

Instant Software Factories

Wenxin Jiang, Mikola Lysenko

Socket Inc.

{wenxin,mik}@socket.dev

Abstract—Coding agents can quickly build prototypes, but a long tail of fixes, optimizations, tests, and releases separates a prototype from a product. A *software factory* handles this ongoing work through a repeatable process. We ask whether this work needs dedicated orchestration and how such factories should be judged. Given a tested repository and a coding agent, an *instant software factory* uses only two coordination inputs, a swarm of *cron-job agents* with standing objectives and a shared *communication board*, such as GitHub, where agents coordinate through the traces they leave. We developed a bootstrap prompt designed to set up both inputs from one round of human input, with governance as a separable layer. We distilled it from two manually configured industrial factories, observed for 15 and 17 days, and have not yet evaluated it. In the GitHub-coordinated CLI factory, daily merges rose 15× during agent rollout, an observed association. Its agents also filed 95% of new issues and authored 40 of 58 PRs that changed CI workflows, and at least 5.6% of issues marked as fixed were later found to have incorrect fixes. Unlike an agent scored on a fixed benchmark, a factory thus changes its own workload, evaluator, and measured progress. We formulate factory development as multi-objective optimization under this *endogenous feedback*. Over five months, we plan to evolve both factories, test the bootstrap on new repositories, and test five falsifiable hypotheses.

Index Terms—Autonomous software engineering, multi-agent systems, software factories, software maintenance

I. INTRODUCTION

Coding agents, systems based on large language models (LLMs) that read code, use tools, and propose changes, have opened over 930,000 pull requests (PRs) on GitHub [1] and quickly build prototypes. Yet a prototype is far from a product. Bugs, edge cases, regressions, compatibility breaks, flaky continuous integration (CI), and releases create a long tail of work (Fig. 1, top). Prompting an agent through every task makes developer attention the bottleneck. A *software factory* handles such work with specialized roles, common tools, and repeatable processes [2].

Building a factory for this tail is often treated as a substantial engineering effort. Stripe, Ramp, Spotify, and Uber have built internal agent platforms [3]–[6], research systems use orchestrators and message passing [7]–[9], and MAGE models factories as production systems with many controls [10]. But how much infrastructure does a software factory actually need, and how should its progress be judged?

We explore a simpler alternative, an *instant software factory*, and separate two claims. Our *sufficiency hypothesis* is that **two coordination inputs suffice for a factory for PR-sized work**. Our *feasibility hypothesis* is that a bootstrap prompt we developed can configure both inputs in one session from a

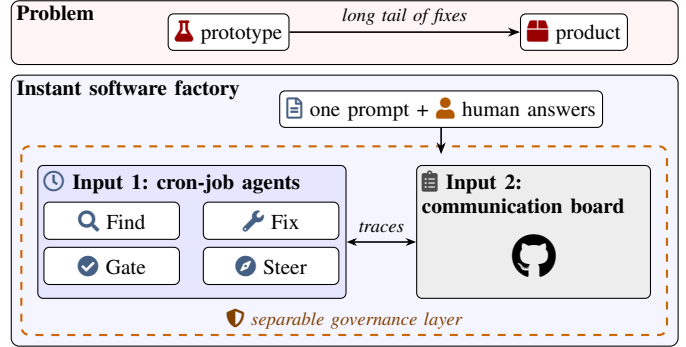


Fig. 1. Instant software factories. Cron-job agents and a communication board such as GitHub coordinate the factory. Governance is separable but mandatory.

single round of human answers (Fig. 1, bottom). One factory gives early evidence for the first, and the second is untested.

