The Unattempted Alternative:

AI, Engineering Complexity, and the Expansion of Feasible Work

Flyxion
Independent Researcher

October 2026

Abstract

A recent anecdote from an online programming forum describes a company whose AI-generated chatbot grew into a nine-container system with a self-repairing maintenance loop, hundreds of generated documents, and thousands of tests, and which the narrator takes as proof that agentic development cannot produce production software. The account is weak as a general indictment, since it supplies no counterfactual, no statement of requirements, and no measurement of the work accomplished. It is stronger as a description of a governance failure, in which automation became the object the application served and the procedures that certified correctness were produced by the process they were meant to check. This essay separates the intensive margin of technological change, where a tool lowers the cost of work that would have been done anyway, from the extensive margin, where it changes which work can be attempted, and shows that comparisons restricted to shared tasks cannot identify the second without further assumptions. It distinguishes attempting a project from deploying it admissibly, and shows that fixed-task evaluations also cannot see how cheap implementation changes which projects an agent recognizes as candidates. It argues further that complexity counts need a functional denominator and that delegated work is legitimate only while it remains recoverable. The controlled studies of AI-assisted programming examined here measure performance on selected tasks and do not directly identify the value of newly feasible or newly discovered projects. The essay makes no claim about the size of the extensive effect; it claims only that an evaluation which ignores it is incomplete.

1 The Anecdote and What It Asserts

The account in question was posted to a programming forum in October 2026 by an engineer with roughly twenty years of experience [1]. Its narrator reports that his employer built a customer-facing chatbot with AI-generated code, that the first version was imperfect but salvageable, and that he was assigned to rewrite it before being replaced by a newer colleague who preferred autonomous agentic loops. Management, in the narrator's telling, rejected his warning that such loops can produce demonstrations and prototypes but not dependable client-facing software. Eight months later the first version had become unrecoverable, a second version built in the same manner missed its deadlines, early customers abandoned it, and the

colleague resigned. The narrator was then told that he had been correct throughout and was offered a promotion, together with responsibility for the second version.

The system he inherited, as he describes it, consists of two virtual machines and seven containers, an inter-container protocol whose purpose is to repair the containers when they fall out of step, a build and deployment pipeline that cannot be bypassed without damaging the system, some four hundred generated documentation files, and about twenty-nine hundred unit tests that do not appear to detect the defects customers actually encounter. Its administrative interface uses vocabulary that users dislike. Most strikingly, it is not meant to be changed by people: it records its own errors, opens tickets, writes and deploys patches, and closes the tickets, and a human edit risks disrupting that cycle. The narrator compares the design to a perpetual motion machine and concludes that an industry afraid of the difficulty of code has replaced boring ordinary software with an apparatus a thousand times more complicated than the thing it avoided.

This essay takes the account seriously as a report of one person's observations and examines what can and cannot be inferred from it. The distinction matters because the account has become a template for two opposed readings. One reads it as confirmation that agentic software development is a fad that experienced engineers will have to clean up. The other reads it as the complaint of a displaced specialist whose frustration is mistaken for evidence. Neither reading is supported by what the narrator actually establishes, and the more interesting questions lie in what the story leaves out.

2 Observation, Explanation, and the Missing Counterfactual

A careful reading separates three kinds of statement. The first kind is observation: the number of environments, the existence of the repair loop, the customer departures, the narrator's difficulty in understanding what the tests do. These are reports, unverified and anonymous, but they are the sort of claim that could in principle be checked. The second kind is causal explanation: that the failure arose because AI wrote the code, or because management accepted autonomous loops, or because the architecture was elaborate when it need not have been. The third kind is comparative judgment: that a conventional rewrite would have been simpler, cheaper, and better. The narrator is well placed to report the first kind, less well placed to adjudicate the second, and not placed at all to establish the third, because he describes only one of the two histories being compared.

The structure of the narrative adds a selection problem. The narrator was removed from the project, predicted its failure, and then watched the failure occur. That sequence does not make his account false, but it gives him a reason to find in the outcome a confirmation of his earlier position, and it leaves the reader with no testimony from the replacement engineer, the customers, or the managers who made the decisions. The narrator himself notes that the first version was salvageable. If that is accurate, then the decisive failures came later and were failures of what was permitted to accumulate, a point that bears on which factor, the use of generated code or the surrounding governance, deserves the blame.

The most serious omission is the denominator. The post offers an inventory of complexity

and no measure of what the complexity was for. A product called a chatbot may be a thin interface to a language model, or it may be a multi-tenant platform with authentication, isolation between customers, retrieval, tool execution, auditing, billing, and integrations with many external systems. Nine deployment units may be absurd in the first case and conservative in the second. Four hundred documents and thousands of tests are neither virtues nor vices until one knows what they describe and what they check. The narrator's phrase for the product, a stupid chatbot, collapses precisely the distinction needed to evaluate the engineering.

