable 5) assisted in this search as well as in locating the sources that the collected material referenced. The tool further assisted in extracting candidate quotes, which the last author then checked against the sources. Fable 5 was trained on data up to January 2026, five months before loop engineering was coined [2], so Claude Code worked from the supplied and retrieved sources, not from prior knowledge of the term or its discourse. We disclose this in line with the community guidelines for empirical studies involving LLMs [3]. 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), reviewed by all authors.

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 [24, 29, 43]. The four-part taxonomy Osmani reported (Section 2.2) operationalizes this core as a sequence of hand-offs [44]. In Orosz's thread, one commenter argued that an agent started on a schedule or by an event is "*still largely a workflow*": It becomes a loop only when an explicit feedback signal feeds one iteration's outcome into the next [41].

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*", and the term repackages event-driven architecture with, as one commenter put it, "*a fuzzy worker*" [41]. The same commenter argued that the novelty of loop engineering comes from having a non-deterministic worker, requiring engineered "*guardrails around it*" [41]. Some developers suspect the AI labs' interests behind a practice that consumes many tokens, a suspicion Orosz discusses under the heading "*tokenmaxxing*" [40]. Orosz's own investigation, drawing on roughly 210 practitioner replies, concluded that most concrete examples of loops fall into two familiar buckets, i.e., cron jobs and event-based triggers, which are now executed by more capable workers [40, 41]. Max Kanat-Alexander expects agentic coding tools to absorb loop engineering, since anything that developers must know about a tool "*would eventually become baked into its default workflow*" [40]. To Orosz, with `/loop` and `/goal` now built-in, loop engineering already seemed "*as good as obsolete*" outside AI infrastructure [40]. However, by our definition, designing loops using built-in mechanisms is still loop engineering. In summary, the discourse has not yet concluded 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 an agent loop 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*" [22]), binding constraints on what an unattended agent loop may change without human approval, and defined escalation points to humans [22, 24, 29, 43]. However, the practitioner sources are not independent. Greyling's community repository and HuaShu's playbook 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*" (while the loop is running) [43], Greyling's verifier skill rejects by default [22], and Lindenberg warns that a loop "*satisfies the gate you wrote*", not the more abstract goal behind it [29].

The sources also recommend staged adoption. Greyling's readiness levels start report-only (L1) and reach unattended operation (L3) only once the verifier has proven reliable [22]. Osmani's later post revises the eight stages of developer evolution that Steve Yegge had published in January 2026 (from near-zero AI use to building one's own orchestrator) [60]. Osmani recommends choosing an autonomy level for each task and distinguishes *agency*, how far one agent proceeds without human intervention, from *orchestration*, how many agents run at once and who coordinates them [42]. Goal-driven autonomy (L3) is the highest agency level in Osmani's classification. The agent iterates until a measurable stopping condition holds. L4 and L5 describe parallel delegation and managed-by-exception orchestration. A team should raise autonomy only as far as it can verify the result at acceptable cost [42].

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 [40]. The boldest claims, however, are self-reported. For example, 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 [55], and Osmani reports that Mozilla used a loop to produce 423 security fixes for Firefox in one month [44]. No published study or dataset we know of supports these numbers, and HuaShu's playbook states that other numbers circulating in the discourse "*are mostly secondhand summaries and should be treated as rough reference*" [24].

O5: The discourse comprehensively documents failure modes. Reported problems include high token usage (one report describes a 'CI sweeper' loop using roughly eight million tokens in 48 hours), infinite fix loops, verifiers that approve without real checks ("*verifier theater*"), notification fatigue, and drift between a loop's committed instructions and its state [22], as well as plain disappointment from developers who tried loops and found they did not help [40]. Two risks were described under the labels *comprehension debt*, i.e., the gap between what exists in the repository and what the developer understands grows with loop velocity, and *cognitive surrender*, i.e., using loops to avoid thinking rather than to move faster on understood work [21, 43]. Comprehension debt is the individual counterpart of what Storey calls *cognitive debt*, a team's eroding shared understanding of the system it changes. She places cognitive debt beside *technical debt* and *intent debt*, the unrecorded goals and rationale that developers and coding agents need to change a system safely [52]. Commenters in Orosz's thread observe that nightly agent loops can open more pull requests than teams can review, so the volume itself defeats the "*reviewed by devs*" guardrail that the loop depends on. Review fatigue turns the human gate into a "*rubber stamp*" while "*the pipeline still reports green*". Another commenter argues that loops remain in use only when someone is already paid to review their output [41].