We distilled this design from two industrial factories built by hand before the bootstrap existed. They suggest that judging a factory is harder than assembling one, which we see as the main new problem. Agent benchmarks fix the tasks and tests, and studies of reward overoptimization fix the reward model [11]. A factory’s actions instead change its product, the work it is given, and the process that evaluates it, a loop we call *endogenous feedback*. In the CLI factory, agents filed 95% of issues (*endogenous workload*) and authored 40 of 58 CI workflow PRs (*endogenous evaluation*), and 7 incorrect fixes shipped unrepaired (*delayed verdicts*). Throughput can then overstate progress, so factories must optimize independently valued, validated, and delivered changes.

Key idea. Given a tested repository with CI and a coding agent with repository access, a factory for PR-sized work may need only two coordination inputs: (1) ⌚ **Cron-job agents** with standing objectives, such as finding bugs or improving CI. Each run reads the board, does one bounded task, and exits. (2) 📋 **A communication board**, such as GitHub’s issues, PRs, checks, and reviews. Agents coordinate through *stigmergy*, following each other’s traces without an orchestrator. 🛡️ **Governance**, such as approval and a stop switch, is a separable, mandatory layer scaled to risk.

We contribute (1) a formulation of factory development as **multi-objective optimization under endogenous feedback**, with a progress measure that agents cannot edit and five testable hypotheses (§V–VI), (2) **observations** of endogenous workload, endogenous evaluation, and delayed verdicts in two factories (§IV), and (3) a **two-input design**, posed as a sufficiency hypothesis, with a released, not yet evaluated **bootstrap prompt** [12] (§III).

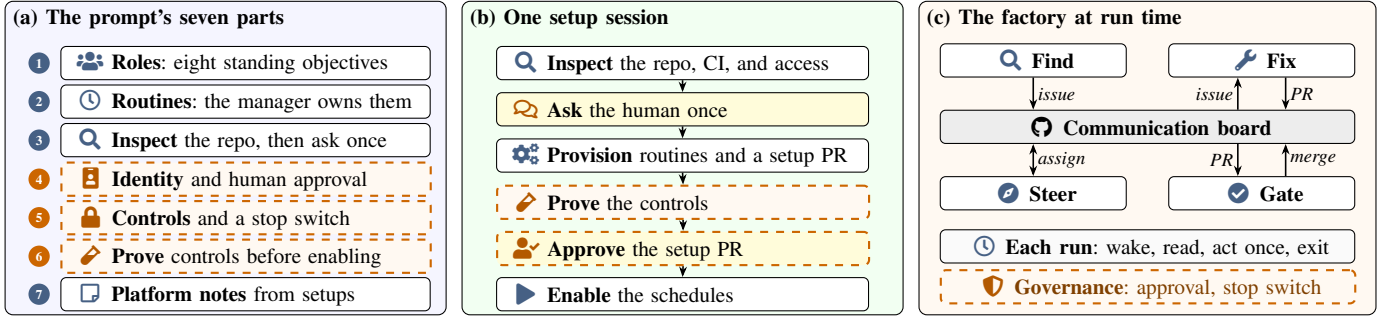


Fig. 2. Anatomy of the bootstrap. (a) Prompt parts. (b) Setup session. (c) Running factory. Orange dashed items form the separable governance layer.

II. AGENT SWARMS AND SOFTWARE FACTORIES

Swarms and stigmergy. Social insects coordinate through *stigmergy*, leaving traces in a shared environment that guide later actions [13], and blackboard systems let specialists coordinate through shared information [14]. LLM agents sharing persistent artifacts develop specialized roles [15], but teams without centralized verification can propagate errors [16], motivating a shared board and a merge gate.

Agent factories. Research systems and toolkits rely on message passing or dedicated orchestrators [7]–[9], [17]. Practitioners have also coordinated scheduled agents through GitHub issues and PRs [18]–[20], but typically configure agents individually and report activity or merge rates. We add no coordination mechanism and instead test whether these primitives suffice for a whole factory initialized by one prompt, judging factories under endogenous feedback by changes that hold after merge and reach users.

