

Why AI-native software development requires a new operating model

October 22, 2026

AI is changing the economics of software development faster than most organizations are changing how they manage it. Coding agents can now produce much of the routine implementation that once consumed an engineer's working day. Yet many established software businesses still plan, staff, and release as if typing code remains the central constraint. This gap creates an opportunity for investors and CEOs, and a risk for companies that confuse faster coding with a better software business.

The central question is how to turn abundant code into reliable commercial software. AI supplies tempo; leadership must supply direction, shared context, testing, security, and accountability. Individual productivity is not organizational productivity. Speed without direction gets you to the wrong place faster.

The wake-up call: pencils down

At Rails World in Austin on September 23, 2026, David Heinemeier Hansson (DHH), creator of Ruby on Rails and co-owner of 37signals, announced that his company had gone "pencils down" on routine hand-written code. His argument is straightforward: for most development teams, continuing to type ordinary implementation code by hand is becoming a poor use of expensive engineering talent.

DHH treats hand-written implementation as an exception that signals an agent workflow fell short. In his Rails World keynote, he reported writing about 30,000 lines of production Ruby annually over the preceding 21 years, versus about 150,000 lines of agent-generated code in August 2026 alone, roughly 60 times his historical monthly pace. He acknowledged that much of the newer code was Rust, a more verbose language. Line counts do not measure delivered value, but the shift in production capacity deserves attention.

I see the implications in the smaller software businesses I assess. Some still organize development around hand-coding. Others have given every engineer a coding agent without changing how teams make decisions, review changes, or run production. The first group leaves productivity on the table. The second accelerates architectural drift and review bottlenecks. "I'll just give everyone Claude and features will build themselves" works far better as a one-person experiment than as a plan for a commercial product with multiple engineers, customers, and regulatory obligations.

The bottleneck moved from creation to assurance

When code is cheap, the scarce resource becomes trustworthy assurance: knowing that teams built the right thing, that the pieces fit together, and that the result can operate safely. Human comprehension remains essential, but exhaustive human inspection of generated code cannot scale with production volume.

Creation and understanding once grew together. An engineer who spent three days writing a feature learned its constraints, dependencies, tradeoffs, and likely failure modes along the way. A reviewer often relied on that accumulated knowledge. An agent can produce the same artifact in minutes without transferring any of that understanding to the person who approves it.

AI breaks the link between producing an artifact and understanding it. As agents generate more changes, the traditional pull-request queue becomes a choke point. Merely adding reviewers will not solve the problem: code volume can grow much faster than human attention. Teams need better specifications, independent checks, test evidence, architectural boundaries, and deliberate human review of consequential exceptions.

Agent-directed work is increasingly asynchronous: specify an outcome, assign the work, and evaluate the result. That makes specification, coordination, and verification central engineering disciplines. A dozen capable agents can produce a dozen locally sensible changes that do not belong in the same product. Someone must own the whole.

The valuable human work increasingly surrounds the code: defining customer problems, choosing what not to build, specifying outcomes, owning architecture, coordinating changes, judging evidence, managing security and compliance, and accepting responsibility for production. Agents should perform much of the implementation and first-pass verification. Humans should decide intent, boundaries, exceptions, and accountability.

No particular coding tool offers a durable advantage. Models, interfaces, and workflows will keep changing. The lasting capability is an organization that adapts while preserving architectural coherence, customer value, and control of production.

Where engineering organizations stand

I see three operating states: teams that have barely adopted agents, teams that adopted them without discipline, and teams that pair agent speed with a coherent operating model. Citizen development creates a separate challenge outside the traditional engineering function.

Companies that have barely started may have excellent products, loyal customers, and skilled engineers. Their delivery model simply assumes that coding capacity is scarce. They still estimate and staff around yesterday's productivity. For an investor, that gap can represent significant untapped operating leverage.

