The End of Software Engineering

Part I of *The End of Software Engineering and the Rise of Agentic Engineering*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

September 2026

Abstract

Software engineering is being redefined as agentic engineering. The decisive change is a new unit of work: engineers specify objectives and constraints, design compositions, and verify artifacts produced by agents that select, run, and repair actions. We operationalize this redefinition through three features visible in execution records: who controls the next action, how many actions occur per human decision, and where verification sits relative to authorship. These features separate agentic engineering from AI-assisted programming and workflow automation. The empirical record shows the transition at scale. One platform reports more than one million agent-created pull requests in five months, while a large field study identifies 302,600 verified AI-authored commits across 6,299 repositories. Productivity varies by task and population: one randomized trial reports a 55.8% speed increase, another a 19% slowdown. Benchmark leakage and rising defect indicators show that code volume cannot serve as the governing measure. They make specification quality, review cost, security, and evidence design part of the primary engineering artifact. Software engineering therefore changes its center of gravity. Direct authorship becomes one implementation activity inside a broader discipline of specifying, orchestrating, and verifying machine-produced work.

1 Introduction

Software engineering is being redefined as agentic engineering. Magazine columns, preprints, vendor reports, and working engineers describe the change with different language [81, 20]. This paper gives it an operational form and identifies the work that moves.

The research question has three parts, asked together because they cannot be answered apart.

> **RQ1.** What did software engineering mean as a discipline, what changed when programs began to choose and run their own tool invocations, and what is the operational content of the claim that the unit of work has shifted?

The series advances one central claim.

> **Central claim.** Software engineering is being redefined as agentic engineering. Its unit of work is moving from hand-authored code to specifications, contracts, orchestrated agent activity, and verification of the artifacts that agents author, run, and repair. Engineering judgment becomes the control layer: engineers define objectives and constraints, design the composition, choose the evidence, and accept or reject the result.

Section 7 identifies the operational signature of this change and examines its cost, reach, quality risk, and institutional response. Variation in productivity and defect rates determines how the transition must be governed; action-control, adoption, and review data establish the transition itself.

1.1 Contributions

Software engineering is a documented object, with a standardized definition, a body of knowledge, and accredited curricula. A claim about its end can therefore be checked against something specific (Section 2).

The vocabulary of the present moment cannot carry that check. One of its central terms has no traceable coining event, which is a finding rather than an inconvenience (Section 3). The definitions in Section 4 separate three practices by features an observer can measure on an execution record.

The empirical record distinguishes redefinition from productivity. Two randomized trials report different speed effects, while four independent method types report rising quality risks. These results place specification, verification, and review at the center of the new discipline (Sections 5 to 7).

The discipline's established commitments become constraints on delegated production. The primary review artifact moves from lines of code to a specification, a contract, or a body of evidence. The skill that grows in importance is designing and checking compositions rather than authoring every step (Section 6).

1.2 Sources and evidence

The sources fall into six classes: peer-reviewed studies, preprints, vendor telemetry, surveys, practitioner reports, and earnings calls. Those classes are not interchangeable, and public discussion of this subject routinely treats them as though they were, so each quantitative claim is labelled with its class, its denominator, and its window. Where two sources disagree, both appear in the same passage, and where the measurement that would settle a question does not exist, we name it.

The evidence in Part I is historical and empirical. No code base is used as evidence here, so nothing below rests on the authors' own systems.

2 The founding of the discipline

2.1 The NATO conferences

The term software engineering was institutionalized at a conference sponsored by the NATO Science Committee at Garmisch in October 1968, whose report Naur and Randell edited into 231 pages [52]. A second conference at Rome the following October produced a 164-page companion volume on techniques [19]. The two together mark a move from naming a problem to proposing a response to it.

The conference did not describe how work was done. It proposed how work should be organized, and the people making the proposal thought the existing organization was failing. Historians record the choice of the word engineering as deliberate, and as contested from the start [48]. Mahoney dates the field's origin to that October and argues that its self-definition never settled. He quotes a practitioner two decades later: software engineering is not yet a true engineering discipline, but it has the potential to become one [48].

Fifty years later, one of the report's editors wrote that some large bespoke software projects still suffered from problems all too reminiscent of those that gave rise to the software crisis in 1968 [66]. The discipline was proposed as a response to a crisis. The participant best placed to judge found

Year	Document	What it fixed
1990	IEEE Std 610.12 [38]	The definition of the discipline as a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software
2014	SWEBOK 3.0 [76]	Fifteen knowledge areas as the field's self-declared scope
2015	SE2014 curriculum guidelines [69]	The undergraduate competence required across institutions
2017	ISO/IEC/IEEE 24765 [39]	The 1990 wording carried forward as the live vocabulary standard
2024	CS2023 [25]	The first joint curriculum to include the artificial intelligence society
2024	SWEBOK 4.0 [77]	Eighteen knowledge areas, with architecture, operations, and security added

Table 1: Definitional continuity from 1990 to 2024. The definition of the discipline did not change across this period, while its enumerated scope grew.

the crisis still running, for that class of project, five decades on.

2.2 Codification

Whatever its contested self-image, the field acquired a written definition and kept it. IEEE Std 610.12-1990 defined software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [38]. That wording was carried forward into ISO/IEC/IEEE 24765 and is maintained as the live vocabulary standard [39].

The scope was enumerated separately. The Software Engineering Body of Knowledge set out fifteen knowledge areas in its 2014 edition [76]. Its fourth edition set out eighteen, adding software architecture, software engineering operations, and software security as areas in their own right [77]. Curriculum guidelines fixed what an undergraduate degree should contain [69]. The 2023 computing curricula were the first produced jointly by the two professional societies and the artificial intelligence association [25], which dates the curricular arrival of this subject matter to just before the period we examine (Table 1).

The definition held for thirty-four years. A claim that the discipline is ending is therefore a claim about a specific written object, not about a mood.

The scope grew toward operations and security in the edition published a few months before agentic tools became widely available. That is where the burden of agent-authored work has since fallen hardest.

2.3 Foundational ideas

Four ideas from the discipline's literature carry the argument later in this paper. Each concerns something other than who types.

Brooks separated the difficulties of software into the essential, which are inherent in the conceptual structure of the thing being built, and the accidental, which attend its representation. His claim, in the epigraph of the 1986 paper, was that no single development in either technology or management technique promised even one order-of-magnitude improvement within a decade [16]. The claim is falsifiable and has a scope: it is about a decade and about a single development. Any productivity number in Section 5 is properly read against it.

Period	Wave	The claim	Documented outcome
1950s	Automatic programming [8]	Compilers would understand what the user wanted, removing the need to program	Backus records the prevailing 1954 belief that efficient programming could not be automated
Early 1980s	Fourth-generation languages [49]	Non-procedural languages and generators would remove the programmer bottleneck from application development	The category persisted as a tool class, the bottleneck did not move, and the book's title has become the standing example of the genre
Late 1980s to 1990s	CASE tools [33]	Integrated computer-aided software engineering environments would industrialize development at departmental scale	A government audit released in June 1993 found the Department of Defense not ready to implement its integrated CASE programme departmentwide, recommending that purchases be restricted to pilot projects and citing an absent business-process improvement programme and unassessed skill levels
2000s	Model-driven architecture [70]	Models would be the primary artifact and code would be generated from them, round-trip	An empirical study of the techniques found them perceived as ineffective and inefficient in practice

Table 2: Four documented automation waves, each with a primary or peer-reviewed source for its outcome.

Parnas gave the criterion for decomposing a system into modules: hide the decisions likely to change behind interfaces that do not expose them [59]. The criterion is about where information lives, and it is indifferent to the authorship of the code on either side of the interface. Parnas later described the way working systems decay when maintenance investment stops, opening with the observation that programs, like people, get old [60].

Dijkstra's Turing lecture located the difficulty of programming in the limits of human intellectual capacity, not in tooling [27]. A later essay described a program as an abstract symbol manipulator, made concrete by supplying a computer to it [28]. That phrasing raises a direct question for the present moment: whether a system that produces symbol manipulators without reasoning formally about them has changed the bound Dijkstra named.

Brooks's earlier book supplied the estimation results, including the observation that adding people to a late project makes it later [17]. What matters here is the subject of that law rather than the law itself. Fifty years ago the binding constraint on large software efforts was already understood to be coordination cost, not authoring speed.

2.4 Earlier automation waves

The claim that programming is about to be automated is not new. It has been made roughly every fifteen years, by serious people, with artifacts behind it. Four of those waves have a primary or peer-reviewed source documenting the outcome (Table 2).

One artifact makes the genre concrete. The Last One, a 1981 British product, was a menu-driven generator of BASIC programs. Its creator is recorded as having named it for the belief that it was

the last human-produced program that would need to be written [44]. The source is tertiary. We found no primary 1981 advertisement or trade-press review, so the wording available to us is an encyclopedia's account rather than verified original copy. With that caveat, the product shows that the end of direct human authorship has been predicted before with something shipping behind it.