III. THE BOOTSTRAP

We developed a bootstrap prompt, distilled from two hand-built factories (§IV), to set up a factory in one session, an untested feasibility hypothesis (§VI). A developer gives a coding agent in the target repository a 1,400-word prompt (Fig. 2). The design targets tasks that fit one PR and can wait an hour. Production deployment and multi-repository work are out of scope, and tightly coupled work may need planners [21].

⌚ Input 1: cron-job agents. Each agent is a saved *routine* launched on a cron schedule at most once per hour [22]. Runs keep no memory beyond the board. Eight roles divide the work. QA and performance agents *find* problems and file issues. Implementation, CI, and architecture agents *fix* them through PRs. Review and merge agents *gate* changes. A manager, itself a routine, *steers* work through the board, assigning tasks, tuning models and schedules within budget, and balancing objectives (§V).

📅 Input 2: the board. Issues record tasks and evidence, PRs carry changes, and checks and reviews provide verdicts. Any board that holds these three kinds of traces suffices. Our bootstrap targets GitHub and scheduled Claude Code tasks.

📅 One setup session. The agent inspects the repository, asks the human once about budget, QA priorities, and approvers,

and opens a setup PR, enabling schedules only after testing governance controls and obtaining approval.

Manual setup lessons. Hand-built routines shaped the platform notes. Tool lists alone did not enforce GitHub permissions, so the prompt tests each restriction. CI capacity and GitHub reliability were the main bottleneck, motivating adaptive run rates (§V).

🛡️ A separable governance layer. Governance restricts agent actions without changing how they coordinate. For software that users install, merges by default require passing checks, independent review of the latest commit, and human approval. Agents check a stop switch before every write and must ignore instructions in retrieved content. Teams scale controls to the risks of their changes, but the layer remains mandatory.

IV. TWO WORKING FACTORIES

We examine four dimensions of factory operation, *continuity* (working without repeated prompting), *governance* (controlling agent actions by risk), *correctness* (changes that hold under independent checks), and *delivery* (changes reaching users). Following case-study and GitHub-mining guidelines [23], [24], we studied two industrial factories operated by one organization: a *CLI system* for 15 days (escapes tracked through day 17) and an *agentic system* for 17 days. The CLI factory instantiates the two-input design, and the agentic factory offers a contrasting architecture. Neither evaluates the bootstrap’s automated setup.

➤ The CLI system applies security patches to dependencies across nine software package registries and about twenty package managers. It uses exactly two coordination inputs, about 30 scheduled agents, introduced over days 0–11, and GitHub as their shared board, with no orchestrator.

🔧 The agentic system patches vulnerabilities in open-source packages, validating each patch with a proof-of-concept attack and regression tests. We study the swarm maintaining its validation *harness*, whose checks returned no verdict in 28% of reruns. It instead uses service loops and a shared database work list. A scout groups failures into defect types, a fixer submits PRs, a verifier reruns checks, a judge rejects package-specific fixes, and a recovery agent restarts blocked runs.

01 (continuity): both factories sustained work with different architectures. After the CLI swarm started, daily merged PRs rose from 2.2 to about 35, peaking at 115 on