Companies that adopted agents without discipline can generate more pull requests than their organization can absorb. DHH described an experiment at 37signals in which designers generated roughly 20 to 30 Basecamp 5 pull requests. The changes looked reasonable separately, but together they undermined architectural coherence. Faster models do not solve a missing coordination model.

The lesson is larger than one demonstration. Good agents can still disagree about architecture, duplicate capabilities, break shared assumptions, and create integration work. Leaders must decide what belongs in the product before agents implement it.

Adoption stage	What you see	What it costs	The lever
Not started	Headcount and timelines sized for hand-written code	Higher cost per unit of output, slower roadmap	Adopt agents with review and release guardrails
Started without discipline	High output, fast demos, long review queues, frequent rework	Architectural drift, hidden tech debt, security and compliance exposure	Architecture ownership, review discipline, release cadence
Disciplined adopter	Agent speed in development, controlled change in production	Investment in review, testing, and operating rhythm	Scale what works across teams and products

The objective is disciplined adoption: let implementation capacity rise without allowing review queues, technical debt, uncontrolled costs, or production risk to rise with it.

When everyone becomes a developer

AI has also created a second software factory outside engineering. Citizen developers, including employees who have never written conventional software, can use coding agents to build useful applications in hours. This is an opportunity: people closest to the work can test ideas without waiting for an engineering backlog. But a successful demonstration is not the same thing as a secure, maintainable, supported commercial application.

Citizen-developed software may carry customer data, bypass identity controls, duplicate existing systems, rely on fragile integrations, or lack an owner when its creator leaves. A CEO who celebrates demonstrations but ignores production obligations can create a shadow application estate faster than IT can inventory it. This risk grows in regulated businesses, but it also affects ordinary enterprise reliability and cost.

The answer is not to ban citizen development. Give employees safe sandboxes, approved data access, reusable integration patterns, and a clear path from prototype to production. Require a named business sponsor and technical owner before real customers, sensitive data, or core operations depend on an application. An organization should encourage experimentation while making production a deliberate decision.

Coordination becomes an engineering discipline

The faster machines move, the more deliberately people must synchronize.

Every developer can now work at warp speed, but the organization does not move at warp speed by default. Dependencies, conflicting changes, architectural drift, review queues, and duplicated effort pile up faster than before.

The problem is not asynchronous work. Good specifications and written decisions often beat a meeting. The problem is unstructured async: sprawling Slack threads where decisions and assumptions go to die.

Engineering teams need enough human contact to challenge assumptions, transfer practical knowledge, and build shared understanding. This matters especially for early-career engineers and distributed teams, where agent-produced implementations can bypass the learning that comes from working through a problem with colleagues.

High-performing teams need an operating rhythm. A short daily discussion can surface conflicting agent work; design reviews can settle architectural tradeoffs; release reviews can assess risk and evidence; retrospectives can turn failures into improved guardrails. The engineering leader owns that rhythm and the production consequences. The rhythm also reduces the risk that one irreplaceable engineer holds the entire system in their head.

The traditional standup needs to change. "What I did yesterday" reveals little when agents handled much of the implementation. Four questions are more useful:

- **Steering:** what am I directing agents to build, and what outcome is it chasing?
- **Reviewing:** what am I reviewing, and what did I find?
- **Collisions:** where do my changes touch yours?
- **Assumptions:** what did I decide on my own that someone should challenge?

What distinguishes high-leverage engineers now

High-leverage engineers will increasingly frame problems, define constraints, interrogate evidence, resolve architectural collisions, and recognize when a human needs to inspect implementation details. Small teams may not have enough people to assign a separate reviewer to every change. They still need independent checks, clear ownership, risk-based escalation, and a defensible approval trail. We should judge engineers by the soundness of what they ship, not by how much code they personally type or read.

Output volume no longer tells us who is effective. DHH's 60x jump shows capacity, not value. Lines of code, pull requests, and closed tickets primarily measure activity.