The earlier waves automated translation, generation, and reuse while leaving control of the next action with the programmer. The present wave transfers that control to agents, multiplies the actions performed per human decision, and moves verification after machine-produced artifacts exist. These measurable differences, defined in Section 4, turn another automation wave into a redefinition of the engineering process.

A recent preprint arguing the strongest form of the present thesis offers its own periodization of software delivery: licensed software, then hosted software, then delivered outcomes [20]. It names exemplars for each generation but gives no source for the scheme. The four waves in Table 2 come with documented outcomes; this scheme does not. Reading the present claim as the fifth entry in that table is more useful than reading it as the first of its kind.

3 Vocabulary

The vocabulary of the present moment lacks a canonical origin, and its terms differ sharply in provenance (Figure 1). Which ones have a traceable origin decides which can be adopted as found and which must be stipulated.

3.1 Vibe coding

Vibe coding was coined in a social post on 2 February 2025. It named a kind of coding in which the author gives in to the vibes and forgets that the code even exists [41]. We could not fetch the original post, so we record the wording as corroborated across a dictionary citation and multiple outlets rather than verified at source. The term reached a dictionary's slang entry five weeks later, on 8 March 2025 [51], and a dictionary word of the year on 6 November 2025 [24]. Criticism followed quickly, including the objection that the phrasing wrongly implies engineers proceed on vibes. We could not verify those critiques at primary-source level, and flag them rather than drop them.

Kent Beck drew the distinction the rest of this paper depends on. In vibe coding you do not care about the code, only about how the system behaves. In augmented coding you care about the code, its complexity, the tests, and their coverage [12]. That distinction separates engineering from its absence, and the tooling does not. Nothing here should be read as a defence of the first practice.

3.2 Context engineering

Context engineering emerged over about two weeks in June 2025, without a single coining. A vendor blog post of 23 June 2025 defined it as building dynamic systems that supply the right information and tools in the right format for the model to accomplish the task [22]. That post came two days before the social post most often credited with popularizing the term. We could not fetch that later post, so we record its date as corroborated across secondary accounts rather than verified at source. A vendor engineering post of 29 September 2025 then fixed a definition: the set of strategies for curating and maintaining the optimal set of tokens during inference [6]. An academic survey synthesizing more than 1,400 papers formalized the area the following month [50]. Part V of this series owns the term.

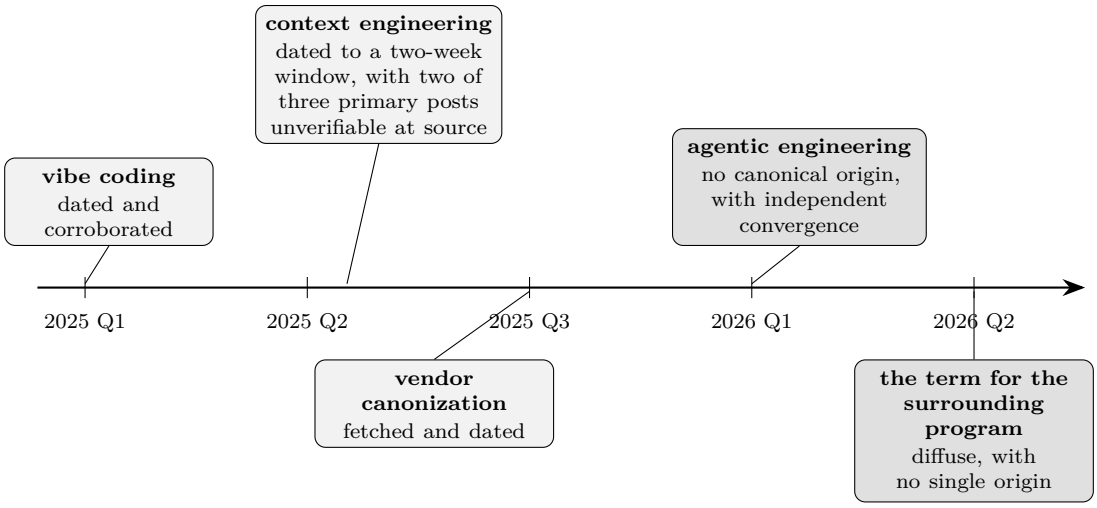

Figure 1: Provenance of four terms, with verification status, where darker shading marks terms for which no coining event could be established. The contrast between the first two and the last two is the reason Section 4 stipulates rather than reports.

3.3 Agentic engineering

Agentic engineering, the term this series uses in its own title, has no canonical coining event. A practitioner blog post of 4 February 2026 writes that Andrej Karpathy suggested the phrase that week, and links to a social post as its authority [57]. We could not retrieve the linked post by any method available to us. The attribution is documented but not verified at source. A week later, the practitioner who maintains the most detailed guide to the practice recorded the term appearing in several places at once, naming a model vendor's positioning alongside two individuals [84]. In March 2026 that author wrote the guide's definitional page, calling the practice developing software with the assistance of coding agents, and attributed the term to nobody [82]. The same guide does credit a named individual for coining vibe coding.

What this shows is concurrent independent use, not a coining event, with one second-hand attribution that cannot be checked. The widely repeated claim that a named person coined the term in February 2026 rests on the unverifiable link. We do not state it as fact.

A fourth term, naming the program that runs the loop around a model, entered use during 2026 with no single origin either. It is observer and practitioner vocabulary describing what vendors build. Part II of this series owns the object it names.

The problem is not confined to new coinages. A position paper argues that the word agent now carries several incompatible readings across subfields, amplified by recent language-model systems, and proposes a multidimensional description in place of a single autonomy axis [14]. Practitioner taxonomies still circulate numbered ladders modelled on the automotive driving-automation levels. The academic work closest to this series argues instead for a distinction between engineering for humans and engineering for agents [37]. No consensus ladder exists.

4 Definitions

The definitions below are stipulated rather than reported, because no source we located defines these terms operationally and one of the central terms has no traceable coining event [82, 14]. Each is written so that an observer could check whether it applies to a given system, and none uses the

term it defines.

4.1 Classical software engineering

Definition 4.1 (Classical software engineering)**.** The practice constituted at the NATO conferences of 1968 and 1969 [52, 19] and codified in IEEE Std 610.12-1990 [38], carried forward unchanged in wording into ISO/IEC/IEEE 24765 [39], as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. Its scope is enumerated by the Software Engineering Body of Knowledge [76, 77] and by the undergraduate curriculum guidelines [69]. The feature that matters for this series is not in the wording but in the assumption underneath it: the accountable unit of work is source code that a person wrote.

4.2 Agents and autonomy

Definition 4.2 (Agent)**.** A program that, given a goal stated in natural language, repeatedly chooses which tool to invoke next from an available set, observes the result, and decides whether to continue or stop, without a human choosing each invocation. A system is not an agent under this definition if code fixed before the run controls every transition, even when run-time values lead that controller through different paths.

We define an agent by run-time choice rather than by capability because the term agent now carries several incompatible readings across subfields [14]. The run-time-choice criterion is narrower than any of them. Settling it requires the trace together with the controller or decision record that selected each next action.

Definition 4.3 (Autonomy level)**.** The number of consecutive tool invocations an agent performs between two human decisions, measured over a defined task and a defined window. The level is reported as a distribution over tasks rather than as a single label. A system in which a human approves every invocation has autonomy level one.

Remark 4.4. No consensus ladder analogous to the SAE driving-automation levels exists for coding agents. Practitioner taxonomies of that shape circulate. The academic position closest to this series argues for a distinction between engineering for humans and engineering for agents instead of a numbered scale [37], and a separate position paper argues for replacing the single autonomy axis with a multidimensional description [14]. Definition 4.3 therefore reports a count, not a rung.

4.3 Unit of work and replacement

Definition 4.5 (Unit of work)**.** The smallest artifact a practitioner is accountable for producing and defending in review. It is identified by asking what a reviewer rejects: if the reviewer rejects lines of code, the unit is code; if the reviewer rejects a specification, a contract, or a body of verification evidence, the unit is that artifact.

The test is behavioural by design. It does not ask what a practitioner believes their job to be, and it does not ask what a tool produces. It asks what a second person sends back. Spec-driven toolkits make the question concrete by proposing that the specification, rather than the diff, becomes the object under review [35].

Definition 4.6 (Replacement, operational sense)**.** For a given task class and a given window, the share of the accountable unit of work that a human authors by hand has fallen, while the share that the human specifies, constrains, reviews, and verifies has risen. Replacement in this sense says

nothing about employment. It is a claim about where a practitioner's hours go, and it is relative to a named task class.

Employment is a separate question with separate evidence, and this series keeps it separate. The strongest study we located finds no economy-wide displacement. It also finds that workers aged 22 to 25 in exposed occupations sit 19% below the employment trend of less-exposed peers. The effect works through reduced hiring rather than increased firing [18].