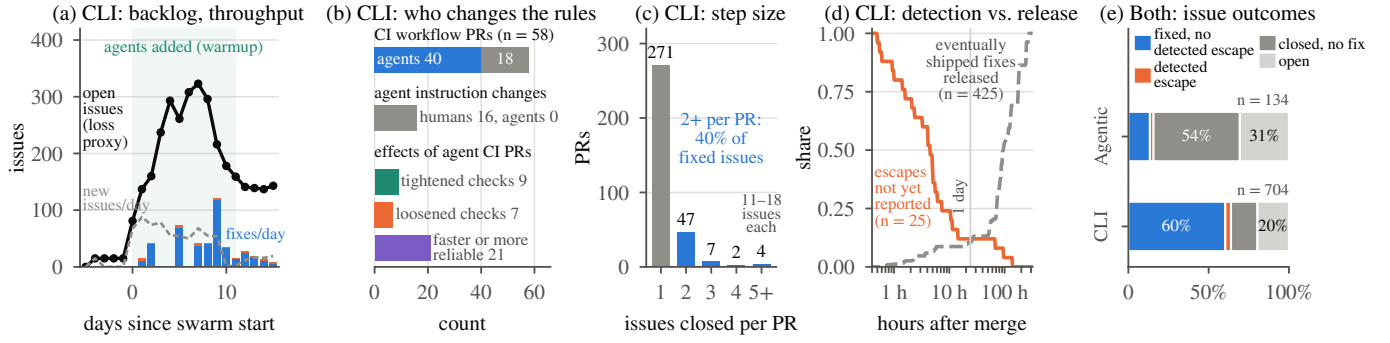


Fig. 3. The two factories through an optimization lens. (a) CLI open issues (loss proxy), new issues per day, and daily fixed issues (throughput) with no detected escape or an escape. (b) CLI changes to CI workflows and agent instructions by agents and humans, and LLM-rated effects of agent CI PRs. (c) CLI step size. (d) CLI detection against release, as the share of 25 escapes not yet reported and, for a different cohort, of 425 eventually shipped fixed issues (escapes included) already released. (e) Outcomes of 704 CLI issues and 134 agentic defect types. No detected escape does not establish correctness.

day 9. Agents filed 95% of 704 new issues, and merged PRs closed 450 (64%) as *fixed* [25] within a median of 46 hours (Fig. 3e). Open issues, a proxy for loss, peaked at 323 as agents discovered bugs, then fell to about 140 and stabilized as new reports matched fixes (Fig. 3a). On a sampled day with 69 merges, agents cost \$1.2k, or \$17 per merge before CI costs. The agentic factory spent \$869 on 21 merged harness fixes, or \$41 each, resolving 21 of 134 defect types (16%).

O2 (governance): controls depend on context and may limit throughput. In the CLI factory, 77% of agent merges changed user-installed code. A second person approved 534 of 541 merged PRs. A merge queue added on day 7 rejected 29 failing agent PRs and delayed others a median 26 minutes as daily merges doubled. Agentic fixes instead affect only an internal harness, and a judge screens them before human approval. One reviewer approved 502 CLI PRs without requesting changes, questioning review effectiveness and, as another agent-first team found [26], scalability.

O3 (governance): agents also modify the rules. CLI agents authored 40 of 58 PRs changing CI workflows, while the maintainer made all 16 changes to agent instructions. Two LLM raters classified CI effects ($\kappa = 0.91$), including 7 PRs that loosened checks (Fig. 3b). Humans reviewed these changes, so edits to the factory’s own controls stayed governed.

O4 (correctness): merged fixes do not always hold. By day 17, at least 25 of 450 fixed CLI issues (5.6%) were *escapes*: incorrect fixes [27] that broke another case of the issue (17) or left part unfixed (8) [28]. None was reopened, although reopening often signals faulty fixes [29]. In the agentic factory, 3 of 14 replayed fixes (21%) recurred.

O5 (delivery): merged fixes must reach users. Two CLI releases shipped 94% of fixed issues after a median *integration delay* [30] of 3.9 days. Agentic harness fixes propagate only when the runs they unblock restart, and a recovery sweep resumed 11 of 22 blocked runs.

O6 (delivery): escapes race releases. Reports, mostly from bug-hunting agents, identified 22 of 25 CLI escapes within a day of merge (Fig. 3d). Release delays allowed 14 to be

corrected before shipment, but 7 shipped unfixed, including 4 before being reported.

Threats to validity. Both factories belong to one organization, the bootstrap’s sole source, and were observed for two to three weeks. Both retained human approval, leaving the minimum necessary governance unknown. CLI agents were introduced over days 0–11, and their instructions changed 16 times, so before-and-after differences cannot establish causality. The different workloads and lack of an architectural baseline prevent claims of superiority. Operators stopped the agentic swarm on day 16, and humans amended 7 of its 21 fixes. Escape counts depend on scouting coverage, classifications rely on LLM raters, and CLI costs come from one sampled day. We found no evidence of intentional gaming and did not test whether agent CI changes caused escapes.