These signals still work:

Signal	High leverage	Low leverage
Can they explain it?	Walks through the tradeoffs of what they shipped	"The agent handled that"
What happens after merge?	Changes stick	Reverts, incidents, rework someone else quietly cleans up
How good are their reviews?	Catches real issues	Rubber-stamps
How do they frame work?	Clear specs, work split so changes don't collide	Vague prompts, accepts whatever comes back

Signal	High leverage	Low leverage
Do they say no?	Pushes back on what shouldn't be built	Builds everything because it's free
Do they adapt their workflow?	Constantly experiments and changes how work gets done as capabilities improve	Uses the same agent workflow because it worked six months ago
Do they hold the whole system?	Understands architecture and downstream effects	Understands only the agent's immediate task

When code generation accelerates, someone who can design an effective verification system, coordinate ten parallel work streams, and identify the handful of changes needing expert attention may create more value than another prolific producer. The new 10x engineer might spend less time writing code and more time shaping and trusting outcomes.

The operating rhythm exposes these differences. Ask an engineer to explain a change's purpose, tradeoffs, test evidence, and residual risks. "The agent handled it" is not an adequate answer. The same test belongs in an investor's diligence interviews.

Build at agent speed, ship at the speed the organization can absorb

Faster development does not have to mean faster change in production.

Agent-driven development can coexist with a deliberate release cadence. Some teams ship continuously through strong automated gates; others group changes into scheduled releases. Both approaches work when risk, evidence, and rollback readiness determine what reaches production. A calendar alone is not a quality control.

These distinctions matter most in regulated businesses. A healthcare, financial, or other sensitive application needs traceable changes, demonstrable tests, access controls, and accountable release decisions. SOC 2 and HITRUST expectations do not require a human to read every line of generated code. They do require an organization to demonstrate that its control design works, including appropriate authorization and separation of responsibilities.

The segregation-of-duties problem deserves explicit treatment. If one engineer directs an implementation agent and also approves the result, who counts as the author, the reviewer, and the approver? Calling the agent the author does not create independent human approval. Nor does an AI review automatically satisfy a control that requires distinct people to authorize a change. Auditors will examine the actual control objective and evidence, not the names we assign to agents.

For small teams, I would favor independent adversarial agents, automated test evidence, restricted production access, a named release owner, logs that identify who initiated and approved work, and periodic human sampling. Those mechanisms can provide meaningful compensating controls when separate human reviewers are impractical. Whether they satisfy a particular SOC 2, HITRUST, customer, or contractual requirement depends on the control language, the risk, and the assessor. Where a control

explicitly requires independent human authorization, the company must provide it or formally agree on an acceptable alternative. The goal is stronger assurance, not a clever way to bypass segregation of duties.

I would give agents wide latitude in isolated development environments while requiring evidence-based production gates, risk classifications, and named human accountability.

The old on-call rule was "you build it, you run it." The engineer who wrote the code answered when it broke. Agents do not carry pagers. In an AI-native organization, "you approved it, you own it" is a more useful rule, provided approval means informed acceptance of evidence and responsibility, not a ceremonial click.

Accountability follows the release decision, not the keystrokes. The person responsible must understand the intended behavior, the evidence that supports deployment, the known risks, and the recovery plan. That person can use agents to investigate failures, but cannot delegate away responsibility for the outcome.

From code review to an assurance system

Traditional code review assumes a human can inspect changes at roughly the speed another human produces them. That assumption breaks when agents generate far more code than reviewers can read. One AI agent approving another creates a different danger: two systems with similar prompts, context, or model weaknesses can confidently miss the same error. The next operating model needs an assurance system, not simply faster pull-request reviews.

Martin Odersky, creator of the Scala programming language, asks how we can trust agents at all. His work explores using type systems and constrained capabilities to prevent entire classes of unsafe actions. That is a technical answer to part of the problem. My argument is the organizational complement: leaders must design independent review, measurable tests, controlled authority, and human accountability around systems that produce code at machine speed.