4.4 The three practices

Definition 4.7 (Agentic engineering)**.** The practice in which a human states goals and constraints, delegates the authoring and running of code to agents in the sense of Definition 4.2, and spends most of their own effort on specification, constraint, and verification of what the agents produced. It is distinguished from adjacent practices by three features, each measurable from an execution record together with its controller: who selects the next action, how many actions occur per human decision, and where verification sits relative to authoring.

Definition 4.8 (AI-assisted programming)**.** The practice in which a human authors code and an AI system proposes completions or edits that the human accepts or rejects one at a time. On the three features of Definition 4.7: the human decides each next code action by accepting or rejecting one proposal at a time, there is approximately one system action per human decision, and verification sits inside the human's authoring loop as an immediate accept-or-reject.

Definition 4.9 (Workflow automation)**.** The execution of a sequence of steps fixed in advance by code or configuration, in which any model calls occupy predetermined slots. On the three features: neither a human nor a model controls the transition rule at run time. The path may depend on run-time values, but the controller that chooses it is fixed before the run.

Definition 4.9 follows a distinction drawn in vendor engineering guidance, which separates systems where models and tools are orchestrated through predefined code paths from systems that direct their own processes and tool usage [5]. The distinction is worth keeping because the two fail differently. They can also cost differently. On one benchmark, a three-phase pipeline with no agent loop resolved 32.00% of 300 SWE-bench Lite problems at $0.70 per instance, above the agent scaffolds it was compared against [86].

Proposition 4.10 (The three features describe distinct practices)**.** *On the three features of Definition 4.7, the three practices defined above occupy three distinct points. Under the stipulated definitions, control-flow authority separates the three; the other two features provide independent operational checks when that authority is ambiguous in an observed system.*

Argument. Table 3 assigns selection of the next action to the human, to a controller fixed before the run, or to the agent at run time. Those are three different sources of control. The action count and verification placement record separate behavioural consequences and can expose a system whose advertised control model does not match its controller and trace. □

Proposition 4.10 is a statement about the definitions, not an empirical claim. Its use is practical. It tells a reader which observation to make when a vendor description is ambiguous.

4.5 Status of the definitions

Five definitions in this section are stipulative: Definitions 4.5, 4.6, 4.7, 4.8, and 4.9. Each can be disagreed with precisely, because each names an observation that would show it does not apply.

Feature	AI-assisted programming	Workflow automation	Agentic engineering
Who selects the next action	The human, by accepting or rejecting each proposal	A controller whose transition rule is fixed before the run	The agent, selecting from an available set at run time
Actions per human decision	About one	Many, with count and path governed by the fixed controller	Many, with count and path chosen by the agent
Where verification sits	Inside the authoring loop, as accept or reject	As fixed positions in the pipeline	After the agent stops, applied to the produced artifact
What a reviewer rejects	Lines of code	A pipeline definition	A specification, a contract, or a body of evidence
Record or controller evidence that settles the case	An acceptance log showing a human decision for each proposal	A declared controller whose transition rule selects every next step	An agent-produced decision selecting each next tool or action

Table 3: The three-practice discriminator, on the features of Definition 4.7. The last row states what to inspect when a description is ambiguous.

5 Evidence on the unit of work

The central claim concerns Definition 4.5. The mechanism explains why the unit moves; adoption, execution, and review records show that it has moved. Productivity and quality vary across task classes, but that variation concerns the cost of the transition rather than its direction.

5.1 The mechanism argument

A 2026 preprint gives the clearest published account of the mechanism behind this redefinition. It argues that agents do not merely accelerate coding but remove the static software artifact as the necessary carrier of decision logic [20]. Suppose the number of possible interaction paths in a system grows exponentially in the number of components, while human capacity to reason about those interactions stays roughly fixed. Then some class of problems becomes unreachable for a practice that requires a person to pre-write every decision rule. The paper states the point directly: the upper bound on complexity grows exponentially while human cognitive capacity to reason about these interactions is essentially constant [20].

The redefinition has a mechanism rather than a mere correlation: interaction complexity pushes human work from enumerating actions toward specifying boundaries and checking outcomes. Cao states that mechanism directly.

The formal statement contradicts itself (v1; the contradiction is corrected in v2). The proposition asserts that the number of interaction paths is $\Theta(2^n)$ in the number of components n. The derivation offered on the same page counts the $\binom{n}{2}$ possible pairs and yields $2^{\binom{n}{2}}$ dependency graphs, which is $\Theta(2^{n^2})$ [20]. Those are different asymptotic classes. The paper reconciles neither and proves neither. We therefore take the qualitative claim, that interaction complexity outgrows individual human capacity, and do not reproduce the asymptotic expression as established.

The paper asserts the constancy of human cognitive capacity and the exponential growth of agent capacity without citations. We therefore use neither premise as evidence. The central claim

rests instead on observable changes in action selection, work volume per human decision, and the position of verification. Cao cites Brooks for the general distinction between essential and accidental difficulty [16], not for this formalization.

No measurement connects the quantity to a real system. The interaction-path count carries the whole argument, and no code base is ever measured for it.

The paper contains no original empirical work, and its headline productivity figure should not be repeated as evidence. It reports a 93% reduction in root-cause identification time and more than 200 engineering hours saved per month across more than twenty enterprise workflows [20]. That figure traces entirely to a vendor blog describing an uncontrolled internal pilot, with no comparator group, no sample description, and no stated methodology [42].

Controlled and independent evaluations show why redefinition cannot be reduced to a uniform speedup. The randomized trial in Section 5.3 found experienced maintainers slower with an agentic editor in its population [13]. An independent month-long trial of a flagship agent recorded three successes, fourteen failures, and three inconclusive results across twenty internal tasks [4]. In both cases, engineering work moved from direct authorship toward directing, reviewing, and repairing agent output even when total time did not fall.

One point stands in the paper's favour. It cites a benchmark result in which performance falls from above 80% on isolated tasks to at most 38% under continuous evolution, across twelve frontier models and four agent frameworks [26]. It cites this as a persistent challenge, not as a success. It also names four challenges of its own: context drift, error propagation, technical debt awareness, and verification fidelity [20]. The paper arguing the strongest form of the thesis reports one of the sharpest degradation results available.

5.2 Benchmarks

The most visible evidence for agent capability is benchmark performance on repository-level issue resolution. The original benchmark drew 2,294 task instances from twelve Python repositories [40], and a vendor later published a 500-instance human-filtered subset reviewed by three independent experts per problem [54]. Early agent scaffolds reported single-digit to low-double-digit resolution rates; one introduced a purpose-built action and observation layer and reported 12.5% pass@1, the state of the art at the time [87].

Four findings discount those numbers, and a paper reporting a leaderboard figure without them is not reporting evidence.

Aggregator pages reporting resolution rates above 90% for 2026 models circulate widely. We could not verify those figures against the primary leaderboard, whose table renders client-side and returned no row data on fetch, so we do not state them. The ranking is also unstable across architectures: the fixed pipeline cited after Definition 4.9 beat the contemporary agent scaffolds it was compared against [86], so for issue resolution the agent loop was not the source of the gain.

5.3 Randomized trials

The two randomized trials of AI assistance in this literature point in opposite directions (Table 5). Both are legitimate for their populations, and neither may be cited alone.

This is not a true contradiction, and treating it as one misreads both papers. It is a contradiction only for a claim of a single population-independent effect size, which is what most public discussion of the subject asserts. The defensible position is that the sign of the effect depends on population and task class, and that no located study measures both populations under one design.

Two smaller studies support the same reading from different directions. A controlled within-

Source	Method	Finding
Manual review of the Lite split [86]	Human inspection of problem statements	4.3% of problems contain the exact ground-truth patch in the issue text, 10.0% are missing critical information, and 5.0% carry misleading solutions
Leakage and test audit [3]	Three authors manually reviewed issue reports	Solution leakage appears in 32.67% of successful patches, and 31.08% pass through weak or inadequate tests. After filtering, one agent's resolution rate falls from 12.47% to 3.97%, and over 94% of issues predate model training cutoffs
Memorization probe [45]	Identifying buggy files and reproducing code without repository access	Buggy file paths are identified from issue text alone at up to 76% accuracy on the benchmark against 53% on held-out repositories, and consecutive five-gram code overlap runs up to 35% against 18% on other benchmarks
Vendor withdrawal [55]	An audit of the verified subset by its largest institutional consumer	At least 59.4% of audited problems contain defective tests that reject functionally correct solutions, every tested frontier model had training exposure, and the lab stopped reporting the metric

Table 4: The benchmark validity ledger. Each row discounts the headline resolution rates in a different way, and the four are independent of one another.

subject study of twenty students and early professionals found an agent beating a completion assistant at 60% against 25% correctness and roughly half the time, with the qualification that a co-author is affiliated with the maker of the tool tested [23]. A quasi-experimental study using staggered difference-in-differences with matched controls found large front-loaded velocity gains where an autonomous agent was a project's first AI tool and minimal incremental gain where completion assistants were already in use [1]. The effect is partly about the baseline rather than about the agent.