O6: Systematic evidence on the value claims is missing. The discourse agrees on a design vocabulary but makes conflicting claims about the benefits and costs of loop engineering. The collected sources provide no systematic evidence to evaluate these claims. Together, O1–O6 answer **RQ1**: The discourse defines loop engineering as designing automated control structures that repeatedly invoke coding agents, agrees on its building blocks, and disputes its novelty and value.

4 Empirically Studying Loop Engineering

4.1 Mining Loop Engineering Traces (RQ2)

Table 2 summarizes the loop mechanisms we searched for, together with related repository artifacts, and the number of repositories where our heuristics matched artifacts for each mechanism. The result columns count repositories with detected matches (Detected), confirmed matches (TP), and confirmed matches and evidence of loop execution (TP+L). For each matched repository, we checked workflow run histories and agent outputs for evidence that an autonomous agent loop had been executed. Evidence of loop execution is per repository, not per artifact.

Detection. Our mining study builds on a repository sample (n = 36,710) from an AI coding tool configuration dataset we built earlier. These repositories were classified with an LLM as engineered software projects, out of a sampling frame of actively maintained repositories in ten programming languages (n = 40,585) [4, 13].

On 19 August 2026, we searched every repository in this sample for committed loop artifacts with deterministic file- and content-based *heuristics*, and tested the commit history of candidate context and state files for three behavioral *signals*, defined below. We searched each repository at its current head commit and dated each detected artifact by its introducing commit. We found that 65 repositories (0.18%) were no longer reachable on GitHub. We further excluded 158 duplicate repositories (0.43%, see Section 6), resulting in a sample of 36,487 repositories. We used a *trigger heuristic* to identify scheduled or event-triggered invocations of eight coding tools in committed files: five tools that we considered when building the above-mentioned AI configuration dataset (Claude Code, GitHub Copilot, Cursor, OpenAI Codex, and Gemini CLI), plus OpenCode, Hermes, and OpenClaw. We added these three open-source tools because they provide built-in scheduling or headless run-until-done commands that our heuristics can recognize in committed files [22].

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 = 36,487); 'TP' (true positives): repositories with a match that we confirmed. 'TP+L': TP plus evidence that an autonomous agent loop ran. We did not validate provenance; local worktrees are not visible in GitHub data.

Mechanism	Role in the Loop	Example Repository Artifacts	Traceability from GitHub Data	Detected	TP	TP+L
Triggering & scheduling	Starts agent loops 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	215
Goal & stop conditions	Specifies when an agent loop 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	0
State & memory	Persists findings and progress across agent loops	`STATE.md`, `LOOP.md`, plan/progress/log files	Files plus commit history revealing cadence, authorship, and append-only updates	2	0	0
Skills & intent	Encodes durable project knowledge read in every agent loop	`.claude/skills/`, `SKILL.md`	Presence detectable; runtime usage invisible [15]	1	1	0
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 subagents (`.claude/agents/`), CI gates, protected branches	Definitions and CI results visible; agent-side verdicts invisible unless logged	0	0	0
Human oversight & escalation	Requires human approval for defined actions; escalates on demand	Pull-request reviews, `CODEOWNERS`, escalation issues, gate rules in loop docs	Review traces and documented gates visible; interactive supervision invisible	15	13	10
Budgets & pause controls	Caps spend per agent loop; restricts what an unattended agent loop may change	`loop-budget.md`, `loop-constraints.md`, pause labels	Declarations visible when committed; enforcement invisible	0	0	0
Outcomes & provenance	Identifies what the loop produced	Bot-authored commits/PRs, `Co-Authored-By` trailers	Attribution conventions inconsistent across tools and teams [46, 49]; attribution to loop executions not assessed	—	—	—
Cost	Tokens and money actually consumed	Run logs with token estimates	Rarely committed; observed logs hold template values, not measurements [22]	0	0	0

Loop building blocks can also occur as ordinary outer-harness configuration. For example, finding a skill describing how to run tests does not tell us whether that skill is part of a loop. Our heuristics therefore required evidence linking artifacts to loops: scheduled or event-triggered agent invocations, skills or verifier subagents referenced by loop artifacts, and dated run-log entries or combinations of loop-related content in state files. For GitHub Actions, the trigger heuristic excluded workflows started only by a human mention, label, or manual dispatch (`workflow_dispatch`), which we treated as interactive agent use. It retained workflows declaring `schedule`, `workflow_run`, or `repository_dispatch`. The supplementary material provides the complete heuristics and code used to apply them [31].

