Hephaestus: A Supervised Agent for AI-Driven Software Development

Diego Camarinha
Irya Solutions
São Paulo, Brazil
diego@iryasolutions.com.br

Abstract

Developers increasingly rely on generative AI to write code, but production use raises a harder question than adoption: how much to delegate, and how far to trust the delivered code? Delegating design and architectural decisions without oversight risks eroding long-term code quality and accruing technical debt. The challenge is to find a working boundary, one where AI handles implementation while developers stay focused on business logic, strategy, and architecture. Hephaestus is a supervised autonomous agent designed to operate within that boundary. Triggered by a developer-assigned label on an issue, it takes over the development cycle, from context gathering to merge request, following a human-in-the-loop methodology. The agent applies test-driven development by default and pauses at structured approval gates: after gathering codebase context it proposes a test plan, and then an implementation plan, waiting for developer approval at each before any code is written. Between gates it works autonomously, implementing, running tests, monitoring CI, and opening the merge request, so developers re-enter at code review through their standard process. We evaluated Hephaestus across 29 issues from six production-deployed projects spanning different programming languages. Re-implementing 10 issues the team had already completed, which allows direct comparison, the agent reached 90% plan acceptance, matched the original solution in 80% of cases, and reduced labor time by 78% at an average AI cost of USD 5.05 per issue. On 19 new production issues, plan acceptance was 84% and 89% of the changes were merged, 32% of them without edits, at USD 6.17 per issue. Developer satisfaction averaged 3.45 out of 5, improving substantially as project-specific context matured.

Keywords

Artificial Intelligence, Human-in-the-Loop, Software Development, Autonomous Agents, Code Generation

1 Introduction

AI-assisted development has moved from novelty to mainstream. A 2026 survey of over 900 software engineers found that 95% use AI tools weekly, with 75% relying on them for at least half of their engineering work [6], and the Stack Overflow Developer Survey corroborates the trend: 84% of developers either use or plan to use AI tools [7]. Adoption is no longer the challenge.

The harder problem is trust and delegation. As the same Stack Overflow data shows, only 33% of developers trust AI-generated output for accuracy, while 46% actively distrust it [7]. We argue that delegating design decisions and architectural choices to AI without oversight introduces a compounding risk: generated code that appears functional in isolation may erode long-term code quality, increase coupling, or accumulate technical debt that surfaces only under future maintenance.

This paper presents Hephaestus, a supervised autonomous agent that addresses this need by automating the full development cycle, from being assigned to an issue to opening the merge request, under a human-in-the-loop (HITL) methodology. Rather than producing code for developers to review in bulk, Hephaestus externalizes its planning step, requiring explicit developer approval before implementation begins.

2 Hephaestus

Hephaestus is a web application that integrates with a GitLab issue tracker (chosen because Irya Solutions operates a self-hosted GitLab instance across all its projects) and uses Claude Code [2] as the AI harness, the execution layer that interfaces with the underlying large language models. The current setup runs on Anthropic's models (Opus 4.6, Opus 4.7, and Sonnet 4.6), chosen for their strong performance on SWE-bench Verified [5], but the AI layer is abstracted, making it straightforward to replace or extend with other models. Each issue is processed through a sequential state machine with well-defined phases: context gathering, test planning and review, implementation planning and review, implementation, CI monitoring, and merge request creation. By default, the agent applies test-driven development: tests are planned, approved, and implemented before the production code is planned. Projects without a test suite can disable this and have the agent plan the implementation directly. The state machine ensures that no phase begins until the previous one completes, either by the agent or by the developer. The agent operates within a dedicated git worktree per issue, isolating each task's changes from the main branch and from concurrent work on other issues. Execution is isolated as well: for each issue, Hephaestus brings up a set of containers declared by the project in a Docker Compose file in its own repository, with the development environment and required services already provisioned. Every command the agent runs, including test suites and linters, executes inside that disposable environment, much as modern CI systems provide per-job environments. Each project can therefore define its own toolchain without the host having to satisfy it.