V. THE SOFTWARE FACTORY AS A MULTI-OBJECTIVE OPTIMIZATION PROBLEM

Inspired by search-based software engineering [31], we frame factory configuration as a *multi-objective optimization problem*, with merged PRs as steps and manager-controlled settings (objectives, schedules, PR scope, and notes) as hyperparameters under governance constraints (Table I). The framing defines what to measure and test and proposes no new optimizer. Let the factory state be $S_t = (P_t, B_t, E_t)$, the product, backlog, and validation process, with measured progress $\hat{R}_t = E_t(P_t, B_t)$. Agent actions a_t reach $\hat{R}_t$ through three channels. They file issues into B_t (workload, O1), edit CI so that $E_{t+1} = G(E_t, a_t)$ (evaluator, O3), and merge before an independent evaluator E^* judges P_t (delay, O4, O6). Maximizing $\hat{R}_t$ can thus reward growing the backlog, loosening checks, or merging early. We therefore measure progress with E^* , which agents cannot edit, and counter each channel with a safeguard, preset task values, held-out tests outside G (H5), and estimates taken after a fixed delay.

Initialization. As initialization affects convergence [32], the bootstrap would initialize objectives, roles, schedules, and controls. Its defaults come from two manual factories, may inherit their biases, and have untested effects.

TABLE I
OPTIMIZATION CONCEPTS MAPPED TO FACTORIES, WITH PRELIMINARY
DATA FROM §IV.

Concept		Preliminary data	
Optimizer	Factory	CLI system	Agentic system
Initialization	bootstrap	manual, 11-day rollout	manual
Objectives	agent goals (H1)	~30 agents	5 roles
Learning rate	run rate (H2)	~30 fixed schedules	service loops
Step size	issues per PR (H3)	450/331 = 1.36	one defect type
Momentum	notes (H4)	issue and PR threads	shared defect list
Generalization gap	escapes (H5)	≥25/450 issues	3/14 recurred

Objectives (H1). A single weighted loss can hide trade-offs between competing objectives [33]. CLI agents accelerated CI by moving end-to-end tests to nightly runs (O3), delaying detection of breakage until after merge. Specialists may expose such conflicts so the manager can protect the worst-served objective, while a generalist folds them into one loss.

Loss, velocity, and cost. Open issues estimate *loss*. We define *velocity* as lasting, delivered progress, like time-to-accuracy in training [34], and treat merge rates as throughput only. $N_i(\pi)$ counts unprompted changes toward objective i per week under configuration π that E^* judges correct, that reach users, and that carry severity weights set independently of the agents. The manager seeks a Pareto-optimal π :

$$\max_{\pi} (N_1(\pi), \dots, N_m(\pi)) \quad (1)$$

subject to weekly LLM, CI, and review cost $C(\pi) \leq C_{\max}$, human time $T(\pi) \leq T_{\max}$, escape rate $\leq \tau$, and governance constraints. Hidden escapes make measured N_i an upper bound, taken after a fixed audit window such as two weeks.

Worked example. Pooled over the CLI study, throughput counts 450 fixed issues. Delivery keeps the 425 that shipped, and correctness drops the 7 that shipped with a detected escape, leaving at most 418. As agents filed 95% of issues (O1), the maintainer would assign each an objective and a severity from a preset rubric, such as broken installs above wrong messages, and divide each N_i by a preset target so objectives are comparable (H1). Agent CI PRs would count only toward the CI objective, never as product progress.