It also helps to disentangle three comparisons that the anecdote runs together. The first concerns the quality of the resulting software. The second concerns the resources consumed to produce it. The third concerns the cost of maintaining it over its useful life. A system may do badly on one and well on another, so that an application that is cheap to produce and expensive to maintain can be advantageous over its lifetime or can be ruinous, depending on magnitudes the story does not give. The strongest conclusion the account supports is that its narrator met a system he considered unreliable and difficult to maintain. The broader conclusion, that agentic development is inherently inferior to conventional programming, requires comparative evidence that a single narrative cannot supply.

3 Two Margins of Technological Change

Ordinary productivity comparisons ask how much faster or cheaper a known task can be done with a new tool. This framing presupposes that the task would have been attempted without the tool. A distinction long familiar in economics separates this intensive margin, where technology lowers the cost of an existing activity, from an extensive margin, where it makes additional activities worth attempting [2, 3]. The distinction is old, but it is routinely neglected when new tools are evaluated on benchmark tasks drawn from what practitioners already do.

Let $\mathcal{Q}$ be a set of conceivable software projects, taken for the moment as fixed. For a method m of producing software and a resource budget B, distinguish two costs. Let $C_{\text{exp}}(p, m)$ be the cost of initiating an experiment on project p by method m, and let $C_{\text{prod}}(p, m)$ be the cost of producing a candidate implementation fit to be evaluated for release, with either cost infinite when the agent cannot incur it at all. An unsuccessful experiment is still an attempt, and it may yield information without yielding an artifact, so the two costs need not move together: a method may make experimentation cheap without making reliable production cheap. Let $A(p, m)$ indicate that an implementation produced by m satisfies the admissibility conditions required for deployment, such as security, recoverability, and preservation of the obligations the system owes its users. Two sets follow:

$$\mathcal{F}_{\text{att}}(m, B) = \{\, p \in \mathcal{Q} : C_{\text{exp}}(p, m) \le B \,\},$$
$$\mathcal{F}_{\text{dep}}(m, B) = \{\, p \in \mathcal{Q} : C_{\text{exp}}(p, m) + C_{\text{prod}}(p, m) \le B \ \wedge \ A(p, m) \,\}.$$

The first is the set of projects that can be attempted and the second is the set that can be both attempted and produced from the same budget and also released. Because costs are

nonnegative, $\mathcal{F}_{\text{dep}}(m, B) \subseteq \mathcal{F}_{\text{att}}(m, B)$. Keeping them apart matters because a method may enlarge the first far more than the second. The ability to generate a prototype is not the ability to deliver a maintained application, and an account that slides silently from one to the other will overstate or understate the effect of a technology depending on the direction of the slide.

A change of method from m_2 to m_1 may leave some projects deployable under both, make some deployable only under m_1, and make others deployable only under m_2. Write $\mathcal{F}_i = \mathcal{F}_{\text{dep}}(m_i, B)$.

Proposition 1 (Non-identification)**.** *A comparison of m_1 and m_2 whose inputs consist exclusively of task-level outcomes observed on $\mathcal{F}_1 \cap \mathcal{F}_2$ cannot identify outcomes on the symmetric difference $\mathcal{F}_1 \triangle \mathcal{F}_2$ without additional assumptions.*

Proof. Consider two worlds that agree on every outcome for tasks in $\mathcal{F}_1 \cap \mathcal{F}_2$ and differ arbitrarily in the value and lifecycle cost of projects in $\mathcal{F}_1 \triangle \mathcal{F}_2$. Any estimator whose inputs are outcomes on the intersection receives identical inputs in the two worlds and so returns the same value in both, although the total effect of changing methods differs between them. The two worlds are taken to be indistinguishable in the membership of the intersection as well as in its outcomes, and an estimator that conditions on further information about how tasks select into the intersection falls under the additional assumptions that the proposition excludes. □

The proposition is elementary, but its methodological consequence is not. A benchmark built from tasks that practitioners already perform measures the shared domain and is silent about work that was out of reach. Identification can be restored by a structural model or by assumptions about how tasks select into the shared domain, but then the assumptions carry the weight that the data cannot. The symmetric difference also has two sides. Some projects may become reachable only through generation, and others, particularly those requiring guarantees that generated artifacts cannot yet provide, may remain reachable only by conventional means. Neither side can be assumed away.

To connect the sets to value, let $V(p)$ be the value of project p, taken for simplicity to be independent of method, and let $K(p, m)$ be its full lifecycle cost under m, including the costs of experimentation and production, maintenance, failure, and the cost of satisfying its obligations. Define the opportunity measure

$$\Omega(m, B) = \sum_{p \in \mathcal{F}_{\text{dep}}(m,B)} \big(V(p) - K(p, m)\big).$$