2.1 Human-in-the-Loop Workflow

Figure 1 illustrates the complete workflow. Hephaestus embeds human judgment at the points in the pipeline where errors are most consequential, following the HITL design principle of integrating human oversight into automated systems at high-stakes decision boundaries [1]. There are four structured developer touchpoints:

(1) **Triggering.** The developer assigns a specific label to the issue, explicitly delegating it to the agent. This preserves intentional control over which work is automated.
(2) **Test plan approval.** Because Hephaestus follows TDD, the first plan it produces is a test plan. After gathering codebase context, the agent proposes the tests that specify the desired behavior and pauses until the developer approves or requests a revision. Approving the plan settles the behavior the change should exhibit before any code is written.
(3) **Implementation plan approval.** With the tests in place, the agent generates a plan for the production code and again pauses for revision and approval. This gate catches a misunderstood requirement or an architecturally inappropriate approach before the implementation is written.
(4) **CI fix approval.** If the CI pipeline fails after implementation, the agent analyzes the failure, proposes a fix plan, and again pauses for developer approval before applying changes. Intermittent infrastructure failures are retried automatically without developer involvement.

Between these touchpoints the agent is fully autonomous: it implements the approved tests and the production code, runs them, monitors the CI pipeline, and opens the merge request. Developers re-enter at code review through the standard process.

The cycle closes with a retrospective. Once the merge request is merged or closed, Hephaestus looks for failure signals: rejected plans, CI fixes it had to apply, and commits a human pushed on top of its branch. If any are present, it traces them to their root causes using the issue and review discussions, then proposes additions to the project's context files, written as rules it will read on future issues, and opens a separate merge request once the developer approves the report. Corrections made during review thus become the agent's learning signal, so improving its context depends less on the developer noticing and writing down what was missing.

This workflow is consistent with the approach proposed by Takerngsaksiri et al. [8], who found that requiring human approval of agent-generated plans increases developer confidence and reduces rework in AI-assisted development. Hephaestus extends this model with a separate approval gate for the test plan, so the specification is agreed before code is written, and by also gating CI fix decisions, keeping developers informed of failures rather than allowing the agent to silently modify code to pass tests. Because the system controls prompt construction and context selection at each phase, token use stays purposeful, keeping the per-issue AI cost low.

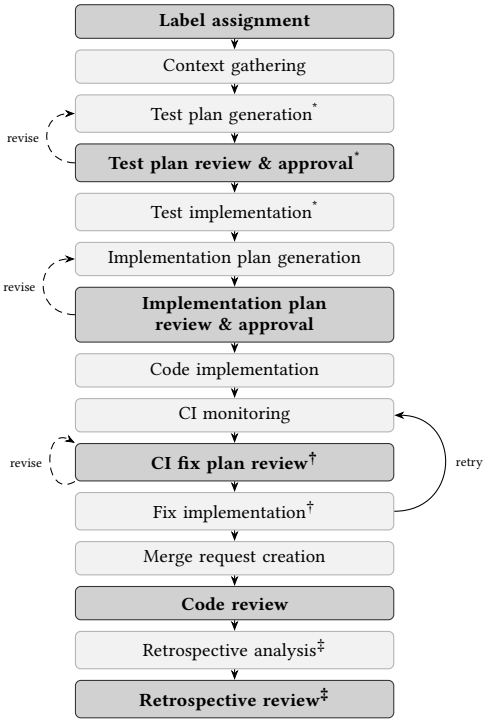

Figure 1: Hephaestus workflow. Dark boxes: human touchpoints; light boxes: autonomous agent steps. Dashed arrows: revision loops taken when the developer rejects a plan or fix. * Skipped for projects configured without a test suite. † Conditional on CI failure. ‡ Conditional on failure signals in the completed issue.

