

It Is Not My Code Anymore

Augusto Camargo

Institute of Mathematics, Statistics and Computer Science (IME)

University of São Paulo, São Paulo, Brazil

7 September 2026

A research note on authorship, responsibility, and evaluation in AI-assisted software production. The discussion uses a hypothetical failure and a selective reading of the literature; it reports no new empirical results.

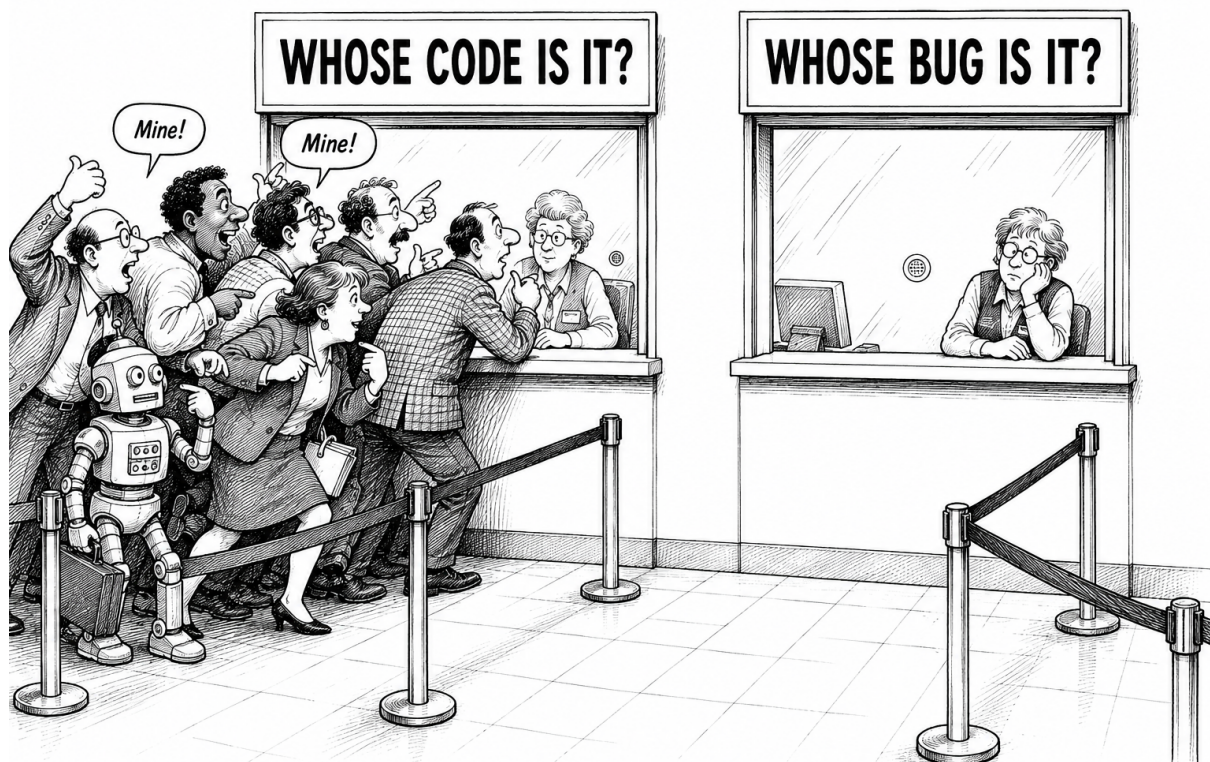


Figure 1: Whose code, whose bug? A satirical contrast between claiming authorship and accepting responsibility after failure, not an empirical finding of the cited studies.

Suppose a large language model (LLM) accidentally introduces `1 / 0` while implementing a feature requested by a developer. The developer reads the diff without paying close attention and approves the change. Evaluating the expression raises an unhandled exception. During enrollment, the program reaches the expression and fails. A student cannot enroll.

Who wrote that expression?

The developer may have specified the objective, selected the model, guided its revisions, and accepted its output. In this example, however, the model produced the expression. Reviewing it did not change who produced it.

Responsibility for the failure is a separate question. We still need to examine the review, the release process, the harness that coordinates model and tool use, and the organization operating the service. Knowing who produced the expression does not tell us who should have caught the defect or who must restore the service.

I consider what a developer means by *my code* when a model produces it, and how shared ownership relates to responsibility after failure. I then ask whether the task needs a separate

program at all. Throughout, the practical requirement is the same: the student must be able to enroll.