This is an aggregate measure of the opportunity that a method opens, and it is not realizable welfare, because projects that are individually affordable need not be jointly affordable. If ten projects each cost half of the budget, all ten belong to the set although only two can be undertaken. The realizable quantity is a portfolio value,

$$W^*(m, B) = \max_{\substack{S \subseteq \mathcal{F}_{\text{dep}}(m,B) \\ \sum_{p \in S} (C_{\text{exp}}(p,m) + C_{\text{prod}}(p,m)) \le B}} \sum_{p \in S} \big(V(p) - K(p, m)\big),$$

and the decomposition below does not carry over to it, since the optimal portfolios under two methods need not be related in any simple way. The decomposition is therefore stated for the opportunity measure only. The budget constraint represents the resources available in the period and subtracts nothing further from welfare, since the outlay enters welfare once, through K, which includes $C_{\text{exp}} + C_{\text{prod}}$. The same outlay therefore appears twice in the model, once as a limit on what can be undertaken and once as a cost of what is undertaken, and these are different roles and not a double count. Both quantities also assume additive values and costs, which is an accounting simplification: projects can share infrastructure, knowledge, and reusable components, so that a generated library may lower the cost of later projects and one experiment may supply the knowledge needed to conceive another, and the model does not capture such complementarities, substitution, or path dependence. Writing $\mathcal{F}_i = \mathcal{F}_{\text{dep}}(m_i, B)$,

$$\begin{aligned}\Omega(m_1, B) - \Omega(m_2, B) = &\sum_{p \in \mathcal{F}_1 \cap \mathcal{F}_2} \big(K(p, m_2) - K(p, m_1)\big) \\ &+ \sum_{p \in \mathcal{F}_1 \setminus \mathcal{F}_2} \big(V(p) - K(p, m_1)\big) \\ &- \sum_{p \in \mathcal{F}_2 \setminus \mathcal{F}_1} \big(V(p) - K(p, m_2)\big).\end{aligned}$$

The decomposition is an identity, since the terms in V cancel on the intersection. Its first sum is the intensive margin, a comparison of lifecycle cost on shared work. Its second and third sums are the extensive margin, and each summand may be negative, which is to say that expanded feasibility does not guarantee expanded value.

The decomposition also locates the narrator's complaint, with a caveat. His claim is comparative: a conventional implementation of the same project would have had lower lifecycle cost, so that the project lies in the intersection and the relevant quantity is $K(p, m_2) - K(p, m_1)$, which he takes to be negative. Whether the project belongs to the intersection is not established by the fact that a rewrite was assigned to him, since management may authorize a project without understanding its cost and an engineer may believe he can complete one without being right. What can be said is that his proposal presupposes membership, and that the firm had a salvageable first version and an engineer willing to attempt a rewrite, which makes the case less like that of an agent for whom the conventional alternative was unavailable. The argument for expanded feasibility therefore offers his employer a weaker defense than it would offer an agent facing nonproduction. The essay's central caution applies to the narrator as well, since it warns against treating an imagined counterfactual as an observed one.

The extensive margin matters most for a different population, those for whom the alternative to generated software is no software at all. For such an agent a build that takes two hours of generation and six hours of correction is not an inefficient version of a four-hour task, since the four-hour task was never in the feasible set. The relevant comparison is with the months of unfamiliar study that would otherwise have been required, or with abandonment.

4 Discovery and the Recognized Domain

The model so far treats the set of conceivable projects as fixed and asks which of its members each method can reach. That is too static, because inexpensive implementation can change which projects an agent recognizes as candidates in the first place. Let $\mathcal{Q}^*$ be the underlying universe of projects and $\mathcal{Q}_t \subseteq \mathcal{Q}^*$ the projects that an agent recognizes at time t as candidates for attention. Feasibility is then defined relative to what is recognized, so that $\mathcal{F}_t(m, B) \subseteq \mathcal{Q}_t$, and both the sets and the costs of their members depend on the history of the agent's work. A simulation that produces an unexpected result may prompt an experiment that had not been conceived beforehand. A generated interface may reveal an interaction that suggests a different application. A failed prototype may expose a research question that was not previously visible. In each case the technology has changed more than the cost of a known project, since it has altered which projects are available for consideration, so that $\mathcal{Q}_{t+1}$ differs from what it would have been under another method.

Proposition 2 (Non-identification of discovery)**.** *Let $\mathcal{Q}_0$ be an initial recognized domain, and suppose two methods m_1 and m_2 produce identical outcomes on every project in $\mathcal{Q}_0$ but induce different subsequent domains, $\mathcal{Q}_1(m_1) \neq \mathcal{Q}_1(m_2)$. An evaluation whose inputs consist exclusively of outcomes on the projects in $\mathcal{Q}_0$ cannot distinguish the two methods with respect to the projects in $\mathcal{Q}_1(m_1) \triangle \mathcal{Q}_1(m_2)$.*

Proof. The evaluation receives the same inputs under both methods and therefore returns the same value, while the values of projects that enter the recognized domain under one method and not the other may differ arbitrarily. □

The first proposition shows that shared-task comparisons omit asymmetric feasibility, and the second shows that fixed-task comparisons omit endogenous discovery. The two limits differ in kind. The first concerns projects that lie within the domain but are reachable by only one method, and the second concerns projects that lie outside the domain at the outset and enter it through the use of the method itself, so that even perfect measurement of every initially recognized project would miss them. Each is an observation about what a design can see, and neither says anything about how large the omitted effect is.