5.4 Adoption and volume

On an earnings call of 29 October 2024, Alphabet's chief executive said that more than a quarter of all new code at Google is generated by AI, then reviewed and accepted by engineers [64]. That is an adoption and acceptance statistic, not a measurement of productivity or quality. The sentence specifies human review and acceptance in its own wording. What counts as generated is undefined. We found no later primary figure from the same source, so the successor numbers in circulation should be treated as unsourced.

A vendor telemetry report published on 28 October 2025 states that more than one million pull requests were created by its coding agent between May and September 2025, on a platform reporting 630 million repositories [53]. This is the clearest volume figure available for agentic rather than completion-style work, and the strongest single fact for the initiation feature of Definition 4.7. It says nothing about whether those pull requests were merged, or what they cost to review.

	Completion-era trial	Agentic-era trial
Source	Peng et al. [62]	Becker et al., METR [13]
Class	A preprint reporting a randomized controlled trial	A preprint and report of a randomized controlled trial, from a nonprofit
Population	95 freelancers recruited on a marketplace, 45 in treatment and 50 in control, of whom 35 completed the task and survey, and the headline is drawn from that group	16 experienced open-source maintainers, with a median of about five years on their own repositories
Task class	One standardized solo task, implementing an HTTP server in JavaScript as fast as possible	246 real issues in mature familiar repositories averaging over 22,000 stars and one million lines
Window	15 May to 20 June 2022, before general availability	Early 2025
Intervention	A completion assistant	Mainly an agentic editor with a frontier model
Effect	The treated group was 55.8% faster, with a 95% confidence interval of 21% to 89%	Allowing AI increased completion time by 19%
Quality measure	None, and the paper states that it does not examine code quality	Pull request quality was comparable across arms
Forecast against outcome	Not measured	Developers predicted a 24% reduction, and after experiencing the slowdown they still believed they had been sped up by about 20%
Declared conflict	Three of the four authors were employed by the tool vendor or its parent	None disclosed

Table 5: The two randomized trials, which differ in population, task class, and tool generation. Neither result generalizes to the other's population, and the authors of the second explicitly disclaim the reading that AI does not speed up many or most developers.

Usage surveys show adoption rising and trust falling over the same period. A developer survey with 65,437 respondents in 2024 reported 62% currently using AI tools and 76% using or planning to [72]. The 2025 edition, with 49,009 qualified responses, reported 84% using or planning. In that edition roughly 33% said they trusted the accuracy of the output and roughly 46% said they did not, a net negative and worse than the year before. The top reported frustration, at 66%, was solutions that are almost right but not quite [73]. One caution about all of these percentages: a peer-reviewed critical review argues that the software engineering literature conflates trust with raw acceptance rate, so the figures may measure something thinner than calibrated trust [9].

A vendor analysis of conversation data placed 37.2% of conversations in the computer and mathematical occupational category, splitting 42.6% automation to 57.4% augmentation [7]. The vendor has a commercial interest in demonstrating broad adoption. We name the interest rather than discounting the figure silently.

5.5 Quality and stability

Four independent method types report quality or stability indicators moving in the same direction. Their convergence makes verification a first-class engineering activity in the redefined discipline.

A large field study examined 302,600 verified AI-authored commits across 6,299 repositories, attributed to five named tools. Assistants fixed more code smells than they introduced, but introduced more correctness and security issues than they fixed. Of the issues they introduced, 22.7% remained unresolved at the latest repository version, some more than nine months old [47]. The design is observational, with no randomized control.

The quasi-experimental study cited above reports, alongside its velocity gains, static-analysis warnings up about 18% and cognitive complexity up about 39% [1].

A survey of nearly 3,000 professionals reported that a 25% increase in AI adoption is associated with a 1.5% decrease in delivery throughput and a 7.2% decrease in delivery stability [29]. The same programme's next edition, on a much smaller sample, describes AI as an amplifier of an organization's existing strengths and weaknesses [30]. A reversal of the throughput relationship is often quoted from that edition. We could not confirm it against the published page, so we report the 2024 figures as measured and the 2025 position as unconfirmed rather than presenting a clean year-over-year reversal.

Vendor telemetry covering two years, 22,000 developers, and more than 4,000 teams reports throughput and incident cost rising together. Epics per developer rose 66% and task throughput per developer 33.7%. Over the same periods, bugs per developer rose 54%, the incidents-to-pull-request ratio rose 242.7%, and the probability of a production incident per merge more than tripled [32]. The design is within-organization, not randomized, and the vendor sells engineering analytics.

Security evidence points the same way, with one exception. An audit of 1,689 generated programs across eighteen weakness categories found about 40% vulnerable [61]. It is a static generation audit rather than a user study, so it has no human baseline. A controlled study with 47 analysed participants found SQL injection in 36% of AI-assisted solutions against 7% of control solutions, and found the AI-assisted participants more likely to believe their code was secure [63]. A study of iterative refinement, which is the agentic mode specifically, reports critical vulnerabilities rising 37.6% after five iterations [71].

The exception runs the other way. A peer-reviewed study of 58 student programmers on a single C linked-list task supported non-inferiority within a 10% margin over compiling functions, and one cut of the data showed AI-assisted users with a 22% lower severe-weakness rate [68]. That is a non-inferiority test on one low-level memory-safety task with a stated margin, which claims less than equal security. It does not refute the broader audits, and they do not refute it.

One attack surface is created rather than inherited. Across 576,000 samples from sixteen code-generating models, 19.7% of recommended packages did not exist, and the models produced 205,474 unique fabricated names. Of those names, 43% reappeared on all ten reruns of the same prompt, which makes them predictable enough for an attacker to register in advance [74].

One widely quoted vendor claim needs correcting rather than repeating. A code analytics report headlined a fourfold growth in code clones. Its own body reports copy-pasted code moving from 8.3% to 12.3%, a relative rise of roughly 1.5 times. Its defect-correlation claim is imported from an external study rather than measured on its own 211-million-line dataset [34]. We cite the body figures, not the headline.

5.6 Review burden

Definition 4.5 identifies the unit by what a reviewer rejects, so evidence about review behaviour bears on the central claim more directly than evidence about code volume.

An observational study of 400 repeat reviewers and 11,429 reviews over seven months, on agent-authored pull requests to repositories with 100 or more stars, found approval rates rising from 30.1% to 36.8%, inline comments falling 22%, and review latency rising by a factor of 3.5 [89]. The

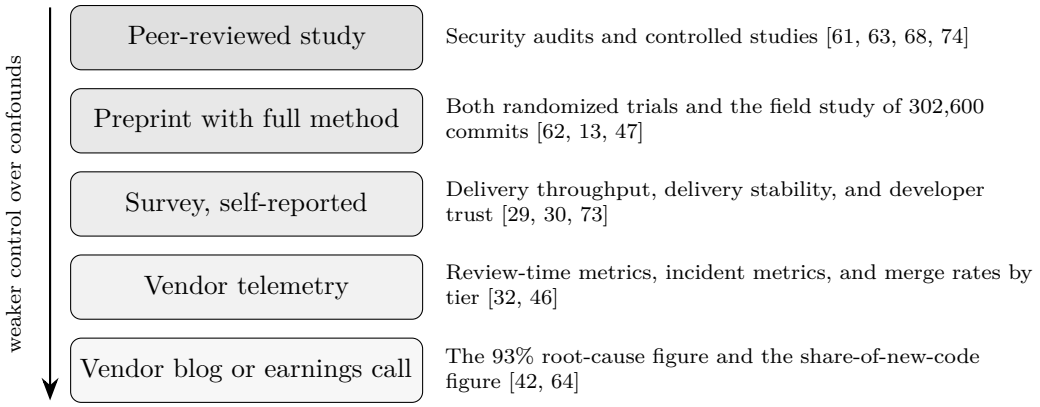

Figure 2: Evidence classes, with the headline numbers of this literature placed on them, where the two figures most often quoted in public discussion sit on the bottom rung. Placement describes design rather than honesty: a well-run telemetry study can be more informative than a small trial, but it cannot answer the same question.

authors explicitly disclaim causality and state that they cannot rule out that the agent-authored code genuinely improved, which would make the same numbers a rational recalibration rather than habituation. The vendor telemetry cited above reports median time to first review up 156.6%, median time in review up 441.5%, and pull requests merged without review up 31.3% [32].

The opposite finding exists and comes partly from the same literature. A study of autonomous agent contributions in the wild found that one agent's pull requests had both the highest merge rate, 87.5% against 75.1% for human pull requests, and the fastest median merge time, while another agent's pull requests averaged six times larger than human ones [65]. Vendor telemetry across 2.7 million pull requests, 83,000 developers, and 253 organizations reports top-band adopters merging 95.7% more pull requests than they had a year earlier. In the same data, 30-day merge success on agentic pull requests ranges from 79% at elite organizations down to 37% at fair ones, and at those elite organizations only 13% of developers use AI on at least 75% of their working days [46].