Authorship: who produced the code?

Calling a vehicle “my ride” may mean that I use it to get somewhere. It does not establish that I designed it, built it, or own it. Similarly, “my code” can refer to code I wrote, inherited, maintain, or am expected to repair. The possessive alone does not distinguish these relationships. Taking over maintenance does not make someone the author of an existing expression. Generative models make this familiar ambiguity harder to ignore.

Seo, Deldari, and Mentis examine these relationships in *Whose Code Is It?* In a study with 30 participants, increasing AI contribution reduced perceived possession and shifted attribution toward AI. Willingness to accept accountability for production systems remained comparatively stable [1]. Draxler and colleagues report a related distinction in text generation: users can feel little ownership of generated text while publicly declaring themselves its authors [2]. Feeling ownership, claiming authorship, and accepting responsibility can be separate judgments.

The human contribution still matters. Naur’s account of programming as theory building emphasizes understanding [3]; Meske and colleagues describe a shift toward orchestration through probabilistic intent mediation [4]. Directing an implementation and producing it are distinguishable contributions. An art director can shape a work without executing every mark. In the opening example, the developer directed the work and the model supplied the expression.

Reviewing a diff may fulfill an assigned duty and support a release decision. It does not make the reviewer the producer of an expression already in the diff. This is the sense in which I use “It is not my code anymore.” The phrase does not assign legal ownership to a model.

Review can change whether code is released without changing the code itself. Once the expression runs, the language and execution environment determine its behavior. A probabilistic generator can produce a fully reproducible defect. Calling it a hallucination adds little to the immediate investigation: how did the expression enter the program, and why did the resulting failure reach the student?

Responsibility after failure

Tholander and Jonsson describe programming with generative AI as a co-constituted practice in which intentions, authorship, and control develop through interaction [5]. That description is compatible with tracing a particular expression to a particular source. Several contributors can produce a service together even when the model introduced the defective expression.

Reconstructing who contributed to a program does not, by itself, determine who has which responsibilities after its failure.

Different production histories can yield the same program, and the recorded history may be incomplete. Even a complete record would leave questions about each participant’s duties and control over the process.

In the opening example, an investigation must establish who introduced the expression and which controls should have detected or contained it. It must also identify who owed the student a functioning service and who must restore it. Each question can have a different answer. Saying “we co-created it” leaves those answers unspecified.

Tracing the expression further back through model training and training-data production adds to its causal history. That history may help an investigation, but it does not assign the same responsibility to everyone involved. Assigning responsibility requires identifying a relevant contribution, an applicable duty, and evidence connecting the two.

Collective psychological ownership describes a group’s relationship to something it regards as its own [6]. After a failure, the team also needs clear duties: who can stop a release, who must question the acceptance evidence, and who investigates, repairs, and explains. Elish’s *moral*

crumple zone describes the risk of blaming a nearby human who had limited control [7]. Santoni de Sio and Mecacci distinguish several responsibility gaps rather than treating responsibility as a single issue [8]. Shared ownership alone does not resolve these gaps.

Would people attribute the code in the same way after a successful delivery and after a failure? Figure 1 asks this question through satire. An answer would require evidence about behavior after failure, beyond the relationships examined in the programming studies. Using ownership to organize production still leaves duties, controls, and investigation procedures to be specified. Release decisions and incident handling depend on these details. We therefore need to examine how model-generated implementation changes the production process.

We put the machine in the loop

People already specified, implemented, reviewed, released, and maintained software using machines. A generative model now joins that process as a producer of implementations. From this perspective, the change is *machine-in-the-loop*. Starting with the model and asking where to insert a human reverses the order in which the process developed.

Preventing the generated defect from reaching the student requires controls across production. Industrial history offers a precedent for reorganizing those controls when work is divided differently. The American Society for Quality (ASQ) links the division of craft work and Taylorist production to changes in inspection [9]. Shewhart’s control-chart work, dated to 1924 by the National Institute of Standards and Technology (NIST), shifted attention toward process variability [10]. Quality did not begin with mechanization, but its organization changed.

For generated software, examining production means considering the model, configuration, context, harness, tools, dependencies, and human interventions together. A model or harness change can alter production even when the requested function stays the same. Reviewing one output provides evidence about that output. Evaluating later outputs requires evidence from repeated production runs.