2.2 Comparison With Existing Tools

AI development tools differ mainly in how much of the workflow they automate and where the developer intervenes. Inline assistants such as Cursor operate at the level of the individual edit, accelerating a developer who stays in control suggestion by suggestion. A more recent class of agents automates the whole task. Commercial issue-to-merge agents, such as GitHub Copilot's coding agent [4] and Atlassian's Rovo Dev [3], take a tracked issue, work autonomously, and open a pull request for review, while open research systems, such as SWE-agent [9], demonstrate comparable end-to-end autonomy on benchmarks like SWE-bench [5], resolving GitHub issues by generating code patches. Hephaestus shares this issue-to-merge scope but differs in where human judgment enters: in each case above, the earliest structured decision point comes once the code already exists, which is where the trust and delegation concerns that motivate this work arise. Hephaestus instead inserts structured approval gates before implementation. The closest prior work, HULA [8], gates the plan before coding; Hephaestus extends this with a separate test-plan gate and a CI-fix gate, distributing oversight across four touchpoints while keeping the agent autonomous between them. It also makes TDD a gated step, approving the test plan, which serves as the specification, before any implementation is planned, a capability the other tools do not enforce. A further difference comes after the merge request: those tools treat each issue as an isolated run, whereas the retrospective feeds the outcome of human review back into the agent's context files, turning oversight into a persistent improvement loop.

3 Validation

We evaluated Hephaestus in two stages across 29 issues from six production-deployed projects spanning different programming languages. In Stage 1 (historical validation), 10 issues previously completed by the development team across three projects were reimplemented by the agent. The agent worked from the codebase

as it stood before the original change and had no access to the human implementation, so each solution was produced independently. This enabled direct comparison against real implementation data: difficulty classification and baseline time were drawn from the original deliveries, and code similarity between agent-generated and original code was rated on the 1–4 scale of Takerngsaksiri et al. [8], where 1 denotes no similarity, 2 weak similarity, 3 high similarity, and 4 highly similar. Scores were assigned automatically by the AI harness (Claude Opus 4.6), prompted to rate how similar the agent's changes were to the human developer's, each taken as a diff against the shared base branch. This makes grading reproducible, but the grades come from an automated assessment rather than independent human review. We interpret the two high levels as follows: a 3 marks the same overall approach with minor divergences, most often the agent adding extra tests or edge cases, while a 4 marks output differing from the human solution only in cosmetic details such as variable naming or loop style. In Stage 2 (production validation), 19 new issues not previously implemented by the team were drawn from four projects; developer estimates served as the time baseline and merge outcome was tracked. Across both stages, per-issue metrics included plan acceptance on the first attempt and AI cost. AI cost is the amount charged by the AI provider's API for each run, priced per token and measured in April and May 2026; Hephaestus adds no fee of its own. For the cost comparison, developer effort is valued at USD 40 per hour.

The projects evaluated are production systems written in TypeScript and JavaScript (React, React Native), Ruby (Ruby on Rails), and C#, spanning customer portals, admin backends, content sites, and interactive web apps, several targeting both web and mobile. The issues cover copy and configuration changes, bug fixes, new UI components, API integrations, features with non-trivial business logic, and cross-cutting refactors. By difficulty, Stage 1 comprised 3 simple, 4 medium, and 3 hard issues; of Stage 2's 19 issues, 4 were simple, 5 medium and 10 hard. Effort was counted in hours; the reviewing developers were the projects' own team rather than independent evaluators, and they tracked the time spent reviewing the agent's merge requests.

3.1 Stage 1: Historical Validation

Table 1 summarizes per-project results for the 10 historical issues. The agent achieved a 90% first-plan acceptance rate. Generated code reached high similarity levels compared to the original implementations (scores of 3 or 4) in 80% of cases. Human review time totaled 10 hours against 45 hours of original implementation effort, a 78% reduction in labor time, at an average AI cost of USD 5.05 per issue. Valued at USD 40 per hour, that is USD 1,800 in developer time for the manual deliveries versus roughly USD 450 with Hephaestus (USD 400 of review plus about USD 50 of API cost), a 75% reduction in delivery cost.

3.2 Stage 2: Production Validation

Stage 2 processed 19 new production issues across four projects (Table 2). Per issue, we additionally recorded the merge outcome and developer satisfaction with the quality of the generated code on a 1–5 scale. Both were provided by the single developer who reviewed that merge request, so every change was evaluated by exactly one rater. The merge outcome fell in one of three categories: *direct*, when the agent's merge request was merged with no changes; *with edits*, when it was merged after the reviewer modified the generated code; and *discarded*, when the output was rejected and the issue was implemented by a developer instead. We refer to the first two categories jointly as *usable output*, meaning output that reached the main branch, with or without correction.