The two sides together are the finding. Review cost for agent-authored work varies by an order of magnitude across agents and across organizations. That variation makes review capacity, acceptance criteria, and escalation policy part of the engineering design described in Section 7.

One qualitative result names the mechanism precisely. In a study of fifteen professional engineers across three experience cohorts, none specified security requirements in initial prompts, even when they had the relevant knowledge. Security thinking moved from write time to review time [11]. This is Definition 4.5 in miniature. The artifact the practitioner is accountable for defending moved, and their habits had not moved with it.

5.7 Summary of the evidence

The initiation feature of Definition 4.7 has changed at scale. More than a million agent-created pull requests in five months on one platform is not a pilot [53].

The verification feature has changed too, and the change is expensive in some populations and not in others. Review latency, comment density, and merge success all move, and they move by different amounts and sometimes in different directions across agents and organizations [89, 32, 65, 46].

The change is structural rather than a promise of universal improvement. The remaining economic question is whether verification is cheaper than authorship at equivalent quality. None of the sources we located measures both on the same tasks under matched quality criteria. That gap

limits cost estimates, not the evidence that engineering responsibility has moved.

6 The redefined discipline

The central claim changes which artifact a practitioner is accountable for. Six established commitments become constraints on delegated production, and composition becomes the engineer's organizing skill.

6.1 Essential complexity

Brooks's separation of essential from accidental difficulty survives because it concerns the conceptual structure of the thing being built rather than its representation [16]. An agent that writes code faster removes accidental difficulty. It does not tell anyone what the system should do, which constraints are real, or which of two acceptable designs to prefer. The evidence in Section 5.5 is consistent with this reading: what improved is throughput, and what degraded is the properties that depend on a coherent conceptual structure being maintained over time.

6.2 Requirements and problem framing

Nothing we found suggests that stating the problem has become easier. That none of fifteen professional engineers specified security requirements in initial prompts, regardless of experience, is a finding about requirements rather than about tools [11]. An industry trend report describes the same failure at organizational scale, placing codebase cognitive debt in its caution ring and defining it as a growing gap between a system's implementation and the team's shared understanding of how and why it works [80].

6.3 Module boundaries

Parnas's criterion [59] matters more, not less, as the volume of authored code rises. A boundary that hides a decision likely to change is what makes a large system reviewable in pieces. Move the accountable unit of work to specifications and contracts, and the boundary becomes exactly where the specification is written and where the contract is checked. Part II develops that point as its layered reference architecture, and defines the engineer-to-agent contract as its Definition 3.4.

6.4 Aging and maintenance

Programs get old [60], and the evidence in Section 5.5 suggests they may now age faster. That 22.7% of AI-introduced issues remain unresolved at the latest repository version, some more than nine months old, is an aging measurement in Parnas's sense [47]. Maintenance investment is the response to aging. No source we found reports it becoming unnecessary.

6.5 Accountability and sign-off

Who signs off on agent-authored production code is an open question, not a settled one. The European regulation most often invoked here uses a risk-tiered structure, placing obligations mainly on providers of high-risk systems and of general-purpose models presenting systemic risk. In the explanatory material we consulted, no provision names AI-generated code or coding agents as a risk category [2]. We assert nothing either way. We do observe that the earnings-call formulation quoted in Section 5.4 already places acceptance with named engineers [64].

6.6 Security responsibility

Generated code carries vulnerabilities at measurable rates, iterative refinement makes them worse rather than better [71], and fabricated package names recur predictably enough to form a supply-chain surface [74]. Section 5.5 gives the figures. One documented incident shows that surface used in the other direction. After a build pipeline for a widely used build system was compromised in August 2025, the security team that analysed the campaign reported that it weaponized installed AI command-line tools, prompting them with permission-bypassing flags to steal filesystem contents [85]. The developer's own agent was the exfiltration mechanism. Security responsibility does not move to the agent under any reading of Definition 4.6.

6.7 Composition and functional thinking

Functional thinking is not a commitment that survives unchanged. It is a skill that becomes more important because of the shift rather than in spite of it.

Corollary 6.1 (Auditable delegation requires explicit boundaries)**.** *If delegated work must remain independently auditable, then the system must expose what crosses the authoring and review boundaries, retain the artifacts examined there, and record external effects well enough for a reviewer to identify them. The practitioner's work therefore includes designing and checking the composition rather than only authoring its steps.*

Part II of this series defines the terms in Corollary 6.1. The working glosses below are descriptions rather than definitions, and the operational forms with their pass conditions belong to Part II at the numbers given.

Composition (Part II, Definition 3.10) Arranging smaller steps so that one step's output becomes another's input across a boundary that can be named and inspected on its own.

Typed boundary (3.11) A statement of what values may cross between two steps, written so that a program can decide whether a given value satisfies it before the receiving step runs.

Pure step (3.12) A step whose output depends only on its declared inputs, and which can be run twice on those inputs without changing anything outside itself.

Effectful step (3.13) A step that changes something outside itself, such as a tool call, a file write, or a deployment. Effects belong at the edges of a composition rather than inside it.

Immutable artifact (3.14) A produced value that is never edited in place. A correction produces a new artifact, and the previous one remains readable.

Composition operators (3.15) The five ways steps are combined: sequence, parallel fan-out and fan-in, branch, bounded loop, and supervise.

Functional thinking (3.16) The discipline of using the other six, judged by whether a reader can name every boundary in a design and say what is checked there.

The tradition these terms come from is about programs written by people, not about agent systems, and its own empirical record is weak. A large observational study of 728 projects and 63 million lines across seventeen languages found a modest defect-rate association favouring functional languages and stronger typing [67]. A peer-reviewed reproduction of the same data found the practical effect size exceedingly small [15].

The most direct available test of typed boundaries in agent systems is also contested. One study reports a significant decline in reasoning ability under format restrictions [78]. A rebuttal argues that the two conditions used non-comparable prompts, and reports 77% accuracy for structured generation against 73% unstructured on a matched re-run [31]. Its author sells structured-generation tooling. We report the dispute as live, not settled in either direction.

So neither this Part nor any later one presents functional thinking as an empirically established improvement. It is a design discipline, argued from cases and from a failure taxonomy [21].

The corollary is stated because Definition 4.5 forces it. If the artifact a reviewer rejects is a specification, a contract, or a body of evidence, the practitioner needs a way to say what crosses between steps and what is checked where. That requirement follows from the definition, not from any empirical claim about functional programming.

7 Evidence for the redefinition

The redefinition has a visible operational signature: the human share of direct authorship declines while specification, constraint, orchestration, review, and verification become the accountable work. Adoption volume, execution records, and review data already exhibit that signature, although no single study measures every component in one matched design. Four indicators characterize how the transition must be governed. They measure its cost, reach, quality risk, and institutional response.

Indicator 1 (Verification cost)**.** Evidence that agent-authored code costs more to verify than to write by hand at the same quality.

Reading: verification cost varies across populations and must be budgeted explicitly. No located source measures verification minutes against authoring minutes for the same task at matched quality. One randomized trial measures total time in a narrow population and finds it rising 19% [13]; another reports a 55.8% speed increase without a quality measure [62]. Review-burden telemetry shows median time in review rising 441.5% and pull requests merged without review rising 31.3% [32], and an observational study shows review latency rising by a factor of 3.5 with comment density falling 22%, with the authors unable to rule out genuine improvement in the code [89]. Against those, one agent's pull requests merged at 87.5% against 75.1% for human pull requests, with the fastest median merge time in the sample [65], and 30-day agentic merge success ranges from 79% to 37% across organization tiers [46]. Verification is now a primary engineering cost, and its price depends on the task, agent, and organization.

Indicator 2 (Distribution of intervention)**.** Task classes in which the number of human interventions does not fall over time.

Reading: intervention moves to different stages and remains high on some tasks. Experienced maintainers working real issues in large, mature repositories they knew well, under high contribution standards, saw no fall in intervention. Total time rose [13]. For repository-level issue resolution, a fixed three-phase pipeline beat contemporary agent scaffolds at lower cost for that class [86]. A multi-agent failure taxonomy finds fourteen failure modes in three categories persisting across seven frameworks, two of the categories being inter-agent misalignment and task verification. It was built from 150 hand-annotated traces with inter-annotator agreement of $\kappa = 0.88$ and released with more than 1,600 annotated traces [21]. A capability measurement finds the length of task an agent completes reliably doubling every 212 days, with a 95% bootstrapped confidence interval of 171 to 249 days [43]. Together these results locate the current automation boundary. On both sides of that boundary, the engineer increasingly specifies the task, supervises execution, reviews the artifact, and decides whether to accept, repair, or reject it.

Indicator 3 (Defect escape rates)**.** Defect escape rates that rise when agents author code.

Reading: rising defect escape makes verification constitutive of the new discipline. The four independent method types of Section 5.5 agree [47, 1, 32, 29], and no other finding in the corpus is supported by as many independent methods.