This gives the title a second sense. Some alternatives go unattempted because they are too expensive to try, and others because they have not yet become recognizable as alternatives. The second kind cannot be priced in advance, since pricing presupposes that the project is on the list. It is the kind with which the earlier work on the establishment of domains is concerned: the domain over which an agent optimizes is acquired through activity, and a change in the cost of activity changes what is acquired [27]. Two cautions apply. Discovery through cheap experiment is not peculiar to generated software, since any technology that lowers the cost of prototyping can have the effect, and the value of what is discovered may be small or negative, because a larger recognized domain contains more distractions along with more opportunities. The point is methodological. A design that fixes a list of tasks in advance cannot observe discovery at all, because the tasks that discovery would have produced are not on the list.

The discussion suggests three stages that a technology can affect separately: recognition of a project as a candidate, the attempt, and admissible deployment. A technology may enlarge the recognized domain without making its members affordable, make experiments cheap without making their results deployable, or improve the reliability of deployment without changing what is discovered. The stages also form a feedback process. Attempts produce evidence that revises the recognized domain, and deployment produces operational evidence that revises what counts as admissible. In the simplest form, $\mathcal{Q}_{t+1} = D(\mathcal{Q}_t, E_t, m)$, where E_t is the evidence produced by experiments and deployments in period t and D is a domain-revision process. The process need not be monotone. Projects can become recognizable and later be discarded, and implementations that were admissible can become inadmissible as requirements change, which is the situation the work on admissible degradation addresses [31].

5 Complexity Without a Denominator

Counts of containers, tests, documents, and generated lines are not measures of engineering complexity in any sense useful for evaluation. A more informative analysis separates the complexity of the implementation, the complexity of operating the system, the cognitive complexity faced by those who must understand it, and the complexity of the requirements it satisfies. Simon's treatment of complex systems makes the relevant point: apparent intricacy may reflect the structure of the problem, in which case it is irreducible, or the organization of a particular solution, in which case it is not [26]. Brooks drew the same line between essential difficulties inherent in what software must do and accidental difficulties introduced by the means of building it [6]. An inventory cannot tell which kind a given component represents.

Decomposition into several deployable units is likewise not self-evidently a fault. Parnas argued that the proper criterion for dividing a system is whether the division hides design decisions likely to change and yields interfaces that can be understood independently [9]. Under that criterion, nine environments might be a sound arrangement or a poor one, depending on whether each boundary conceals a real decision and whether the interfaces are intelligible. The narrator's complaint is, in effect, that the boundaries do not hold: the containers drift out of agreement and require a machine to reconcile them. That is evidence that the decomposition is not doing the work decomposition is supposed to do, which is a more specific and more credible charge than the claim that there are too many parts.

The converse error is equally available. Elaborate infrastructure can become genuine waste when its components create obligations without corresponding benefits, and the economics of software lifecycles have long held that maintenance, not initial construction, dominates total cost [7, 8]. A cheap first implementation that multiplies the surfaces requiring continuing care can therefore be a net loss even when it is a large saving at the moment of creation. The honest position is that complexity must be assessed against a functional denominator, the work the system performs and the obligations it carries, and that neither a large nor a small inventory settles the matter.

6 Scarcity as an Implicit Regulator

Early programming under severe limits of memory and processing speed supplies a historical contrast with the present. A program that had to fit in a few kilobytes could not afford a component that did not repay the space it occupied, so that every routine, table, and feature was weighed against its footprint at the moment it was added. The constraint priced complexity at the point of decision, where the builder felt it, and it produced a characteristic craft of economy in which small resources were made to do a great deal. The question is whether anything comparable operates when the scarce resource is a budget of tokens, and what follows for the argument of this essay if it does not.

A simple observation shows what scarcity was doing. Let $b(c)$ be the benefit of a component c, $\kappa(c)$ its lifecycle cost, and $c_{\text{perc}}(c)$ the cost the builder perceives at the point of decision, which may include some knowledge of later burdens but ordinarily understates them. A builder who adds a component whenever $b(c) > c_{\text{perc}}(c)$ accepts every component for which $c_{\text{perc}}(c) < b(c) \leq \kappa(c)$, that is, every component whose benefit exceeds its perceived cost but not its lifecycle cost. Holding b and κ fixed, this set weakly grows as perceived cost falls. Under tight memory limits, at least one important dimension of complexity carried an immediate and unavoidable cost, since footprint is paid on every run and on every change. That did not make lifecycle costs transparent, because maintenance burden, correctness risk, and interaction costs could remain largely unknown, but it prevented that dimension from expanding without consequence, and it raised c_{perc} toward κ for the components that mattered most. When the scarce resource is instead the effort of production, and generation makes that effort small, the production part of the perceived cost falls while the lifecycle cost, which is dominated by maintenance [7, 8], does not. Lehman's observation that complexity increases unless work is done to reduce it then acquires a sharper form: the work of adding has become cheap, and the work of reducing must be done deliberately. This is one reason that generated projects can grow out of proportion unless they are supervised at each stage and refactored continuously, since the supervision performs the pricing that scarcity once performed automatically. Wirth made a related complaint about an earlier era, arguing that growth in available memory and speed had been absorbed by growth in software, much of it unnecessary [22], and the economics of resource use suggests why cheaper use of a resource can raise total use [23]. The parallel to Jevons is offered as an analogy, since the artifacts and obligations that cheap generation multiplies are not a single commodity.