Learning rate and momentum (H2, H4). Like an adaptive learning rate [35], run frequency could track verified progress, escapes, and cost, which neither factory did, although CI and review limited the pace (O2). Like momentum [32], *notes* in issue threads or the defect list carry context across memoryless runs but can mislead when stale.

Step size (H3). Issues per PR coarsely approximate step size, and each PR incurs review, CI, and merge costs (O2). Of 331 issue-closing CLI PRs, 60 fixed multiple issues, 40% of all fixed issues (Fig. 3c), with a per-issue escape rate of 3.9% versus 6.6% for single-issue PRs, an inconclusive gap (Fisher’s $p = 0.29$, ignoring PR clustering).

Generalization gap (H5). A patch may pass its own tests but fail held-out ones [36], as some fixes in both factories did (O4). Bug-hunting agents already act as held-out tests (O6).

TABLE II
PLANNED EXPERIMENTS AT EQUAL COST. H1–H4 TEST BOTH FACTORIES,
AND H5 ONLY THE CLI FACTORY.

Hyp.	Comparison (unit)	Prediction
H1	Specialists vs. generalist (area)	Higher min. N_i/target
H2	Adaptive vs. fixed rates (week)	Higher velocity at equal cost
H3	Up to 5 vs. 1 issue/PR (root cause)	More fixes/review hour, same escapes
H4	Recent vs. all vs. no notes read (run)	Recent notes, highest velocity
H5	Held-out vs. standard gate (CLI PR)	Fewer escapes, slowdown $\leq \delta$

H5 tests whether held-out tests, run before merge outside G and hidden from the agents they judge, reduce escapes more than agent-modified CI checks (O3).

VI. FUTURE PLANS

Software engineering uses case studies to develop and test hypotheses [23], as Mockus et al. did with Apache and Mozilla [37]. Over five months, we will evolve both factories and restart the agentic swarm. For *feasibility*, we will run the bootstrap on new repositories, comparing setup effort, control-test failures, and time to the first E^* -validated merge with manual setup. For *sufficiency*, we will record whether operators must add coordination, such as an orchestrator or a work database, to sustain N_i for a month. **H5**, our central experiment, tests the evaluator channel. **H1–H4** score arms by N_i , which discounts agent-filed value and unverified merges, so gains cannot come from inflation.

Learn within each factory. Each experiment in Table II randomizes its own unit under existing governance at equal budgets. **H1** alternates each package manager or defect type between specialist and generalist arms across periods, so each area serves as its own control. **H2** randomizes schedule type within consecutive week pairs, modeling trend and carryover. **H3** groups issues by shared root cause before randomization, stratified by group size, with PR-clustered analysis. **H4** hides notes older than a week or all notes from a run without changing its tasks. **H5** randomizes CLI PRs, and arm-blind scouts judge escapes per fixed issue for 14 days after merge, with PR-clustered analysis and a preset velocity margin δ . Detecting a halved 5.6% escape rate at 80% power and $\alpha = 0.05$ needs 800+ fixes per arm, about three months at late-study rates, more under clustering.

Generalize across factories. Settings that hold in both factories and, by architectural similarity [38], in factories with public boards [19], [20] become candidate best practices.

Domain-specific or general. Prior software factories targeted particular product families or domains [39], [40]. On DARPA’s AIXCC tasks, specialized systems fixed bugs missed by general agents [41]. Since no optimizer performs best on every problem class [42], we will test whether lessons from our security factories transfer to other domains. Generalizable settings can become bootstrap defaults, and others setup questions.

Self-tuning and evolution. Like an experience factory [43], an *adaptation skill* will let the manager tune settings from cost, correctness, and delivery within governance [44]–[46].

DATA AVAILABILITY

The artifact [12] contains the bootstrap and will include its revisions and experimental scripts. Case records remain private for confidentiality.