Dedicated loop artifacts are rare. We therefore complemented our search for committed loop artifacts with behavioral signals from the commit histories of context files (e.g., `AGENTS.md`, `CLAUDE.md`) and candidate state files (e.g., `STATE.md`, `LOOP.md`, plan and progress files). Loops that commit updates to these files can leave patterns in their commit histories. In each file's latest 15 commits (or all commits if fewer), we looked for three signals: (1) *cadence*, i.e., a regular inter-commit interval between ten minutes and seven days; (2) *authorship*, i.e., a strict majority of commits attributed to one of the eight coding tools; and (3) *append-only*, i.e., edits that add dated run-log entries with few deletions. Because maintenance bots produce similar commit patterns and humans also edit context and state files [34], we excluded 12 maintenance bots and classified a repository as loop-adopting only if *authorship* co-occurred with one of the other two signals on a detected state or budget artifact. We also treated co-occurring signals in a context file as a reason to inspect the repository, even without a detected loop artifact.

For our commit-history analysis, we selected repositories with artifact matches or candidate state files. From our AI coding tool configuration dataset, we also selected repositories with context files that had at least five commits, including at least one attributed to a coding tool. This yielded 792 repositories. In 637 of them, we analyzed 1,334 commit histories of context or candidate state files with at least five commits.

Validation and inspection. We drafted the heuristics and thresholds from Greyling's repository and checked tool-invocation patterns and the trigger heuristic against a 100-file GitHub code-search sample. The sample contained no state or budget files on which to test the state and budget heuristics. We inspected every matched repository to estimate precision, but did not construct a labeled set of repositories without loops to estimate false-positive rates. The supplementary material explains the thresholds and reports per-file signal measurements [31].

In the full repository sample, our heuristics produced 341 artifact matches across 255 repositories. With guidance from Claude Code, the last author reviewed these matches to check whether each file contained the claimed loop mechanism, confirming 325 matches.

Authorship and cadence co-occurred in a context file in one additional repository. We inspected all 256 repositories for evidence of autonomous agent loop execution. Using a codebook included in our supplementary material [31], Claude Code with Fable 5 judged whether each repository runs an autonomous agent loop (*yes*, *no*, or *unclear*); with Opus 5, it then tried to refute these labels and changed disputed *yes* or *no* labels to *unclear*. The last author then approved or revised these labels. We disclose this LLM-as-annotator design following the community guidelines [3].

For adjudication, we archived GitHub Actions run histories, job and step records, and evidence of outputs such as commits and issue comments on 19 and 20 August 2026. The last author checked the model-assigned labels for autonomous agent loop execution against these archived records and retained 196 of the 256 labels.

GitHub marked some workflow runs as successful even though the agent step was skipped or ended without doing the requested work. The last author checked the agent step's duration and output to determine whether the agent had actually run. For example, `AI-Hypercomputer/xpk`'s Gemini triage logged 6,290 workflow

runs, but the search for issues to triage returned none at the head commit we examined, and none of the workflow runs that we checked reached the agent step.

The archived GitHub Actions histories also showed Copilot review runs without corresponding committed workflow files. GitHub offers an account-level setting that requests a Copilot review automatically and leaves no trace in the repository [19]. In 95 of the 159 repositories for which we measured review timing, Copilot reviews clustered on individual authors' pull requests, most arriving within minutes of one being opened. That pattern suggests the account setting rather than a repository-wide rule. We did not count this pattern as adoption by the repository because an individual contributor's account settings can generate reviews across projects. We confirmed a loop only on committed configuration or where Copilot's review volume approached the repository's pull request volume.

Loop execution. The last author's review confirmed autonomous agent loops in 217 of the 256 repositories that we inspected, 0.59% of all analyzed repositories. We excluded two repositories from "*TP+L*" because they had no confirmed artifact match (Table 2). We found no autonomous loop in 33 of the inspected repositories. For the remaining six repositories, we could not determine whether an autonomous agent loop had run. The precision of our repository detection was 0.87 (217/250), excluding these six repositories.