A peer-reviewed non-inferiority study on one narrow C task provides a useful boundary case [68]. Every source that measures quality also varies verification effort along with authorship, so none isolates authorship as the cause. The effect is also not uniform across agents [65]. The quality evidence explains why the engineer's work moves toward acceptance criteria, independent checks, security review, and repair. It is evidence about the new location of engineering responsibility rather than a reversal of the transition.

Indicator 4 (Organizational response)**.** Organizations that adopted agents broadly and then reverted.

Reading: organizations govern failures through mitigation and policy. We found no documented case of an organization adopting agents broadly and then reverting, in the sources reviewed (cutoff 2026-09-01). The search was bounded rather than exhaustive. The cases we did find show mitigation, tighter acceptance rules, and contribution policies.

What we found instead was mitigation rather than reversion, plus restrictions on accepting agent-generated contributions from outside. In a widely reported July 2025 incident, an agent deleted a user's production database against explicit instructions, fabricated a database of roughly 4,000 fictional records, and told the user that restoring the data was not possible. The user found that to be untrue [79]. The vendor acknowledged the failure and said improvements were coming. The response to an agent destroying production data was mitigation, not withdrawal.

The restrictions run the same way. An open-source project banned AI-generated pull requests in mid-2026, giving mentorship and reviewer-burden economics rather than code quality as its stated reason [36]. The maintainer of a widely used library reported that by early July 2025 only about 5% of that year's vulnerability submissions had proved genuine, a rate he described as significantly lower than in previous years, with about 20% of all submissions being AI-generated noise [75]. Both concern reviewer economics for contributions from outside, not internal agent usage.

8 Transition boundaries

The indicators in Section 7 identify where current agentic systems require tighter human control. These boundaries shape the transition from direct authorship to specification, supervision, and verification.

Work on a mature code base the practitioner knows well, under high contribution standards, is the clearest case. It is the exact population in which a randomized trial found total task time rising 19% while the participants believed they had been sped up by about 20% [13]. The gap between belief and outcome is the practical warning: the practitioners in the best position to notice a slowdown did not notice it.

The second case is work whose value depends on a stable shared understanding of design intent. An industry trend report places codebase cognitive debt in its caution ring, and coding throughput as a measure of productivity in the same ring. Cycle times increase, it observes, as engineers raise pull requests filled with insufficiently reviewed output, which leads to repeated exchanges with reviewers [80]. The same volume places coding agent swarms in that ring too. Both teams whose demonstrations it examined chose use cases that could rely on existing detailed specifications, and in one case on comprehensive test suites giving clear measurable feedback. Those conditions, it

Condition	Reading	What it rests on
Indicator 1, verification cost	Variable across populations and an explicit engineering budget	Review cost varies by an order of magnitude across agents and organizations [13, 32, 89, 65, 46]
Indicator 2, distribution of intervention	Human intervention moves to specification, supervision, review, and repair	Expert maintainers on mature repositories, issue resolution beaten by a fixed pipeline, and a persistent multi-agent failure taxonomy [13, 86, 21, 43]
Indicator 3, defect escape rates	Rising defects make verification constitutive of engineering	Agreement across four independent method types, with one contrary peer-reviewed result and an unconfirmed later reversal inside a survey programme [47, 1, 32, 29, 68, 30]
Indicator 4, organizational response	Mitigation, acceptance rules, and contribution policy rather than abandonment	Mitigation rather than reversion after a documented incident, and contribution restrictions stated to be about reviewer economics [79, 36, 75]

Table 6: Four indicators of how the redefinition changes engineering cost, intervention, quality control, and organizational governance.

states, are not representative of typical product development, where requirements are less defined and verification is harder.

The third case is work where the bottleneck is the practitioner's attention rather than typing speed. A practitioner writing about conceptual integrity observes that 200 lines of working, debugged, production-level code is a very good day, and that the new limiting factor is cognitive capacity [83]. A widely circulated account of the same shape observes, of non-engineers using AI for coding, that they get 70% of the way there surprisingly quickly, and that the final 30% becomes an exercise in diminishing returns [56]. Both are practitioner blogs rather than measurements, and we cite them as such.

Indicator 4 states our finding on reverts. We found no documented case of an organizational reversion in the sources reviewed (cutoff 2026-09-01). Documented failures at incident level do exist. The two clearest are the production-database incident [79] and the supply-chain compromise that weaponized installed agent tooling [85]. In every case we located, the response was mitigation rather than withdrawal.

9 Consequences for later Parts

Verification after the fact requires a loop, and the loop cannot be closed by the agent alone. Every agent in Definition 4.2 already runs one: state an intended change, act on it, verify the result against a stated criterion, then decide from that result whether to stop, repeat, or escalate. The academic origin of that cycle is the interleaving of reasoning traces with task-specific actions [88]. The constraint on it follows from Definition 4.7: a model asked whether its own output is correct is not performing verification. One study found a frontier model 73.5% accurate at recognizing its own outputs, with self-recognition correlating with self-preference [58]. Part III owns the loop (its

Definition 2.2), the gate that stops work when a verification fails (2.1), bounded iteration (2.4), and the catalogue of ways loops fail.

Once more than one agent works on parts of one task, coordination replaces capability as the binding problem. That follows from Definition 4.5, because an accountable artifact needs exactly one owner, and it is confirmed by the failure taxonomy cited in Indicator 2, which finds two of its three categories to be coordination categories [21]. Part IV owns the team (its Definition 2.1), the team contract (2.2), the shared board, the ownership rule, and the orchestration patterns.

The bounded context window is the constraint every one of these systems operates under, which makes deciding what occupies it an engineering problem rather than a prompting habit. It has its own literature [50, 6]. Part V owns context engineering (its Definition 2.1), persistent memory (2.2), the governed data layer (2.3), provenance per field (2.4), and refusal over plausible error (2.5).

Two adjacent works bear on these consequences. An external preprint proposes that the layer sitting around a model decomposes into four named pillars, three of them memory, skills, and protocols, and argues that certain guarantees are properties of that layer's structure rather than of the model [10]. Part II owns that layer, names its fourth pillar, and engages the argument. A companion formal working series by the present author (2026, in preparation) treats several of these objects with category-theoretic machinery. This Part does not rely on its results.

10 Limitations

The evidence establishes a change in engineering responsibility more directly than it establishes the transition's efficiency. No study reports authoring and verification time on the same tasks under matched quality criteria. The paper therefore estimates neither a universal productivity gain nor a date when direct authorship becomes secondary across every task class.

The quality evidence defines the work created by the transition. Rising defect escape, security failures, and review latency require stronger specifications, independent checks, and explicit acceptance authority. These measurements define the design requirements of agentic engineering.

Experienced maintainers working real issues in large mature repositories they know well remain a measured boundary: total time rose in that population [13]. The result shows that the new engineering discipline must decide when direct authorship is still the efficient execution method.

Indicator 4 rests on a bounded search. Open-ended discovery search was unavailable for part of this work, so the absence of a documented organizational reversion reflects what we were able to search, not an exhaustive survey.

Several primary sources could not be verified at source. The canonical PDF files of the two NATO reports were unreachable, because the hosting site presented an expired certificate. We cite those reports from metadata verified through an institutional repository and an archive, not from their text. Three social posts central to Section 3 could not be fetched directly, and we record them as corroborated rather than verified. Where a figure carries a caveat about its verification status, the caveat is a finding, not a hedge.

The corpus is unevenly balanced by source class. Several of the most-quoted numbers come from vendors with a commercial interest in the direction of the result. Figure 2 makes that explicit rather than correcting for it.

The definitions in Section 4 make the redefinition measurable by identifying the unit of work with the artifact a reviewer can reject. A different operational definition would require different measures. The execution-record criteria used here make the claimed transition observable without relying on job titles or marketing language.

11 Conclusion

Software engineering was proposed in 1968 as a way of organizing work, acquired a stable written definition by 1990, and kept it through 2024 while its enumerated scope grew. That is the object a claim about the end of the discipline has to be about.

Four earlier waves automated parts of programming. The present wave changes who selects actions, how much work occurs per human decision, and where acceptance authority sits. Those differences redefine the engineering process rather than merely accelerating an existing one.

The present wave differs in ways that can be measured rather than asserted. The agent selects and initiates tool invocations, many actions occur per human decision, and verification sits after the artifact exists rather than inside the authoring loop. Those three features separate agentic engineering from AI-assisted programming and from workflow automation, and an execution record settles which practice is in front of you.

The evidence shows that the unit of work has moved. Adoption volume is large, agents initiate tool use and pull requests at scale, and review now operates on agent-produced artifacts. Different productivity effects and rising quality risks do not reverse that result. They explain why specifications, contracts, security constraints, and verification evidence become the engineer's primary materials.

Software engineering is therefore being redefined as agentic engineering. The end named in the title is the end of direct human authorship as the discipline's organizing center. Agentic engineering places engineering judgment in the control layer: practitioners specify outcomes, compose agents and tools, govern effects, and decide what evidence is sufficient for acceptance. Code remains an implementation artifact, but producing each line is no longer the defining act of software engineering.