REFERENCES

- [1] H. Li, H. Zhang, and A. E. Hassan, “AIDev: Studying AI coding agents on GitHub,” in *Proc. 23rd Int. Conf. Mining Softw. Repositories (MSR)*, 2026, pp. 1029–1033.
- [2] M. A. Cusumano, “The software factory: A historical interpretation,” *IEEE Softw.*, vol. 6, no. 2, pp. 23–30, 1989.
- [3] A. Gray, “Minions: Stripe’s one-shot, end-to-end coding agents—Part 2,” stripe.dev blog, Feb. 2026. [Online]. Available: <https://stripe.dev/blog/minions-stripe-one-shot-end-to-end-coding-agents-part-2>
- [4] Z. Bruggeman, J. Quense, and R. Sengottuvelu, “Why we built our own background agent,” Ramp Builders Blog, Jan. 2026. [Online]. Available: <https://engineering.ramp.com/post/why-we-built-our-background-agent>
- [5] M. Charas and M. Bruggmann, “1,500+ PRs later: Spotify’s journey with our background coding agent (Honk, Part 1),” Spotify Engineering Blog, Nov. 2025. [Online]. Available: <https://engineering.atspotify.com/2025/11/spotify-background-coding-agent-part-1>
- [6] U. K. Medisetty, “Running a software factory efficiently at Uber scale,” Uber Engineering Blog, Aug. 2026. [Online]. Available: <https://www.uber.com/us/en/blog/efficient-software-factory/>
- [7] C. Qian *et al.*, “ChatDev: Communicative agents for software development,” in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguist. (ACL)*, 2024, pp. 15174–15186.
- [8] S. Hong *et al.*, “MetaGPT: Meta programming for a multi-agent collaborative framework,” in *Proc. 12th Int. Conf. Learn. Represent. (ICLR)*, 2024, pp. 23247–23275.
- [9] W. Lyu, Y. Xiao, Y. Zhang, and Y. Sun, “TheBotCompany: Self-organizing multi-agent systems for continuous software development,” in *Proc. 41st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2026, pp. 349–361.
- [10] J. C. Davis, “Model-based agentic engineering: A case study in software engineering,” Online book, 2026. [Online]. Available: <https://davisjam.github.io/model-based-agentic-software-engineering/book/mage-book/>
- [11] L. Gao, J. Schulman, and J. Hilton, “Scaling laws for reward model overoptimization,” in *Proc. 40th Int. Conf. Mach. Learn. (ICML)*, vol. 202, 2023, pp. 10835–10866.
- [12] M. Lysenko, “instant-factory: One prompt that turns a GitHub repo into a Claude Code routine software factory,” GitHub repository, 2026. [Online]. Available: <https://github.com/mikolajlysenko/instant-factory>
- [13] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. New York, NY, USA: Oxford Univ. Press, 1999.
- [14] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, “The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty,” *ACM Comput. Surv.*, vol. 12, no. 2, pp. 213–253, 1980.
- [15] S. Pal, F. Y. Wang, and M. J. Buehler, “SwarmWorld: Stigmergic technological evolution in societies of language-model agents,” arXiv preprint arXiv:2608.26081, 2026.
- [16] Y. Kim *et al.*, “Towards a science of scaling agent systems,” arXiv preprint arXiv:2512.08296, 2025.
- [17] S. Yegge, “Gas Town,” Project website, 2026. [Online]. Available: <https://yegge.ai/gastown>
- [18] GitHub Next, “Agentic workflows: Towards natural-language programming for GitHub Actions,” GitHub Next research prototype, Aug. 2025. [Online]. Available: <https://githubnext.com/projects/agentic-workflows/>
- [19] D. Syme, P. de Halleux, and M. Kiefer, “Welcome to Peli’s Agent Factory,” GitHub Agentic Workflows blog, Jan. 2026. [Online]. Available: <https://github.github.com/gh-aw/blog/2026-01-12-welcome-to-pelis-agent-factory/>
- [20] Z. Malguitou and L. Bichard, “We built a software factory in 10 days. Here’s what we learned,” Ona blog, Apr. 2026. [Online]. Available: <https://ona.com/stories/software-factory-what-we-learned>
- [21] W. Lin, “Scaling long-running autonomous coding,” Cursor blog, Jan. 2026. [Online]. Available: <https://cursor.com/blog/scaling-agents>