The most common tool in the confirmed loops was Claude Code (189 of 217), followed by OpenAI Codex (11), OpenCode (6), Gemini CLI (5), Cursor (4), and Copilot (2).

We grouped the confirmed loops by the trigger that accounted for most of their archived workflow runs: 21 ran on a schedule (e.g., `dimagi/commcare-android`, with 276 scheduled workflow runs in the archived GitHub Actions history), 180 ran on repository events (mostly automatic pull request reviews), 15 on both, and one ran until its GitHub Actions workflow broke. In 12 of the 217 confirmed loops, every trigger workflow that we detected was marked as manually disabled. We counted those loops and the one with the broken workflow as confirmed because they demonstrably ran.

Detected mechanisms. Table 2 reports how many of the 36,487 analyzed repositories had artifact matches for each mechanism. The trigger heuristic matched 253 repositories (35 scheduled only, 205 event-triggered only, 13 both). The last author's file review confirmed agent-invoking trigger definitions in 242 of these repositories. All but one of the 323 trigger matches were GitHub Actions workflows; the exception was a committed shell script.

For state and memory (Table 2), our searches of committed files and their histories found no loop state files or updates on the default branches that we scanned. Our artifact heuristics matched only two state files, one skill referenced by a loop artifact, no stop condition, no budget file, no verifier subagent referenced by a loop artifact, and no cost log with measured values. We rejected both state files on inspection as homonyms of unrelated "*loop*" vocabulary. The one skill match was genuine: an issue-triage skill written for a loop. Its repository's committed agent workflows required a human request for each run, so we did not count them as autonomous loops. In 96 repositories, we found a skill with a name or content indicating a loop, and in 18 a subagent with a name or content indicating a verifier, but no loop workflow or loop document in those repositories referred to these skills and subagents.

Claude Code used workflow definitions, prompts, and archived GitHub Actions run histories to annotate the repositories. The annotations recorded a goal or stop condition in 55 of the 256 inspected repositories, including committed prompts that told the agent when to stop. Table 2 reports no goal artifact because we matched a fixed list of stop-condition phrases against the workflow or document that started a loop and the skills anchored to it, and none of these phrases occurred in those files. The annotations also recorded state persisted across agent loops in 35 of those repositories, a spending or turn limit in 101, a verifier in 76, and a cost record in 10. We also checked for the autonomy levels described in the practitioner literature (Section 3), but no inspected repository declared an autonomy level for its loops.

For parallel isolation (Table 2), GitHub data includes branch names but not the local Git worktrees used to separate concurrent agents' work. We found 3,507 repositories with two or more branches under agent name prefixes (e.g., `claude/`). These branches indicate coding-agent use, but do not establish whether a loop isolated parallel work [46].

For outcomes and provenance (Table 2), the commit histories showed coding-tool contributions to context files. Among the context files that we analyzed, 384 files in 217 repositories had a majority of their latest 15 commits (or all commits if fewer) attributed to coding tools. Of these repositories, 30 also had a detected trigger, state, or budget artifact. Only 25 of the 217 repositories had confirmed autonomous loops. These attributions did not establish that agent loops produced the commits to the context files; we did not link the commits to specific loop executions.

Together, these results answer **RQ2**: Committed loop artifacts primarily documented triggers. Although the artifact heuristics found no goals, budgets, or verifiers, Claude Code recorded goals in 55 of the 256 inspected repositories, budgets in 101, and verifiers in 76. However, neither the searches of committed files nor the commit-history checks provided evidence of the persistent loop state files suggested by practitioners.

4.2 Loop Goals and Feedback (RQ3)

To answer **RQ3**, we propose a controlled experiment focusing on individual agent loops (Figure 1). The experiment varies three *factors*: loop goals, static analysis feedback, and a decision policy. The loop goal factor has two *levels*: no goal and a goal to resolve the provided issue. Static analysis feedback is independently disabled or enabled. The decision policy is either 'accept-all' or a simulated developer. The eight combinations of factor levels define the *treatment conditions*. We compare outcomes between treatment conditions that differ in only one factor, keeping the other two factors fixed. All eight conditions use the same harness and model versions and the same time and token limits. We will publish the prompts and tool parameters for every treatment condition. The design follows established guidelines for experimentation [57] and empirical studies involving LLMs [3].