Collecting and using this evidence requires an assigned role. I would give quality engineering a central role: defining acceptance evidence, questioning the process, tracking failures, and supporting release decisions with explicit criteria. The role needs clear duties and decision authority. Software already has assurance disciplines; the question is how their responsibilities change when implementation is generated.

This work extends beyond reviewing the finished diff. U.S. Food and Drug Administration (FDA) guidance states that inspection and testing alone cannot assure pharmaceutical quality [11]. Applied to software production, the principle is to examine both the generating process and its outputs. Statistical evaluation can help when the populations and measurements are justified. A probabilistic generator does not make unrelated programs a homogeneous production batch.

Evaluating the service and its production

A process could consistently produce code that passes local checks while leaving the student unable to enroll. Repeatability alone would not establish that the service meets the demand. Acceptance criteria therefore need a reference beyond the producing process.

Inside-out evaluation starts with the producer, the code, and their properties, then extends toward the service. **Outside-in** evaluation starts with the required outcome and conditions of use, then identifies the evidence needed from each contributing system.

For enrollment, outside-in evaluation asks whether software production supports the required outcome. Component checks still provide evidence, but the demand determines what that evidence must establish. Figure 2 contrasts quality as a verification stage with a broader role in organizing assurance of engineering and operation, including model-based production.

The Café in Amsterdam asks whether the incumbent or the demand supplies the oracle [12]. Here, the incumbent is a view of software work centered on code. Adding review and oversight around the code preserves that starting point. Enrollment may also depend on academic records,