- [22] Anthropic, “Automate work with scheduled tasks,” Claude Code Docs, 2026. [Online]. Available: <https://code.claude.com/docs/en/routines>
- [23] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.
- [24] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *Proc. 11th Work. Conf. Mining Softw. Repositories (MSR)*, 2014, pp. 92–101.
- [25] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows,” in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, 2010, pp. 495–504.
- [26] R. Lopopolo, “Harness engineering: Leveraging Codex in an agent-first world,” OpenAI blog, Feb. 2026. [Online]. Available: <https://openai.com/index/harness-engineering/>
- [27] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, “How do fixes become bugs?” in *Proc. 19th ACM SIGSOFT Symp. Found. Softw. Eng. and 13th Eur. Softw. Eng. Conf. (ESEC/FSE)*, 2011, pp. 26–36.
- [28] J. Park, M. Kim, B. Ray, and D.-H. Bae, “An empirical study of supplementary bug fixes,” in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories (MSR)*, 2012, pp. 40–49.
- [29] E. Shihab *et al.*, “Studying re-opened bugs in open source software,” *Empir. Softw. Eng.*, vol. 18, no. 5, pp. 1005–1042, 2013.
- [30] D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan, “The impact of rapid release cycles on the integration delay of fixed issues,” *Empir. Softw. Eng.*, vol. 23, no. 2, pp. 835–904, 2018.
- [31] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.
- [32] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proc. 30th Int. Conf. Mach. Learn. (ICML)*, vol. 28, 2013, pp. 1139–1147.
- [33] O. Sener and V. Koltun, “Multi-task learning as multi-objective optimization,” in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, vol. 31, 2018.
- [34] C. Coleman *et al.*, “Analysis of DAWNbench, a time-to-accuracy machine learning performance benchmark,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, pp. 14–25, 2019.
- [35] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015, arXiv:1412.6980.
- [36] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? Overfitting in automated program repair,” in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 532–543.
- [37] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.
- [38] R. Wieringa and M. Daneva, “Six strategies for generalizing software engineering theories,” *Sci. Comput. Program.*, vol. 101, pp. 136–152, 2015.
- [39] J. Greenfield and K. Short, “Software factories: Assembling applications with patterns, models, frameworks and tools,” in *Companion 18th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2003, pp. 16–27.
- [40] A. Pavlenko *et al.*, “Towards an AI software factory for data systems,” arXiv preprint arXiv:2609.36323, 2026.
- [41] C. Zhang *et al.*, “SoK: DARPA’s AI Cyber Challenge (AICC): Competition design, architectures, and lessons learned,” in *Proc. 35th USENIX Secur. Symp. (USENIX Security)*, 2026, pp. 2705–2724.
- [42] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, 1997.
- [43] V. R. Basili, G. Caldiera, F. McGarry, R. Pajarski, G. Page, and S. Waligora, “The Software Engineering Laboratory: An operational software experience factory,” in *Proc. 14th Int. Conf. Softw. Eng. (ICSE)*, 1992, pp. 370–381.
- [44] Y. Lee, R. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn, “Meta-Harness: End-to-end optimization of model harnesses,” in *Proc. 3rd Conf. Lang. Model. (COLM)*, 2026.
- [45] J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune, “Darwin Gödel Machine: Open-ended evolution of self-improving agents,” in *Proc. 14th Int. Conf. Learn. Represent. (ICLR)*, 2026, pp. 104223–104294.
- [46] S. Hu, C. Lu, and J. Clune, “Automated design of agentic systems,” in *Proc. 13th Int. Conf. Learn. Represent. (ICLR)*, 2025, pp. 21344–21377.