We will select tasks from SWE-bench [27] or similar benchmarks. Each task starts from the benchmark's *reference commit*. Tests that pass at that commit serve as *regression tests*. The issue-resolution tests supplied by the benchmark serve as *benchmark tests* for final scoring. Before collecting developer interactions, we will check

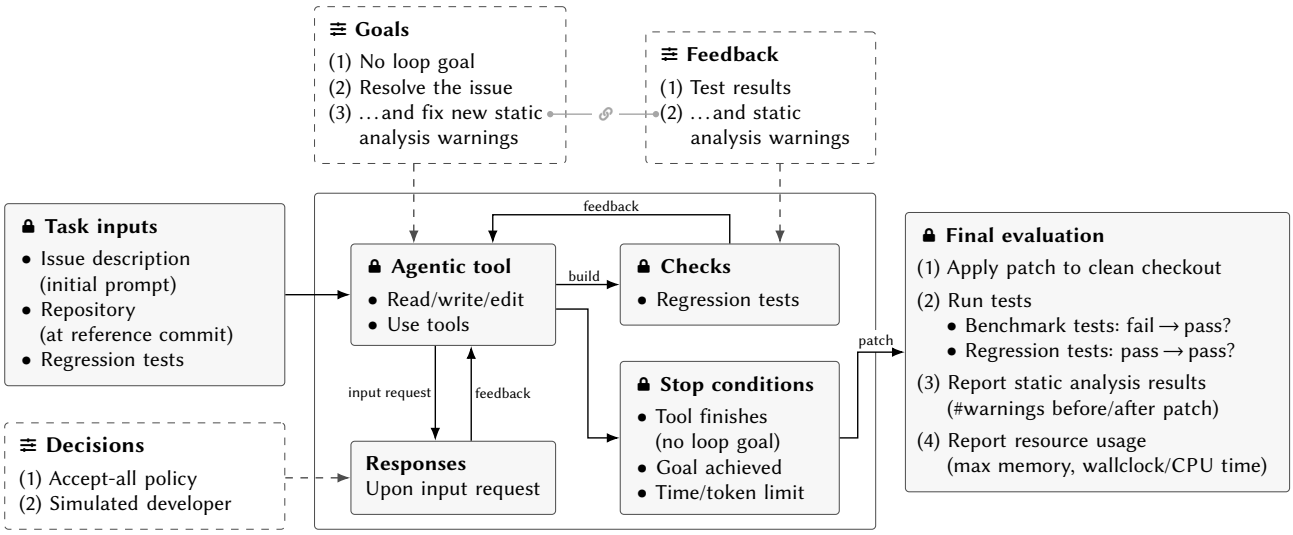

Figure 1: Experimental design for RQ3: Goals, tool feedback, and decision policy vary (☲). The issue description is the initial task prompt in every agent loop. When setting a loop goal, we explicitly require resolving the issue. With static analysis feedback, we extend the goal to require fixing new static analysis warnings (🔗). Each decision policy is tested with and without a loop goal and with and without static analysis feedback, giving eight treatment conditions. Benchmark tests are used only for final scoring, but agents may create and run additional tests in every treatment condition. Task inputs, harness and model versions, time/token limits, and final evaluation remain fixed (🔒).

that benchmark tests fail at the reference commit and pass with the benchmark's reference patch, while regression tests remain passing. We will also manually compare the issue description, test assertions, and reference patch to identify requirements without tests and assertions that impose unstated requirements. Issues with unexpected test results or identified mismatches will be excluded [38, 39]. These checks may still miss test defects. The supplementary material for the planned journal extension will specify the selected benchmark release and the version or commit of the code used to score patches.

While the agent loop is running, regression tests are available in the repository checkout. We apply the benchmark's test changes only when scoring the final patch produced by the agentic coding tool [27]. *Benchmark contamination* remains possible if benchmark tests or solutions were included in the LLM's training data [59]. Such contamination can inflate benchmark scores [39]. The agentic coding tool uses the same LLM version in all experimental conditions, so any training-data contamination is shared. This makes contamination less concerning for comparisons between conditions than for absolute scores, although it could still affect conditions differently.