Table 1: Stage 1 per-project results. Plan Acc.: first-plan acceptance rate; Code Sim.: mean code similarity (1–4 scale); Human (h): original implementation hours; Review (h): hours spent reviewing the agent's output; Cost: average AI cost per issue (USD).

Project	Issues	Plan Acc.	Code Sim.	Human (h)	Review (h)	Cost
Project A	3	100%	2.7	7.5	2.0	6.16
Project B	4	75%	2.8	18.0	4.0	4.02
Project C	3	100%	3.3	19.5	4.0	5.31
Total	10	90%	2.9	45.0	10.0	5.05

Table 2: Stage 2 per-project results (satisfaction on 1–5 scale; avg. cost in USD)

Project	Issues	Direct	Edited	Disc.	Sat.	Cost
Project 1	1	0	1	0	2.0	14.62
Project 2	2	0	0	2	0.0	8.81
Project 3	4	0	4	0	3.4	4.30
Project 4	12	6	6	0	4.2	5.64
Total	19	6	11	2	3.45	6.17

The agent achieved an 84% first-plan acceptance rate and a 32% direct-merge rate (6 of 19 issues). Counting merge requests that were merged after reviewer edits, 17 of 19 issues (89%) produced usable output. Total human review time was 66 hours against a pre-task estimate of 123 hours, a projected 46% reduction in labor time, at an average AI cost of USD 6.17 per issue.

The per-project breakdown reveals a clear maturity pattern. The earliest projects (Projects 1 and 2) suffered from incomplete context configurations, leading to higher costs, discarded outputs, and low satisfaction scores. After developers refined the projects, results improved greatly, and that improvement came from project-level work rather than from modifying the agent: they made each project's `CLAUDE.md` context file more specific to Hephaestus, corrected the per-project container configuration so builds and CI ran reliably, and, for later issues, did light upfront preparation such as researching common pitfalls before assigning the work. Project 4, the most mature project, with 12 issues evaluated, achieved a 50% direct-merge rate, an average satisfaction score of 4.2 out of 5, and a drop in reviewed effort from an estimated 72 hours to 10.5 actual hours, roughly an 85% reduction against the aggregate 46%. The satisfaction average of 0.0 for Project 2 reflects two issues that were fully discarded; one was later found to be mathematically impossible to implement as specified, not a failure of the agent per se.

Those results support a key practical finding: agent performance scales with the quality of the project context provided, so the initial

investment in configuring it pays off over subsequent issues. This is what the retrospective step is meant to sustain, keeping that iteration from resting entirely on the developer.

A few task types proved harder for the agent. The clearest is the large cross-cutting refactor, where success hinges on picking the right abstraction: here the agent tended to spread broad, repetitive edits across many files instead of the compact extraction a developer would write, and one such refactor scored low on similarity. UI work with strong visual or cross-platform requirements was another, producing functional but visually incomplete results that needed guidance. The agent also struggled when a task hinged on unwritten domain knowledge absent from the codebase and context files, and it did not push back on weakly specified or infeasible work, generating plausible but low-value code instead. Well-scoped work with clear specifications fared best, including a hard, domain-heavy feature on which the agent independently reached essentially the human solution.

Hephaestus is also in active production use at Irya Solutions beyond this evaluation, providing continued battle-testing in real development environments. Its cost advantage reaches the customer as well: on Project 4, a short-scope client project delivered end to end by the agent, the actual cost (developer review time plus AI usage) came in well below the estimate-based budget, letting Irya deliver the project at 20% lower cost while remaining profitable.

4 Conclusion And Future Work

Hephaestus demonstrates that autonomous code development is viable in real production environments when human oversight is embedded at planning, failure-recovery, and review stages. The HITL design keeps developers in control of architectural decisions without requiring them to micromanage implementation, with a measured 78% reduction in labor time against historical baselines and a projected 46% on new issues, both at a low per-issue AI cost.