The analogy with memory limits is imperfect in a way that matters. A memory limit bounds the artifact, so that bloat is impossible beyond a fixed size. A token budget bounds the process that produces or maintains the artifact, and the artifact itself can grow until the later work of reading, repairing, and extending it has to be paid for. Bloat therefore acquires a price eventually, as when a system grows large enough that keeping it running consumes a substantial amount of paid model usage, which the narrator of the anecdote identifies as a symptom of its design. The price arrives late and falls on whoever inherits the system, and the delay is the problem. Scarcity regulates only when the constrained resource is one that the design consumes and when the builder bears the downstream cost of what is added.

Two further mechanisms connect resources to resourcefulness. The first is the substitution of purchase for acquisition. An agent with ample funds can buy delegation where an agent without them must learn, so that a task which would have forced the acquisition of a capability is instead handed to a model. The delegation may be entirely sensible, and it is the source of the extensive margin discussed above, but it forgoes the theory that the builder would have formed in doing the work [19], and with it some of the ability to recover control when the delegate fails. A constraint on funds therefore acts, among its other effects, as a pressure toward acquisition, and relaxing it removes that pressure. The second mechanism is that a constraint need not be real in order to regulate. Work on mental accounting finds that people divide fungible money into separate categories and treat spending from each as limited by its own allowance, so that separate accounts behave as budgets even though the underlying resource is a single pool [21]. If the same holds for tokens or effort, a deliberately ring-fenced allowance for a project, a component, or a repair loop could reproduce part of what scarcity did, by raising the perceived cost of addition toward its lifecycle cost at the point of decision. This is a conjecture and not a result, and it is consistent with the conditional claim stated below: the regulation works through where a cost is perceived and who bears it, and it does not require that the resource be physically exhausted.

The claim must also be stated with restraint. Scarcity is not generally beneficial. Research on the psychology of scarcity finds that having too little can capture attention and reduce the capacity available for other tasks [24], the literature on creativity under constraint finds that limits foster inventiveness under some conditions and not others [25], and the remembered cleverness of early programmers is a survivorship sample of those whose constrained projects worked. For an agent without funds, constraint also excludes projects, which is a contraction of the extensive margin and not a discipline. The defensible claim is conditional: where the scarce resource coincides with what the design consumes, and the builder bears what the design later costs, constraint can discipline design and favor smaller, better understood, more recoverable systems. Whether agents working with small token budgets in fact produce such systems is an empirical question that this essay does not settle. The implication for the argument is nonetheless direct. Where cheap production removes the scarcity that once limited accumulation, the regulation has to become explicit, in the form of ring-fenced budgets for complexity, which may bind even though they are nominal, deletion treated as a required operation, independent review, and verification tied to requirements. These are the conditions of admissibility and recoverability discussed above, and they are what the system in the anecdote appears to have lacked.

7 Verification and the Loop That Grades Itself

The post's most instructive details concern verification. Twenty-nine hundred tests exist, and customers nevertheless encounter defects that the tests do not report. The point generalizes beyond this system. The existence of a verification procedure does not establish that the procedure corresponds to the requirements that matter, and Dijkstra's old warning that testing can show the presence of faults but never their absence applies with special force when the

tests are numerous and uniformly passing [18]. A suite generated alongside the code it checks is liable to encode the same assumptions as the code, so that agreement between them is evidence of consistency and says little about correctness against what users need. Leveson's analysis of accidents in systems whose components each behaved as specified shows the same gap between component-level conformance and system-level safety [20].

The self-repair cycle sharpens the issue. A loop that detects an error, creates a ticket, writes a patch, deploys it, and closes the ticket executes a sequence of operations, and the closing of a ticket is a record that the sequence completed. It is not a demonstration that the defect was removed or that the system now behaves as intended. Whether it is such a demonstration depends on the independence of the check from the repair. If the verifier that confirms a fix shares the blind spots of the process that produced the fix, then a closed ticket carries little information about the underlying property. The earlier discussion of verification without correspondence [28] states the general form: a system can satisfy every procedure it applies to itself while those procedures fail to establish the properties its users depend on, and the remedy is to keep verification tied to independently specified obligations and to outcomes observable from outside the system.