Agents may create and run additional tests in every treatment condition, even without an explicit instruction to do so. We will record all agent actions in interaction traces to determine retrospectively whether and when agents created additional tests.

We use the issue title and body as the initial task prompt in every treatment condition [27]. All experimental conditions use the same system prompt and tool definitions. In conditions with a loop goal, we explicitly set a goal using `/goal` or an equivalent command. The default goal is to resolve the provided issue (*Goals* in Figure 1). With static analysis feedback enabled, we extend the goal to also require fixing new warnings. We identify new warnings by comparing static analysis results with those at the reference commit. Without an explicit loop goal, we assess the effect of static analysis feedback while keeping the issue description as the initial task prompt. With an explicit goal to resolve the issue, we assess the combined effect of static analysis feedback and the requirement to fix new warnings.

For the repositories from which the selected benchmark tasks were derived, we will check existing build scripts and modify them as needed to run the regression tests and, when enabled, static analysis. We will add a build script if none exists. When the agentic coding tool runs these scripts, we provide their output as feedback (*Feedback* in Figure 1).

The *simulated developer* responds to user-input requests from the agentic coding tool (*Decisions* in Figure 1). It uses an LLM guided by an agent skill or context file [14] derived from recorded developer interactions to provide responses similar to those of human developers. As a simple baseline, we use an accept-all policy to approve proposals automatically. For mutually exclusive or open-ended choices, we ask the agentic coding tool to recommend an option and automatically accept its recommendation.

To derive such an agent skill or context file, we will record developers' responses to user-input requests while they work with an agentic coding tool on selected benchmark tasks. Participants will be recruited from industry contacts, open-source contributors, and experienced students. Published interaction logs of developer-agent sessions may provide an additional source for deriving the agent skill or context file. We will validate the simulated developer on held-out sessions from different tasks. For each user-input request in these sessions, we will compare the simulated developer's response with the recorded human response.

We will report the proportion of evaluated user-input requests for which the simulated developer's response agrees with the recorded human response. For fixed-choice requests, agreement means selecting the same option. We will compare free-text responses using an LLM-as-judge approach. If agreement falls below a minimum agreement threshold specified before starting the evaluation, we will omit the simulated developer and retain the accept-all baseline. We may then compare accept-all with a baseline that selects uniformly at random among permitted option combinations. If we use the simulated developer, we will repeat the experiment with a second LLM for that role. We will check whether the differences between experimental conditions change in size or direction for each outcome measure.

The experiment uses three *Stop conditions* shown in Figure 1. For agent loops without a loop goal, the agentic coding tool's final response stops the loop. For agent loops with a loop goal, the loop stops when the goal is achieved. All agent loops also stop if they reach the specified time or token limit.

In the *Final evaluation* (Figure 1), we save the changes made by the agentic coding tool as a patch when a stop condition is met. We then apply this patch to a clean checkout of the task's repository at the reference commit. To assess task success, we run the benchmark and regression tests. Task success requires all of these tests to pass [27]. An agent loop counts as unsuccessful if it did not result in a patch or if the patch fails any benchmark test or regression test.

Besides functional correctness, we will assess the code quality of the generated patch. To this end, we will count static analysis warnings before and after each patch, including added and removed warnings. We will use static analysis tools appropriate to each language, with comparable rule sets and severity thresholds across languages. Tool versions and configurations will remain fixed across experimental conditions.

Another non-functional property that we want to measure is resource usage. During the final test execution, we measure wall-clock time, maximum memory use, and CPU time using identical hardware and execution environments. We also record the duration of each agent loop and input and output token consumption for the agentic coding tool and the simulated developer. We will then compare test-execution costs across experimental conditions per benchmark task. For this comparison, we retain only tasks for which all conditions being compared produced patches that pass the tests.

5 Discussion

We discuss the implications of our findings for software engineering researchers, practitioners, and educators.

Implications for researchers. Our gray literature review identified recommendations for loop goals, independent verification, and budgets. However, the reviewed sources did not provide systematic evidence of their effects. Researchers evaluating the effectiveness and cost of loop engineering could compare agent loops with and without explicit goals, independent verification, or budgets, measuring task success and resource usage. Our proposed experiment tests explicit goals, static analysis feedback, and decision policies for user-input requests on individual issue-resolution tasks.