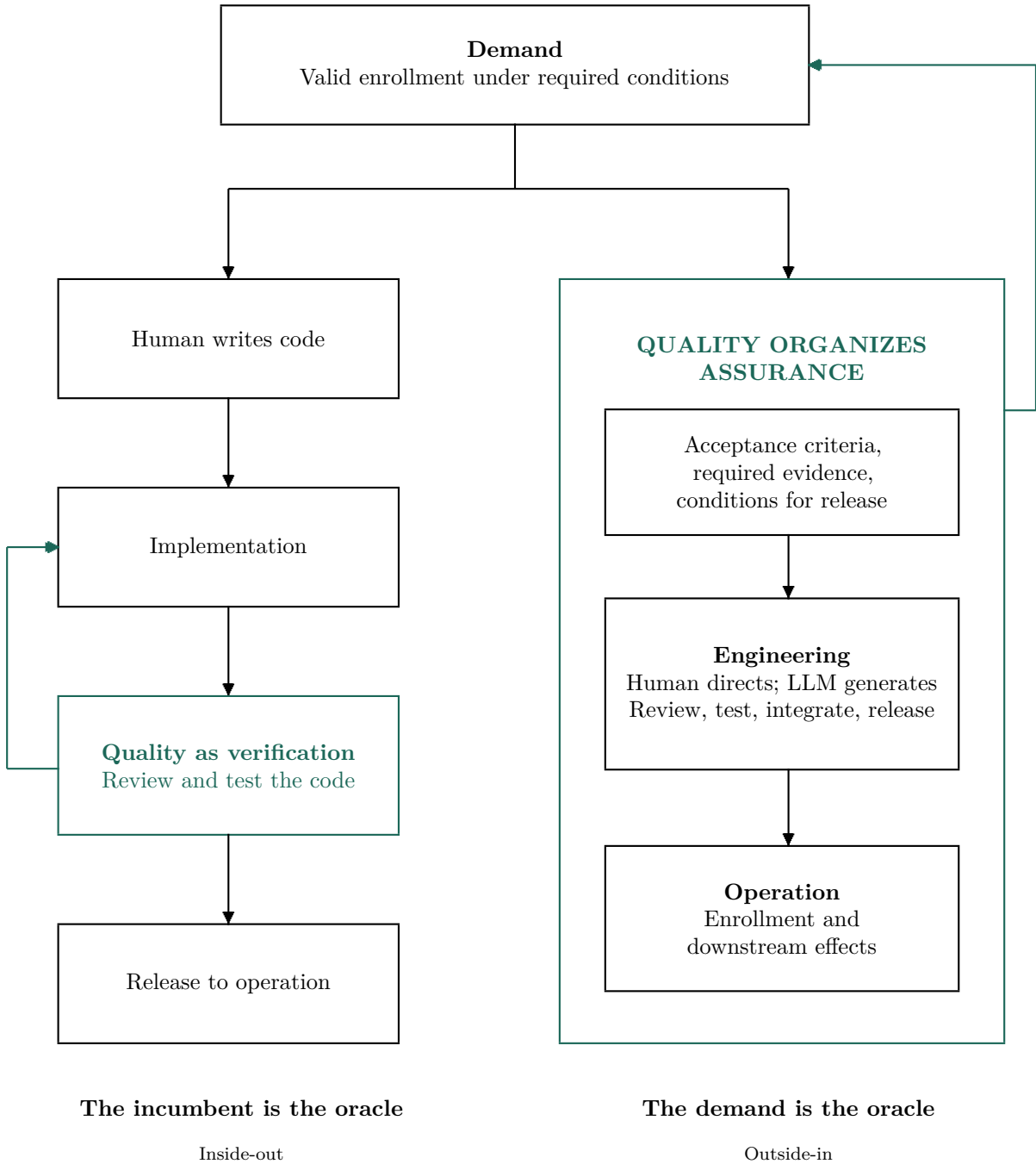


Figure 2: Both flows start with the demand. Green return arrows show what guides acceptance: the implementation on the left and the demand on the right. Black arrows show the forward flow. On the right, quality organizes assurance of engineering and operation using demand-derived criteria; the green boundary marks that scope. The comparison concerns the reference for acceptance. It is not an exhaustive history or an automatic consequence of using an LLM. Human-driven implementation can also be demand-centered.

payment, access rights, service recovery, and administrative action. The student’s demand crosses these boundaries.

Test-driven development can start before implementation while remaining within a software layer. Tests of individual operations can pass even when the enrollment journey fails. An oracle that checks only the confirmation screen can report success without a persistent enrollment record.

Testing the steel in a car’s body provides evidence about a material. A proving ground tests the assembled vehicle under required conditions. Both provide useful evidence, but they evaluate different objects against different criteria. The distinction is independent of whether testing is automated or performed early.

A software proving ground could exercise the enrollment service through user journeys: different legitimate paths, concurrent requests, interruptions, and recovery attempts. Its oracle would check valid enrollment and required downstream effects, not simply responsiveness. A thousand simulated users help only if their behavior represents the relevant conditions. Repeating one synthetic behavior a thousand times does not establish that match. Table 1 gives examples of evidence required beyond successful component behavior.

Table 1: Component evidence and acceptance evidence for the enrollment service.

Evidence about a part	Evidence required by the demand
The generated change passes its tests	The eligible student obtains a valid enrollment
The service responds under load	Enrollment completes within the required conditions and time
The retry handler executes	Retrying causes no duplicate enrollment or charge
The interface reports success	The academic record and required access confirm that success

The acceptance criteria in Table 1 need to be justified against the demand. Deriving every expected result from the generated implementation would make the evaluation depend on the system being evaluated.

The same criteria can evaluate a delivery and the process that produced it. Artifact evaluation asks whether this delivery works. Process evaluation asks how consistently a recorded production setup delivers acceptable results. To evaluate the process, produce multiple implementations under recorded configurations and evaluate them against the same externally justified criteria.

Representative operating scenarios and deliberately adversarial scenarios serve different purposes. Failures found by actively searching for problems do not directly estimate failure frequency in ordinary use.

Does the task need a separate program?

Evaluating against the demand allows us to compare different ways of providing the service. If acceptance criteria specify what a valid enrollment requires, they need not assume a particular implementation. We can therefore ask whether meeting those criteria requires generating a program specifically for the task.

Three possibilities need to be distinguished. A program can exist while its code is hidden from the user. Code can also be generated dynamically as the task is performed. In both cases, an implementation is still produced. Hassan and colleagues envision intent-centric development with code secondary and hidden by default [13]. Rost explores computing through reflective interaction, including dynamically generated code [14]. These changes concern how people interact with code and when it is produced; they do not necessarily remove it.

The third possibility is that an existing trained system performs the task without generating a new program for it. Welsh’s vision of trained systems replacing much conventional software motivates this possibility [15]. This does not mean dispensing with software: the trained system

still depends on an implementation and operating infrastructure. The question is whether each task needs an additional, separately produced program. Generating machine code instead of source code would still produce a task-specific program.

For enrollment, the distinction is between generating a program to carry out the required operations and having an existing system carry them out without generating that program. Whether the latter can meet the demand is an evaluation question, not an assumption. It would need to satisfy the same acceptance criteria, including a valid enrollment record and the required downstream effects, under its actual operating conditions.