A formal treatment of a neighboring problem sharpens what a repair loop can honestly claim. When a verified artifact is transformed, an earlier verification result does not transfer automatically, because correspondence between the original claim and its successor, taken as mutual entailment, is undecidable once the claim language can express first-order validity over a relation of arity two or more [29]. The same work shows that correspondence becomes decidable inside a restricted class of transformations generated by a terminating and confluent rewrite system, and it draws the practical conclusion that an honest procedure has three outputs: a positive certificate, a negative result relative to its own rule set, and an admission that the transformation lies outside the certified fragment. Its governing maxim is that failure of certification is not certification of failure, and its notation separates a relation that is computed from a judgment that is merely attested, since a signed record of a verdict is evidence of a claim and not a derivation of it. The setting there is logical claims and not program patches, so the transfer to software is an analogy. The analogy is nonetheless close. A generated patch is a transformation of the program, the passing of an earlier test suite is a verification result about the program as it was, and the closing of a ticket is an attested judgment. Unless the patch falls inside a fragment where preservation can be certified, a closed ticket reports that a procedure was followed and leaves the inheritance of earlier verification unestablished, and the loop has no output corresponding to the third value, the admission that a change lies outside what it can certify. A repair also carries three obligations that automated ticket closure tends to merge: preservation of the invariants previously specified, satisfaction of the repair obligation that the reported defect created, and assessment of possible regressions outside the previously specified property set. The third is not a verification obligation in the same sense as the others, since no finite procedure can establish the absence of violations of properties that were never stated. It calls for exploratory testing, monitoring, additional specification, or explicit acceptance of residual uncertainty, and a loop that closes tickets without a place for that acceptance has no output for it. A patch can correct the reported defect while violating

an unrelated invariant, and it can preserve every previously checked property while leaving the defect in place. A closed ticket speaks, if it speaks to anything, to the second obligation, and the inheritance of earlier verification is at stake in the first and third, which lie at and beyond the boundary of what the loop can certify.

This does not show that autonomous maintenance is incoherent, as the narrator's analogy with perpetual motion suggests. Automated monitoring, recovery, and deployment are established practice, and a self-repairing component is a recognizable extension of them. The relevant questions are engineering questions about whether the system has rollback, whether it escalates to a person when its confidence is low, and whether anything outside the loop checks the loop's results. The fault, if the account is accurate, lies in a design whose evidence of repair is produced and judged by the same machinery, and that fault would be as serious in a system of human authors who reviewed only their own work. The difficulty is not peculiar to generated software. Human teams also build elaborate procedures that certify conformity with their own representations while overlooking external requirements, and the question in every case is whether verification stays connected to something independent.

8 Subordination, Recoverability, and Delegated Work

The report that human edits endanger the system inverts the usual relation between a tool and the thing it serves. Normally automation is subordinate to an application that people can read and change. Here the application is said to be subordinate to the automation, so that the maintainers must refrain from touching the system lest they break the process that maintains it. Bainbridge's analysis of the ironies of automation anticipates the structure: as automation takes over routine activity, the human operator is left with the rare and difficult cases, with less practice and less understanding of the system than the routine work would have supplied [10]. A system designed so that human intervention is hazardous makes the irony absolute, since the people held responsible for it are the people it is least safe for.

Frameworks for human interaction with automation provide a vocabulary for what has gone wrong. Parasuraman, Sheridan, and Wickens distinguish what is automated, at what level of authority, and where human intervention remains available [11], and the guidelines for human-AI interaction propose that systems communicate their limits and permit correction when their behavior is inappropriate [13]. Naur's argument that programming is the construction of a theory held by the programmers, rather than the production of text, explains why the cost of separating generation from understanding can be high: when the code is produced by a process whose reasoning no person holds, a person asked to modify it has no theory with which to do so [19]. These considerations bear a substantive warning against an uncritical reading of delegated implementation.

The distinction between the explicit and the delegated aspects of work gives the point a more general form. Programming labor is not simply removed by generation; it is redistributed. Syntax, boilerplate, dependency management, and routine implementation may be relegated to automated processes, while requirements, experimental design, integration, interpretation of errors, and judgments about acceptable outcomes concentrate in human attention. The

legitimacy of such relegation depends on the conditions under which control can be recovered [30]. A delegated aspect is acceptable to the extent that it can be inspected, corrected, and reclaimed when it fails, and unacceptable when reclaiming it would require dismantling the arrangement that depends on it. By this criterion the system described in the post fails in a precise way, and the failure is independent of whether the code was written by a person or a model. A hand-written system whose maintenance depended on never being edited by hand would fail the same criterion. Recovery of control is the test, and the author of the code is incidental.

The same distinction recurs in institutional settings. An organization can keep its officers, committees, and legal identity while the capabilities that determine its practical effectiveness are supplied from outside, so that legal separation and functional dependence are different properties, and a formal rule against coordination between two bodies is compatible with one of them being unable to function without the other. The test of recoverability asks whether the delegating party could resume the delegated function, and not whether it retains the title.

9 Feasibility Is Not Admissibility

An expansion of what can be produced does not expand what is acceptable to deploy. A system that is inexpensive to build may still violate security requirements, destroy information that cannot be reconstructed, resist correction, or expose users to risks they have not accepted. It is therefore necessary to keep two questions apart: whether a project has become feasible to attempt, and whether a particular implementation is admissible to release. The sets $\mathcal{F}_{\text{att}}$ and $\mathcal{F}_{\text{dep}}$ defined above keep the two questions apart, but the distinction is easily lost in discussion, where cheapness is taken for adequacy and inadequacy for infeasibility.