I would organize assurance into five layers:

1. Human intent. Define the outcome, architectural boundaries, safety rules that must always hold, and the business risk before coding begins.
2. Agent implementation. Let the building agent work inside those boundaries and produce code, tests, and a clear account of what changed.
3. Adversarial verification. Give an independent reviewer the specification and changed software, not the builder's persuasive explanation. Direct it to break the change, write failing tests, find security weaknesses, and challenge unnecessary work. Prefer a different model family; at minimum use a fresh session, separate prompts, and independent testing tools. Combine that attack with automated tests, dependency scans, access-policy checks, and security analysis.
4. Evidence and escalation. Preserve test outcomes, attack findings, unresolved issues, changed dependencies, and rollback plans. Escalate failed tests, disagreements, major design changes, or sensitive-data risks to a human.

5. Human accountability. A named person decides what should ship based on the evidence and the risk. Humans control intent, adjudicate exceptions, and take responsibility for production. They do not rubber-stamp every generated line.

Risk determines review depth. Routine, reversible changes with strong test evidence may clear automated gates under an accountable release policy. Authentication, authorization, sensitive data paths, payment logic, shared architecture, and irreversible migrations call for deeper review. A failing test, weak coverage, ambiguous requirement, or disagreement between reviewers must trigger an explicit escalation. Humans arbitrate consequential exceptions rather than pretending they can inspect every generated line.

This is a proposed operating model, not a proven universal standard. Its adversarial reviewer must earn trust by catching seeded defects, identifying unnecessary changes, and finding failures that conventional tests miss. Track false positives as well as escapes. Separate the builder from the reviewer, rotate reviewer prompts or models where useful, and periodically challenge the whole review system. Humans still resolve consequential disagreements. Neither AI prompting AI indefinitely nor AI reviewing AI without human direction is a defensible model.

Measuring organizational velocity

Our measurement systems lag the operating model. Gergely Orosz, author of *The Pragmatic Engineer*, reports a useful counterexample to headline productivity claims: at one company, pull-request frequency stayed flat while the average request became about three times larger. Elsewhere, human reviewers caught changes that agent reviewers missed, including unnecessary implementation and changes far larger than the task required. More generated code can increase review load without delivering more customer value. That is why I would measure the economics of shipping trustworthy changes, not code volume.

A pull request (PR) packages a proposed code change for review. Candidate measures include:

- PR cycle time and time waiting for review
- Review load per engineer, including adversarial review effort
- Adversarial reviewer catch rate on seeded defects, and false-positive rate
- Reverts and rollback rate
- Escaped defects and production incidents
- Rework after merge
- Change failure rate
- Lead time from approved requirement to production
- Share of high-risk changes with documented human review and release accountability
- Key-person concentration: systems only one person can explain or change safely
- Agent spend per engineer per month
- Agent spend per change shipped to production

Compare the measures as a system, not as isolated targets. Faster delivery with stable quality and lower total cost matters; larger changes with slower review, more incidents, or rising agent bills do not. Count the cost of generating, reviewing, testing, and operating software, not only the subscription price.

For an investor, the critical question is not how many coding-agent subscriptions a target buys. Ask whether the business delivers customer value faster without more incidents, runaway costs, growing architectural debt, or dependence on one engineer who alone understands the system. Review actual production changes, not just demos and vendor dashboards.

Hard problems, and where I'd start

1. How do we know the adversarial assurance model earns its keep? The approach described above creates a testable proposition: independent reviewers should catch meaningful errors without burying engineers in false alarms. Measure their effectiveness and use escaped defects and production incidents to challenge the results over time.
2. How do we move capable, often small legacy teams forward without destroying what made them successful? Introduce agents alongside tests, design decisions, observability, and clear ownership. Use human review where risk warrants it rather than imposing a process that a five-person team cannot sustain.
3. Which measures demonstrate real economic improvement? Investors and CEOs should compare end-to-end lead time, review burden, escaped defects, rework, agent spend, and key-person concentration with customer outcomes and operating margin. Spend per engineer and per change that actually reaches production belong in the same scorecard.