Where this substitution is feasible, there is no newly generated program whose authorship needs to be assigned. Responsibility for providing the service remains, including responsibility for the system’s dependencies and failures. The question about ownership of generated code thus gives way to the same practical question that guided the evaluation: does the chosen means of delivery meet the demand?

Limitations and conclusion

The division by zero is a hypothetical example, not an incident report. The cited studies do not show that collective ownership collapses after failure, nor do they claim to certify safe delivery. I ask how the relationships they describe inform release decisions and incident handling.

A proving ground covers selected conditions of use and can miss relevant ones. Its scenarios and acceptance criteria need justification against the conditions in which the service is required. Observation after deployment is needed to detect omissions and revise the tests.

In the example, the model produced the expression, a production process accepted it, and the student experienced the failure. Distinguishing these facts lets us ask who must act, what they must do, and what evidence is needed before the next release.

“It is not my code anymore” changes how I describe my relationship to the implementation. Calling the code ours, or replacing the program with another means of delivery, still leaves the service to be evaluated. The student must be able to enroll.

References

- [1] J. Seo, E. Deldari, and H. M. Mentis, “Whose Code Is It? How AI Autonomy Reshapes Ownership, Responsibility, and Disclosure in AI-Assisted Programming,” in *Proceedings of the 31st International Conference on Intelligent User Interfaces (IUI)*, 2026, pp. 393–425. doi:10.1145/3742413.3789121.
- [2] F. Draxler, A. Werner, F. Lehmann, M. Hoppe, A. Schmidt, D. Buschek, and R. Welsch, “The AI Ghostwriter Effect: When Users Do Not Perceive Ownership of AI-Generated Text but Self-Declare as Authors,” *ACM Transactions on Computer-Human Interaction*, vol. 31, no. 2, art. 25, 2024. doi:10.1145/3637875.
- [3] P. Naur, “Programming as theory building,” *Microprocessing and Microprogramming*, vol. 15, no. 5, pp. 253–261, 1985. doi:10.1016/0165-6074(85)90032-8.
- [4] C. Meske, T. Hermanns, E. von der Weiden, K.-U. Loser, and T. Berger, “Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda,” *IEEE Access*, vol. 13, pp. 213242–213259, 2025. doi:10.1109/ACCESS.2025.3645466.
- [5] J. Tholander and M. Jonsson, “Vibe Coding Entanglements—Repositioning Boundaries of Intention, Authorship, and Responsibility in Programming with Generative AI,” in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, art. 90, pp. 1–16, 2026. doi:10.1145/3772318.3791847.
- [6] J. L. Pierce and I. Jussila, “Collective psychological ownership within the work and organizational context: Construct introduction and elaboration,” *Journal of Organizational Behavior*, vol. 31, no. 6, pp. 810–834, 2010. doi:10.1002/job.628.
- [7] M. C. Elish, “Moral Crumple Zones: Cautionary Tales in Human-Robot Interaction,” *Engaging Science, Technology, and Society*, vol. 5, pp. 40–60, 2019. doi:10.17351/ests2019.260.
- [8] F. Santoni de Sio and G. Mecacci, “Four Responsibility Gaps with Artificial Intelligence: Why they Matter and How to Address them,” *Philosophy & Technology*, vol. 34, no. 4, pp. 1057–1084, 2021. doi:10.1007/s13347-021-00450-x.
- [9] American Society for Quality, “History of Quality.” [ASQ online resource](#). Accessed 7 September 2026.
- [10] NIST/SEMATECH, “How did Statistical Quality Control Begin?” *e-Handbook of Statistical Methods*, sec. 6.1.1. [NIST online handbook](#). Accessed 7 September 2026.

- [11] U.S. Food and Drug Administration, *Process Validation: General Principles and Practices*, guidance for industry, January 2011. [FDA guidance](#).
- [12] A. Camargo, “The Café in Amsterdam: When the Incumbent Becomes the Oracle,” arXiv:2607.13393v3, 2026. arxiv.org/abs/2607.13393v3.
- [13] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, and Z. M. Jiang, “Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap,” *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 9, pp. 1–25, 2026. doi:10.1145/3807901.
- [14] M. Rost, “Co-Disclosing the Computer: LLM-Mediated Computing through Reflective Conversation,” in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2026. doi:10.1145/3772318.3791769.
- [15] M. Welsh, “The End of Programming,” *Communications of the ACM*, vol. 66, no. 1, pp. 34–35, January 2023 (published online December 2022). doi:10.1145/3570220.