These results come with a few limitations. Chief among them, the evaluation has no control baseline: the agent is compared against human-only development in Stage 1 and against developer estimates in Stage 2, not against an alternative agent configuration or workflow. We therefore do not claim the gains follow specifically from the human-in-the-loop design, since a more autonomous agent might reach comparable numbers under the same conditions. What the study supports is that the HITL workflow is effective in production, not that it outperforms less-supervised alternatives; a controlled comparison, for instance running the same issues through two developer groups, one using the agent and one not, is a natural next step. The baselines also carry caveats. In Stage 1 the reviewers were already familiar with the original solution, so their review was likely faster than a fresh reviewer's, which may overstate the reduction. Stage 2 compares against developer estimates rather than measured effort, which may skew the baseline in either direction. Code similarity was graded by an LLM rather than by independent human review, and satisfaction is a single-rater score. Finally, the scale is modest: 29 issues across six projects surfaces trends rather than statistically robust conclusions. The evaluation also does not measure the long-term risks that motivate the work, such as technical debt or architectural erosion, which we leave to future work.

The productivity gains concentrate where project context is most mature, as Project 4's results show. Part of the reason is TDD: in mature TDD codebases much of the code written is test code, so delegating it to the agent removes a large share of routine implementation work. This does not become a proportional productivity multiplier, however, because the generated tests still have to be read and approved. Tests encode the specification, so reviewing them absorbs much of the developer's remaining effort. Beyond these figures, Hephaestus points toward a broader shift in the developer's role: one where architecture, system design, and business logic become the primary focus, while translating those decisions into working code is left to AI agents.

Hephaestus is open source and welcomes contributions from the community. The near-term roadmap focuses on broadening platform support beyond GitLab to GitHub and Bitbucket, wiring the merge request feedback loop so the agent can address reviewer comments autonomously, and expanding the evaluation dataset. Further out, the planned work includes parallel sub-task execution within a single issue (currently each issue is a single sequential thread) and multi-LLM support, allowing teams to configure alternative models as execution engines. In the longer term, the goal is to instrument the full lifecycle impact of agent-generated code: how it affects code quality metrics (for example, cyclomatic complexity or code duplication), technical debt accumulation, and maintenance burden over time, providing empirical grounding for the delegation boundary that Hephaestus is designed to enforce.

Artifact Availability

The source code and the full validation dataset for both stages, including per-issue metrics and developer comments, are publicly available at https://gitlab.com/irya/hephaestus. The validation data used in this paper is under `doc/validation`.

References

[1] Saleema Amershi, Maya Cakmak, William Bradley Knox, and Todd Kulesza. 2014. Power to the People: The Role of Humans in Interactive Machine Learning. *AI Magazine* 35, 4 (2014), 105–120. doi:10.1609/aimag.v35i4.2513

[2] Anthropic. 2025. Claude Code. Anthropic. https://claude.ai/code Accessed: 2026-05-21.

[3] Atlassian. 2025. Rovo Dev: Agentic AI for Software Teams. Atlassian. https://www.atlassian.com/software/rovo-dev Accessed: 2026-07-15.

[4] GitHub. 2025. GitHub Copilot: Meet the New Coding Agent. The GitHub Blog. https://github.blog/news-insights/product-news/github-copilot-meet-the-new-coding-agent/ Accessed: 2026-07-15.

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2310.06770

[6] Gergely Orosz and Elin Nilsson. 2026. AI Tooling for Software Engineers in 2026. The Pragmatic Engineer. https://newsletter.pragmaticengineer.com/p/ai-tooling-2026 Accessed: 2026-05-21.

[7] Stack Overflow. 2025. 2025 Stack Overflow Developer Survey. Stack Overflow. https://survey.stackoverflow.co/2025 Accessed: 2026-05-21.

[8] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-In-the-Loop Software Development Agents. In *Proceedings of the 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Ottawa, ON, Canada, 342–352. doi:10.1109/ICSE-SEIP66354.2025.00036

[9] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*. Curran Associates, Inc., Red Hook, NY, USA. http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html