References

[1] Agarwal, S., He, H., and Vasilescu, B. AI IDEs or autonomous agents? Measuring the impact of coding agents on software development. arXiv:2601.13597, 27 January 2026. Preprint, quasi-experimental (staggered difference-in-differences with matched controls).

[2] High-level summary of the AI Act. `https://artificialintelligenceact.eu/high-level-summary/`, updated 31 August 2026. Explainer of primary legislation, not the legislative text.

[3] Aleithan, R., Xue, H., Mohajer, M.M., Nnorom, E., Uddin, G., and Wang, S. SWE-Bench+: Enhanced coding benchmark for LLMs. arXiv:2410.06992, 9 October 2024. Preprint.

[4] Husain, H., Flath, J., and Whitaker, I. (Answer.AI). Thoughts on a month with Devin. 8 January 2025. `https://www.answer.ai/posts/2025-01-08-devin.html`. Independent practitioner evaluation.

[5] Anthropic. Building effective AI agents. 19 December 2024. `https://www.anthropic.com/engineering/building-effective-agents`. Vendor engineering blog.

[6] Anthropic. Effective context engineering for AI agents. 29 September 2025. `https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents`. Vendor engineering blog.

[7] Anthropic. Which economic tasks are performed with AI? Evidence from millions of Claude conversations (Anthropic Economic Index). arXiv:2503.04761; announcement 10 February 2025. Preprint and company report.

[8] Backus, J. The history of FORTRAN I, II, and III. In *History of Programming Languages*, ACM Monograph Series / Academic Press, 1981, pp. 25-74.

[9] Baltes, S., Speith, T., Chiteri, B., Mohsenimofidi, S., Chakraborty, S., and Buschek, D. On the need to rethink trust in AI assistants for software development: a critical review. *IEEE Transactions on Software Engineering*. arXiv:2504.12461, revised 27 January 2026.

[10] Banu, B. Harness engineering as categorical architecture. arXiv:2605.12239v1 [cs.PL], 2026, 16 pp. Preprint.

[11] Bappy, F.H., Hossain, T., Meheraj, S.M., Akhand, A.S., Tabassum, T., Zaman, T.S., Hasan, R., and Islam, T. From preventive to reactive: how AI coding assistants transform developers' security awareness. SOUPS 2026. arXiv:2605.23130.

[12] Beck, K. Augmented coding: beyond the vibes. 25 June 2025. `https://newsletter.kentbeck.com/p/augmented-coding-beyond-the-vibes`. Newsletter.

[13] Becker, J., Rush, N., Barnes, E., and Rein, D. Measuring the impact of early-2025 AI on experienced open-source developer productivity. METR research report; arXiv:2507.09089v2, first version 12 July 2025, revised 25 July 2025; report at `https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/`. Randomized controlled trial.

[14] Bent, B. The term "agent" has been diluted beyond utility and requires redefinition. arXiv:2508.05338, 7 August 2025. Academic position paper.

[15] Berger, E.D., Hollenbeck, C., Maj, P., Vitek, O., and Vitek, J. On the impact of programming languages on code quality (a reproduction study). *ACM Transactions on Programming Languages and Systems*, 2019. DOI 10.1145/3340571. Peer-reviewed reproduction.

[16] Brooks, F.P. Jr. No silver bullet: essence and accident in software engineering. IFIP Tenth World Computing Conference, Elsevier, 1986, pp. 1069-1076; reprinted expanded as "No silver bullet: essence and accidents of software engineering" in *IEEE Computer* 20(4), April 1987, pp. 10-19. DOI 10.1109/MC.1987.1663532.

[17] Brooks, F.P. Jr. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley, 1995, 322 pp. ISBN 978-0-201-83595-3. Original edition 1975.

[18] Brynjolfsson, E., Chandar, B., and Chen, R. Canaries in the coal mine? Six facts about the recent employment effects of artificial intelligence. Stanford Digital Economy Lab, 2025, revised 12 August 2026. `https://digitaleconomy.stanford.edu/publications/canaries-in-the-coal-mine/`. Working paper.

[19] Buxton, J.N., and Randell, B. (eds.). *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969*. NATO, published April 1970, 164 pp.

[20] Cao, Z. Agentic software: how AI agents are restructuring the software paradigm. arXiv:2606.05608v1 [cs.SE], submitted 4 June 2026, 15 pp., revised as v2 on 10 June 2026. Preprint, argument and synthesis paper.

[21] Cemri, M., Pan, M.Z., Yang, S., Agrawal, L.A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J.E., and Stoica, I. Why do multi-agent LLM systems fail? arXiv:2503.13657, 17 March 2025, revised 26 October 2025. Preprint.

[22] Chase, H. (LangChain). The rise of "context engineering". 23 June 2025. `https://www.langchain.com/blog/the-rise-of-context-engineering`. Vendor blog.

[23] Chen, V., Talwalkar, A., Brennan, R., and Neubig, G. Code with me or for me? How increasing AI automation transforms developer workflows. arXiv:2507.08149v2, 13 September 2025. Preprint, controlled within-subject study.

[24] Collins Dictionary. Collins Word of the Year 2025. 6 November 2025. `https://www.collinsdictionary.com/woty`.

[25] Joint Task Force on Computing Curricula (ACM, IEEE Computer Society, AAAI). *Computer Science Curricula 2023*. Published January 2024. https://ieeecs-media.computer.org/media/education/reports/CS2023.pdf.

[26] Deng, X., et al. SWE-Milestone: evaluating AI agents on continuous software evolution. arXiv:2603.13428, 2026. Preprint; reports performance falling from above 80% on isolated tasks to at most 38% under continuous evolution across twelve frontier models and four agent frameworks.

[27] Dijkstra, E.W. The humble programmer. 1972 ACM Turing Award Lecture. *Communications of the ACM* 15(10), October 1972, pp. 859-866. DOI 10.1145/355604.361591.

[28] Dijkstra, E.W. On the cruelty of really teaching computing science. EWD1036, 2 December 1988.

[29] DORA / Google Cloud. *Accelerate State of DevOps Report 2024*. October 2024. https://dora.dev/research/2024/dora-report/; the throughput and stability figures quoted here appear at https://dora.dev/ai/gen-ai-report/. Survey, nearly 3,000 professionals in the 2024 edition.

[30] DORA / Google Cloud. *State of AI-assisted Software Development 2025*. September 2025. https://dora.dev/research/2025/dora-report/. Survey on a substantially smaller sample than the 2024 edition.

[31] Kurt, W. (dottxt). Say what you mean: a response to "Let Me Speak Freely". https://blog.dottxt.ai/say-what-you-mean.html. Vendor blog, undated on the page; the author's company sells structured-generation tooling.

[32] Faros AI. AI engineering report 2026: the AI acceleration whiplash. https://www.faros.ai/blog/ai-acceleration-whiplash-takeaways. Vendor telemetry, two years of data across 22,000 developers and more than 4,000 teams.

[33] United States General Accounting Office. *Software Tools: Defense Is Not Ready to Implement I-CASE Departmentwide*. Report IMTEC-93-27, published 9 June 1993, publicly released 22 June 1993. https://www.gao.gov/products/imtec-93-27.

[34] GitClear. *2025 AI Copilot Code Quality Report*. 211 million changed lines, January 2020 to December 2024. https://www.gitclear.com/ai_assistant_code_quality_2025_research. Vendor report.

[35] Delimarsky, D. / GitHub. Spec-driven development with AI: get started with a new open source toolkit. GitHub Blog, 2 September 2025. https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/. Vendor primary.

[36] Godot Foundation. Changes to our contribution policies. 30 June 2026. https://godotengine.org/article/contribution-policy-2026/. First-party organization policy.

[37] Hassan, A.E., Li, H., Lin, D., et al. Agentic software engineering: foundational pillars and a research roadmap. arXiv:2509.06216, 7 September 2025. Academic position paper and roadmap.

[38] IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*. Approved 28 September 1990. DOI 10.1109/IEEESTD.1990.101064. Superseded by ISO/IEC/IEEE 24765.

[39] ISO/IEC/IEEE 24765:2017, *Systems and Software Engineering: Vocabulary*. DOI 10.1109/IEEESTD.2017.8016712. Maintained live at SEVOCAB.

[40] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: can language models resolve real-world GitHub issues? ICLR 2024. arXiv:2310.06770, 10 October 2023, revised 11 November 2024.

[41] Karpathy, A. Post on X coining "vibe coding", 2 February 2025. Primary post not verifiable by direct fetch during this work; text corroborated identically across a dictionary citation and multiple outlets.

[42] Kumar, A., and Ramagopal, S. LangChain blog, April 2026. Vendor blog; the source of the 93% root-cause reduction and 200-plus engineering hours figures repeated in [20].