The work on admissible degradation treats the admissibility of a system as a property of what it preserves under failure, namely which obligations it keeps, which capabilities it can shed without harm, and which compensating continuations remain available [31]. Applied to the present case, the framework changes the question asked of generated software. The existence of defects in an application does not distinguish its admissibility from that of a hand-written application, since defects are common to both. What distinguishes a recoverable from an unrecoverable system is whether its defects can be found and repaired without sacrificing its essential obligations. An application with many shallow defects and a practical route to repair may be preferable to one with few defects and no such route. Equally, repeated repair stops being a sign of healthy iteration when it consumes the resources needed to honor the system's commitments to users.

This distinction permits an acknowledgment that the narrator's company may have deployed an inadmissible system without generalizing the finding to the method. The failure he describes, a build process that cannot be bypassed, a repair loop with no independent check, a codebase that people cannot safely alter, is a failure of admissibility. It is a failure that any development method could produce, and one that disciplined use of generated code could avoid.

10 Visible Failure and Invisible Production

Generated software may contain thousands of successful implementation decisions and a few conspicuous failures. The successes recede into the background of a working system, while the failures interrupt work and attract attention. The result is an observational asymmetry. The person doing the correction experiences its cost directly, whereas the labor that generation avoided is counterfactual and has no occasion to be noticed. Subjective frustration is accordingly an unreliable measure of aggregate productivity, in either direction, and the same asymmetry can cause an enthusiast to overlook the cost of the defects that eventually surface.

Controlled studies are the corrective, and the available evidence is mixed in a way that illustrates the problem of margins. A controlled experiment in which developers implemented a defined task in JavaScript found that those using an AI pair-programming assistant completed the task 55.8 percent faster than the control group [14]. A later randomized trial with experienced open-source developers working on tasks in repositories they knew well found that access to AI tools increased completion time by roughly nineteen percent [15]. A subsequent report from the same group described the difficulty of continuing such measurements, noting that developers had become reluctant to work without AI and that tasks done by parallel agents resisted straightforward timing, so that selection effects were distorting the sample [16]. These findings are not contradictory. They measure different tasks, different populations, and different tools at different times, and none of them was designed to identify the value of projects newly attempted because of AI assistance, since participants worked on assigned or selected tasks.

The designs do not directly identify the value of projects that would not otherwise have existed, and the third finding suggests why the problem is hard: as the technology changes what people are willing to attempt, the comparison set changes with it. The present argument does not assume that the extensive margin is large, and it would be a mistake to suppose that its size can be inferred from the plausibility of its existence. It is an open empirical question whose answer would require designs that follow what people do when the cost of attempting work changes, and not merely how quickly they do what they were already doing.

A METR analysis from May 2026 approaches the same problem from the side of measurement [17]. It distinguishes uplift on the tasks a person did before AI, uplift on the mix of tasks chosen once AI is available, which may include tasks not previously attempted, and uplift in value, and it argues under simplifying assumptions that the first is no greater than the third, which is in turn no greater than the second. It also names a class of tasks done mainly because AI made them cheap, for which uplift on the new mix can look very large while the gain in value is small. The result supports two cautions made here. A rise in the number of projects attempted is not a rise in value, which is why the summands of the extensive margin above may be negative, and an evaluation confined to old tasks understates how the mix of work has changed. The post treats the change as a reallocation of effort among tasks, and it lists exploration and learning among the complications that could reverse its inequalities. It does not appear to make the formation of the recognized domain its object, which is the process this essay treats separately.

11 Competence and the Selection of Work

The traditional picture of programming competence identifies it with the ability to produce implementations directly. AI assistance separates the ability to specify, evaluate, and direct an implementation from the ability to write every component by hand. The separation does not make implementation knowledge irrelevant, and Naur's argument suggests that it makes understanding more valuable and not less. It changes how competence is distributed across the human and machine parts of a working system. A person unable to write a sophisticated simulation from scratch may still be able to design its experiments, recognize conceptual errors in its output, validate its results, and coordinate improvement over successive iterations. Competence is then partly a property of the trajectory of work that an agent together with its tools can sustain, and it is poorly measured by a test of unaided production.

The economics of general-purpose technologies offer a longer perspective on how such shifts are recognized. David observed that the benefits of electric power appeared in measured productivity only after organizations had restructured around it, and that comparisons made during the transition understated what was changing [4]. Bresnahan and Trajtenberg described general-purpose technologies as those whose significance lies in the complementary innovations they induce across many sectors [5]. Weitzman's account of recombinant growth, in which the limiting factor on innovation is the capacity to test combinations of existing ideas, suggests why cheaper implementation could matter beyond the savings on any given task [3], and Romer's treatment of growth as driven by an expanding stock of nonrival ideas and designs supports the same distinction between producing known goods more cheaply and enlarging what can be produced [2]. Autor's analysis of automation cautions against equating the automation of tasks with the elimination of occupations or expertise [12], which fits the redistribution of programming responsibilities described above.

The same literature implies a new selection problem. When the cost of attempting a project falls, projects once rejected as impractical become reasonable experiments, and speculative ideas can be built, tested, and discarded cheaply. Greater feasibility does not ensure greater value. An enlarged domain contains more useful experiments, but also more distractions, more defective systems, and more occasions for effort to be spent on work that should not have been begun. The central question is therefore how agents choose, evaluate, and govern work once the cost of attempting it has changed, and it is a question that neither enthusiasm nor nostalgia answers.