Researchers studying the adoption of autonomous agent loops need to verify agent execution because some successful GitHub Actions runs in our dataset skipped their agent step. A follow-up adoption study could survey maintainers of repositories in our mining sample about their use of agent loops. Researchers could then invite maintainers who report that they use agent loops to share interaction traces and scheduler logs, including for loops whose configuration is not committed.

These execution records could also support longitudinal studies of agent loops used for pull request review and maintenance, the main tasks identified in our mining study. Linking successive agent loops to their starting commits, verification results, and resulting changes could reveal repeated reviews of unchanged code or reversals of earlier changes. Recording the time, token consumption, and developer interventions associated with repeated reviews and reversals could quantify the cost of this rework across successive agent loops.

Implications for practitioners. Teams planning to implement agent loops for maintenance could test verification approaches from our gray literature review. For example, they could compare verification by the agent making changes with verification by a second agent. The comparison could show differences in the proportion of patches that pass tests and receive developer approval, as well as in resource usage and token consumption.

Even when a second agent verifies changes, teams need to decide when agents should ask developers to clarify requirements. In our gray literature review, Ng emphasized developers' product and user knowledge as a "*context advantage*" [36]. Teams could therefore configure escalation policies that require agents to ask developers about unresolved product requirements even when automated checks pass.

Implications for educators. Educators in universities and professional developer training could ask learners to set a loop goal and budget for a maintenance task. Learners could then inspect execution records and resulting changes to check whether the goal was achieved within the specified budget.

Beyond individual loops. On 17 July 2026, Peter Steinberger asked whether the discussion had shifted from loops to graphs yet [50]. Within hours, Hamel Husain published "*Loop Engineering Is Dead. Enter Graph Engineering*" [26]. Lindenberg describes graph engineering as connecting loops in which specialized agents perform different tasks and share state [28]. To study graph engineering, experiments could keep individual loop designs fixed while varying task assignment and information sharing between loops. Outcomes could include task success, resource usage, and token consumption.

6 Threats to Validity

Construct validity. Construct validity concerns whether our measures capture the intended properties [45]. Definitions in our gray literature review differed in how they distinguished loop engineering from other agent-invoking automation. For our mining study, our working definition and codebook required autonomous agent execution, excluding workflows that needed a person to start every agent loop. We included loops that had executed before being disabled, so the adoption rate does not measure how many loops were active on the collection date.

For the *authorship* signal, which we defined to identify files predominantly updated by coding tools, we inferred coding-tool contributions from names, email addresses, and commit trailers. We excluded 12 maintenance bots and counted `github-actions[bot]` commits only when their messages referred to a workflow run and the repository had a detected agent workflow. We did not match these run references to specific GitHub Actions runs, so we could misattribute commits from unrelated workflows to a coding tool. We mitigated the effect of commit-attribution errors on our estimate of autonomous agent loop adoption by requiring the last author to confirm loop execution using archived GitHub Actions run histories and agent outputs.

Internal validity. Internal validity concerns the extent to which we can rule out alternative explanations for our results [8]. For the gray literature review, source selection and interpretation by the last author could have favored prominent practitioners or favorable experiences. We included skeptical positions and failure reports, the last author checked extracted quotes, and all authors reviewed the six observations. Several posts reused the same seed sources, so agreement among them was not independent corroboration.

Our mining study's artifact heuristics (Section 4.1) might miss loops with unrecognized file conventions or tool invocations. We checked trigger and tool-invocation patterns against a 100-file sample, and the last author reviewed all 341 artifact matches, rejecting 16 false positives. These checks cannot establish how many loops the searches missed. We did not inspect a sample of repositories without matches or qualifying behavioral signals, so the reported adoption rate may underestimate autonomous agent loop adoption.

The capture limits of our data collection pipeline omitted files in 26 repositories and truncated large files in 795 repositories, potentially hiding loop artifacts. The affected repositories are identified in the supplementary material. Our search of committed files covered only default branches. For example, `dimagi/commcare-android`'s workflow stored memory on the `memory/pr-comment-handler` branch, outside that search. Our commit-history checks required at least five commits per candidate file, so they could not detect patterns in files with shorter histories.

The last author checked the LLM-assigned loop labels against archived run histories and agent outputs, but read the model's judgments before making his own decisions. Those judgments could have influenced him, so the label-retention rate does not measure agreement between independent annotators. In six inspected repositories, he could not determine whether an autonomous agent loop had run. We excluded these repositories when calculating precision.