What investors and CEOs should do now

For investors evaluating an acquisition and CEOs running an established software business, this is an operating-model question. A strong legacy product may conceal a major productivity opportunity; a fast-moving AI team may conceal technical debt, weak controls, and inflated expectations. Neither a headcount reduction thesis nor a blanket license purchase counts as a transformation plan.

The most promising small software teams combine powerful agents with a few people who understand the architecture, coordinate change, inspect exceptions, and remain accountable for production. The point is not to eliminate engineering judgment. It is to make that judgment govern much more software than one person could ever write.

During diligence, I would ask:

1. How much implementation now comes from agents, and where do humans still hand-code?
2. What checks run before merge, which changes receive human review, and where do queues form?
3. As agent-generated output increased, what happened to review queues, test coverage, incidents, and rework?
4. Can key engineers explain the architecture, deployment controls, and failure modes of what they ship?
5. Who authorizes production changes, and how do release rules vary by risk?
6. How do teams coordinate parallel agent work and record architecture decisions, assumptions, and context?
7. What independent tests and adversarial reviews challenge the work, including whether a change was needed at all?

8. Who holds the architectural mental model, and who can operate the system if that person leaves?

Post-close, the right intervention depends on the starting point. A team that has not adopted agents needs an evidence-backed pilot and safe access. A team with high output and poor coordination needs architectural ownership, independent review, and release discipline. Every team needs an operating rhythm that turns local agent speed into reliable organizational results.

A practical first 90 days

Days 1 to 30: Establish the baseline. Interview the engineers who understand the core product. Map the architecture, release path, critical dependencies, test coverage, incidents, review queues, and key-person risks. Examine representative changes from requirement to production. Inventory AI use inside engineering and citizen-built applications outside it. Ask whether sensitive data reaches unapproved tools. Identify one bounded, high-value workflow where agents can help without risking the business.

Days 31 to 60: Run a disciplined pilot. Give a small cross-functional group suitable coding agents and clear specification, security, testing, and release rules. Name an owner for architectural coordination. Apply the adversarial assurance model described above, with human escalation for consequential changes. Measure end-to-end delivery, reviewer effectiveness, quality, and agent spend per engineer and shipped change.

Days 61 to 90: Decide what to scale. Review evidence with the CEO, engineering leader, and business owners. Expand productive workflows, strengthen weak verification gates, and document decisions and responsibilities. Establish controls for citizen-built applications. Publish a small scorecard for lead time, quality, review effectiveness, and costs. Do not declare victory because engineers generated more code. Demand evidence that customers receive useful changes faster and safely.

A five-person software vendor, a regional healthcare platform, and a highly regulated financial product need different controls. Each still needs an explicit owner for architecture, trustworthy verification, production releases, and the consequences of failure.

The winners will not be the companies that generate the most code. They will be the ones that convert cheap software creation into customer value while maintaining architecture, trust, security, and control. That requires an operating transformation, not a larger AI subscription budget.

References and further reading

David Heinemeier Hansson, Rails World 2026 keynote, September 23, 2026. Agent-first engineering, pencils down, and the economics of implementation. https://www.youtube.com/watch?v=vDjW_dRyKXY

Martin Odersky, creator of Scala, How Can We Trust Our Agents? Technical approaches to trust, types, and capabilities. <https://www.youtube.com/watch?v=t460XVJA944>

Gergely Orosz, The Pragmatic Engineer, What's the ROI of Software Engineers' Spending on AI?
Engineering adoption, review, and costs.
<https://newsletter.pragmaticengineer.com/p/whats-the-roi-of-software-engineers>