12 What the Account Leaves Standing

Returning to the anecdote, three conclusions survive scrutiny and one does not. The conclusion that does not survive is the sweeping claim that agentic development cannot yield production-quality software, which requires a comparison the account does not offer and cannot offer from a single case. The first of those that remain is that the system described, if accurately reported, failed on criteria of governance: it allowed automation to outgrow the application with no scarcity or deliberate budget to price that growth, it let verification become self-referential,

and it made the system dangerous for the people responsible for it. The second is that the narrator's claim is a comparative claim about a supposedly shared task, whose membership in the shared domain is presupposed and not established, and which, if that membership is granted, concerns lifecycle cost on work a conventional method could do and is not answered by appeal to expanded feasibility. The third is that the narrator's question, whether generated software is worse than conventional software, is a legitimate one but covers only a portion of the transformation under way.

A complete evaluation must also ask which projects would not have existed without the technology, which experiments would not have been attempted, which alternatives would not have become recognizable as candidates, and which capabilities would have remained out of reach, and it must ask these in a way that does not presume the answers are large. The most significant consequence of automated implementation may lie less in reducing the labor required to produce familiar artifacts than in moving the boundary between what can be conceived and what can be attempted. An engineering evaluation that counts only the cost of correcting generated defects, without accounting for the work that generation made possible, takes a visible expense for a complete account of production, and an enthusiast's evaluation that counts only the work made possible, without the lifecycle obligations it creates, makes the opposite error. The comparison worth making fixes the requirements, measures the shared domain and the asymmetric difference separately, and holds every implementation, however produced, to the same standard of admissibility.

References

[1] u/BurnedOutCodeMonkey. "Update on VibeCoding Nightmare Job." Post on r/programmer, Reddit, October 2026. `reddit.com/r/programmer/comments/1x6b09v/`

[2] Romer, P. M. (1990). Endogenous technological change. *Journal of Political Economy*, 98(5), S71–S102.

[3] Weitzman, M. L. (1998). Recombinant growth. *Quarterly Journal of Economics*, 113(2), 331–360.

[4] David, P. A. (1990). The dynamo and the computer: An historical perspective on the modern productivity paradox. *American Economic Review*, 80(2), 355–361.

[5] Bresnahan, T. F., and Trajtenberg, M. (1995). General purpose technologies: "Engines of growth"? *Journal of Econometrics*, 65(1), 83–108.

[6] Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4), 10–19.

[7] Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

[8] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060–1076.

[9] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.

[10] Bainbridge, L. (1983). Ironies of automation. *Automatica*, 19(6), 775–779.

[11] Parasuraman, R., Sheridan, T. B., and Wickens, C. D. (2000). A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 30(3), 286–297.

[12] Autor, D. H. (2015). Why are there still so many jobs? The history and future of workplace automation. *Journal of Economic Perspectives*, 29(3), 3–30.

[13] Amershi, S., et al. (2019). Guidelines for human-AI interaction. In *Proceedings of CHI 2019*, Article 3, 1–13.

[14] Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590.

[15] Becker, J., Rush, N., Barnes, E., and Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. arXiv:2507.09089.

[16] Becker, J., Rush, N., Cunningham, T., Rein, D., and Mahamud, K. (2026). We are changing our developer productivity experiment design. *METR*, 24 February 2026. `metr.org/blog/2026-02-24-uplift-update/`

[17] Cunningham, T., and Whitfill, P. (2026). Task substitution and uplift. *METR*, 8 May 2026. `metr.org/blog/2026-05-08-task-substitution-and-uplift/`

[18] Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10), 859–866.

[19] Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15(5), 253–261.

[20] Leveson, N. G. (2011). *Engineering a Safer World: Systems Thinking Applied to Safety.* MIT Press.

[21] Thaler, R. H. (1999). Mental accounting matters. *Journal of Behavioral Decision Making*, 12(3), 183–206.

[22] Wirth, N. (1995). A plea for lean software. *Computer*, 28(2), 64–68.

[23] Jevons, W. S. (1865). *The Coal Question.* Macmillan.

[24] Mullainathan, S., and Shafir, E. (2013). *Scarcity: Why Having Too Little Means So Much.* Times Books.

[25] Stokes, P. D. (2005). *Creativity from Constraints: The Psychology of Breakthrough.* Springer.

[26] Simon, H. A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6), 467–482.

[27] Flyxion (2026). The domain and its establishment. Unpublished manuscript.

[28] Flyxion (2026). Verification without correspondence. Unpublished manuscript.

[29] Flyxion (2026). Verification inheritance under transformation: Robust commitments, claim correspondence, and epistemic validity in mathematical documents. Manuscript, August 2026. `github.com/8b-is/claimshift`

[30] Flyxion (2026). Aspect relegation theory. Unpublished manuscript.

[31] Flyxion (2026). Admissible degradation. Unpublished manuscript.