External validity. External validity concerns the extent to which our results generalize to other contexts [5]. We estimated the adoption rate of autonomous agent loops in a repository sample from the AI coding tool configuration dataset [13]. This sample was drawn in March 2026 from actively maintained projects in ten programming languages. Searching its current head commits in August updated the repository data, but did not add projects outside the original sample. The results do not establish prevalence across all open-source software or the internal company systems described in our gray literature review. The artifact heuristics recognized only the eight coding tools listed in Section 4.1, excluding loops implemented through other tools or direct LLM API calls. We excluded repositories with identical head commits to reduce duplication, but forks with different head commits may still contain copies of the same agent workflow files.

Data availability. Our supplementary material is openly available on Zenodo [31]. It contains the mining scripts, rule catalog, result data, annotation protocol and codebook, adjudication record, per-repository evidence files, and archived GitHub Actions run histories. It also includes a glossary, a snapshot of the MIT-licensed loop engineering reference repository, and a source list linking to the archived academic publications, practitioner posts, and tool documentation.

7 Conclusion

Our gray literature (**RQ1**) review found broad agreement on the building blocks of loop engineering but no systematic evidence for its claimed benefits. Our mining study (**RQ2**) extended Galster et al.'s analysis of harness configuration [15] to autonomous agent loops, which we confirmed in 217 open-source repositories. Most of the identified loops reviewed pull requests or ran scheduled maintenance. We found no loop state files on the default branches that we scanned.

Our proposed experiment (**RQ3**) will test how explicit loop goals, static analysis feedback, and decision policies for user-input requests affect task success, code quality, resource usage, and token consumption. It is a first step toward closing the evidence gap identified in our gray literature review. In the journal extension, we will run this experiment and repeat the gray literature review and mining study to examine changes in definitions and adoption.

Future studies of loop engineering should consider execution traces that connect goals, state updates, verification, and the changes produced by agents. We call on software engineering researchers to measure how autonomous agent loops affect issue resolution, code quality, development costs, and developers' understanding of their software.

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] Anthropic. 2026. Claude Fable 5. Claude Platform Docs, model overview. https://platform.claude.com/docs/en/models/fable-5/overview Accessed 2026-09-15.

[3] 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.

[4] 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

[5] Sebastian Baltes and Paul Ralph. 2022. Sampling in Software Engineering Research: A Critical Review and Guidelines. *Empirical Software Engineering* 27, 4, Article 94 (2022), 31 pages. doi:10.1007/s10664-021-10072-8

[6] 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

[7] 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.

[8] Marilynn B. Brewer. 2000. Research Design and Issues of Validity. In *Handbook of Research Methods in Social and Personality Psychology*, Harry T. Reis and

Charles M. Judd (Eds.). Cambridge University Press, Cambridge, UK, 3–16. https://assets.cambridge.org/97805215/59034/excerpt/9780521559034_excerpt.pdf

[9] 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

[10] 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.

[11] 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

[12] 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

[13] 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 archived on Zenodo, DOI 10.5281/zenodo.19375880.

[14] 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

[15] 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".

[16] 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

[17] 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

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

[19] GitHub. 2026. Configuring code review by GitHub Copilot. GitHub Docs. https://docs.github.com/en/copilot/how-tos/copilot-on-github/set-up-copilot/configure-code-review Accessed 2026-09-15.

[20] 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

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

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

[23] 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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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).

[31] 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

[32] 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.

[33] 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

[34] 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.

[35] 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.

[36] 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.

[37] 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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018 (EASE '18)*. Association for Computing Machinery, New York, NY, USA, 13–23. doi:10.1145/3210459.3210461

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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.

[51] 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.

[52] Margaret-Anne Storey. 2026. From Technical Debt to Cognitive and Intent Debt. *Queue* 24, 2 (March 2026), 15 pages. doi:10.1145/3807966

[53] 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

[54] 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

[55] 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.

[56] 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

[57] 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

[58] 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.

[59] Cheng Xu, Shuhao Guan, Derek Greene, and M-Tahar Kechadi. 2024. Benchmark Data Contamination of Large Language Models: A Survey. arXiv:2406.04244 doi:10.48550/arXiv.2406.04244

[60] 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.