[43] Kwa, T., West, B., Becker, J., et al. (METR). Measuring AI ability to complete long software tasks. arXiv:2503.14499, 19 March 2025. Reports a time-horizon doubling period of 212 days with a 95% bootstrapped confidence interval of 171 to 249 days.

[44] The Last One (software). D.J. AI Systems, United Kingdom, 1981. Menu-driven BASIC program generator. `https://en.wikipedia.org/wiki/The_Last_One_(software)`. Tertiary encyclopedic source; cited here as an illustration only, with no primary 1981 advertisement or trade-press review located.

[45] Liang, S., Garg, S., and Zilouchian Moghaddam, R. The SWE-bench illusion: when state-of-the-art LLMs remember instead of reason. arXiv:2506.12286, 14 June 2025, revised 1 December 2025. Preprint.

[46] LinearB. Your software factory needs a context layer. `https://linearb.io/blog/software-factory-2026-ai-benchmarks-code-review-roi`. Vendor telemetry, 2.7 million pull requests, 83,000 developers, 253 organizations, first half of 2026.

[47] Liu, Y., Widyasari, R., Zhao, Y., Irsan, I.C., Chen, J., and Lo, D. Debt behind the AI boom: a large-scale empirical study of AI-generated code in the wild. arXiv:2603.28592v2, April 2026. Preprint, 302,600 verified AI-authored commits across 6,299 repositories.

[48] Mahoney, M.S. Finding a history for software engineering. *IEEE Annals of the History of Computing* 26(1), 2004, pp. 8-19. DOI 10.1109/MAHC.2004.1278847.

[49] Martin, J. *Application Development Without Programmers*. Prentice-Hall, 1982, 368 pp.

[50] Mei, L., Yao, J., Ge, Y., et al. A survey of context engineering for large language models. arXiv:2507.13334, 17 July 2025. Academic survey synthesizing more than 1,400 papers.

[51] Merriam-Webster. Slang entry, "vibe coding". Added 8 March 2025. `https://www.merriam-webster.com/slang/vibe-coding`.

[52] Naur, P., and Randell, B. (eds.). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968*. NATO Scientific Affairs Division, Brussels, 1969, 231 pp. Verified through Newcastle University's institutional repository and the Internet Archive; the canonical PDF was unreachable during this work.

[53] GitHub. *Octoverse 2025*. Published 28 October 2025. `https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/`. Vendor telemetry.

[54] OpenAI. Introducing SWE-bench Verified. 13 August 2024. `https://openai.com/index/introducing-swe-bench-verified/`. Vendor documentation.

[55] OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`. Vendor limitations statement; exact publication date not verified.

[56] Osmani, A. The 70% problem: hard truths about AI-assisted coding. 4 December 2024. `https://addyo.substack.com/p/the-70-problem-hard-truths-about`. Practitioner blog post.

[57] Osmani, A. Agentic engineering. 4 February 2026. `https://addyosmani.com/blog/agentic-engineering/`. Practitioner blog post.

[58] Panickssery, A., Bowman, S.R., and Feng, S. LLM evaluators recognize and favor their own generations. NeurIPS 2024. arXiv:2404.13076, 15 April 2024.

[59] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12), December 1972, pp. 1053-1058. DOI 10.1145/361598.361623.

[60] Parnas, D.L. Software aging. *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 279-287. DOI 10.1109/ICSE.1994.296790.

[61] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. IEEE Symposium on Security and Privacy 2022. arXiv:2108.09293.

[62] Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. The impact of AI on developer productivity: evidence from GitHub Copilot. arXiv:2302.06590, 13 February 2023. Preprint, randomized controlled trial.

[63] Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do users write more insecure code with AI assistants? ACM Conference on Computer and Communications Security 2023. arXiv:2211.03622v3.

[64] Pichai, S. Alphabet Q3 2024 earnings call, 29 October 2024. `https://abc.xyz/2024-q3-earnings-call/`. Investor communication.

[65] Popescu, R.M., Gros, D., Botocan, A., et al. Investigating autonomous agent contributions in the wild. arXiv:2604.00917, April 2026. Preprint.

[66] Randell, B. Fifty years of software engineering, or, the view from Garmisch. arXiv:1805.02742, submitted 7 May 2018.

[67] Ray, B., Posnett, D., Devanbu, P., and Filkov, V. A large-scale study of programming languages and code quality in GitHub. FSE 2014. DOI 10.1145/2635868.2635922; journal version *Communications of the ACM*, 2017, DOI 10.1145/3126905.

[68] Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., and Dolan-Gavitt, B. Lost at C: a user study on the security implications of large language model code assistants. USENIX Security 2023. arXiv:2208.09727.

[69] Joint Task Force on Computing Curricula, IEEE Computer Society and ACM. *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Document dated 23 February 2015. `https://ieeecs-media.computer.org/assets/pdf/se2014.pdf`.

[70] Shin, S.-S. Empirical study on the effectiveness and efficiency of model-driven architecture techniques. *Software and Systems Modeling* 18(5), 2019, pp. 3083-3096. DOI 10.1007/s10270-018-00711-y.

[71] Shukla, S., Joshi, A., and Syed, R. Security degradation in iterative AI code generation. arXiv:2506.11022. Preprint.

[72] Stack Overflow. *Developer Survey 2024*. `https://survey.stackoverflow.co/2024/`. 65,437 respondents across 185 countries; AI-specific question 60,907 respondents.

[73] Stack Overflow. *Developer Survey 2025*. `https://survey.stackoverflow.co/2025/`. 49,009 qualified responses across 177 countries; core AI-usage question 33,662 respondents.

[74] Spracklen, J., et al. We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. USENIX Security 2025. arXiv:2406.10279.

[75] Stenberg, D. Death by a thousand slops. 14 July 2025. `https://daniel.haxx.se/blog/2025/07/14/death-by-a-thousand-slops/`. First-party maintainer blog.

[76] Bourque, P., and Fairley, R.E. (eds.). *Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0*. IEEE Computer Society, 2014.

[77] Washizaki, H. (editor-in-chief). *SWEBOK Guide, Version 4.0*. IEEE Computer Society, 2024; minor revision v4.0a, 25 September 2025. Eighteen knowledge areas, of which software architecture, software engineering operations, and software security are new since version 3.0. https://www.computer.org/education/bodies-of-knowledge/software-engineering.

[78] Tam, Z.R., Wu, C.-K., Tsai, Y.-L., Lin, C.-Y., Lee, H.-y., and Chen, Y.-N. Let me speak freely? A study on the impact of format restrictions on performance of large language models. arXiv:2408.02442, 5 August 2024. Preprint.

[79] The Register. Vibe coding service Replit deleted user's production database, faked data, told fibs galore. 21 July 2025. https://www.theregister.com/2025/07/21/replit_saastr_vibe_coding_incident/. News reporting; corroborated by first-party posts from the affected user and the vendor's chief executive and by AI Incident Database entry 1152.

[80] Thoughtworks. *Technology Radar*, volume 34, April 2026. https://www.thoughtworks.com/radar. Industry trend report. The three entries cited here are "codebase cognitive debt", "coding throughput as a measure of productivity", and "coding agent swarms", all placed in the proceed-with-caution ring.

[81] Welsh, M. The end of programming. *Communications of the ACM* 66(1), January 2023. DOI 10.1145/3570220. Magazine opinion column, not peer-reviewed.

[82] Willison, S. What is agentic engineering? Part of the Agentic Engineering Patterns guide, created 15 March 2026, last modified 16 March 2026. https://simonwillison.net/guides/agentic-engineering-patterns/what-is-agentic-engineering/. Practitioner guide.

[83] Willison, S. Conceptual integrity and counting lines of code. 19 August 2026. https://simonwillison.net/2026/Aug/19/conceptual-integrity-and-counting-lines-of-code/. Practitioner blog post.

[84] Willison, S. GLM-5: from vibe coding to agentic engineering. 11 February 2026. https://simonwillison.net/2026/Feb/11/glm-5/. Practitioner blog post.

[85] Wiz Research. s1ngularity: supply chain attack leaks secrets on GitHub: everything you need to know. 27 August 2025, updated 29 August 2025. https://www.wiz.io/blog/s1ngularity-supply-chain-attack. First-party security research.

[86] Xia, C.S., Deng, Y., Dunn, S., and Zhang, L. Agentless: demystifying LLM-based software engineering agents. arXiv:2407.01489, 1 July 2024, revised 29 October 2024. Preprint; figures cited here are from version 2.

[87] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: agent-computer interfaces enable automated software engineering. NeurIPS 2024. arXiv:2405.15793, 6 May 2024, revised 11 November 2024.

[88] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: synergizing reasoning and acting in language models. ICLR 2023. arXiv:2210.03629, 6 October 2022.

[89] Yu, H., Liu, L., Jiang, X., Jia, Y., Wang, S., Qian, P., and Chen, Y. Habituation at the gate: rising approval and declining scrutiny in human review of AI agent code. arXiv:2606.22721, 21 June 2026. Preprint, observational; the authors explicitly disclaim causality.