develocity

A Model of the AI-Native SDLC

Rebuilding Delivery to Keep Pace with Coding Agents

Hans Dockter · Founder and CEO, Develocity

WHERE DELIVERY IS LIMITED TODAY:

01	02	03	04	05	06	07	08
Intent	Specification	Implementation	**Coding**	**Code review**	**Release**	Deploy	Maintain

OCTOBER 2026

What's inside

THE SHIFT 1

Executive summary 2
Delivery has not kept pace with coding agents

Introduction 4
A model of the SDLC and a deep look at verification

THE MODEL 7

01 The stages 8
The eight stages of the model, and human review

02 Verification across the stages 12
The verification pyramid, and what each run costs

03 Stage by stage 19
Coding, Code review, Release, Deploy, and Maintain

04 The two loops 27
The verification health loop and the eval loop

05 SDLC metrics 30
The measures for the people who run the SDLC

THE DESTINATION 32

06 From delivery-limited to idea-limited 33
The bottleneck moves to deciding what to build

PART ONE

The Shift

Coding got fast. Delivery has not kept pace.

Executive summary

Coding agents have collapsed the time it takes to write code, but delivery has not kept pace. Today, most SDLCs are delivery-limited: the bottleneck is verifying and reviewing what the agents build, and, where there is a Release stage, getting changes through it. An AI-native SDLC is the SDLC rebuilt around what agents can do, so that verification, review, and release keep pace with coding. When they do, the bottleneck moves to deciding what to build.

This paper offers a model of the AI-native SDLC in eight stages, from Intent to Maintain. Its focus is verification, and it goes deepest in Coding, Code review, and Release. My company, Develocity, sells tooling for fast and reliable builds and tests.

Verification is the basis for trusting the agents' work. It covers every check in which a fixed rule decides the result, such as the build, the tests, and static analysis. Unlike a review by an agent, verification is deterministic in that it runs exactly the same logic against the code every time. An agent codes in a loop, running the build and the tests until the checks pass. In most enterprises, much of what experienced engineers know about a codebase was never written down. A test suite captures part of that knowledge: it is the codebase's memory of where the code has broken, and an agent that runs the tests is held to that memory. Agents also make far more changes than people. In a 2026 report on the ROI of AI-assisted software development, DevOps Research and Assessment (DORA) states that AI adoption increases delivery throughput, but also delivery instability.[1]

Verification also drives throughput and cost. Writing the code now takes a fraction of the time it used to, while the build and the tests have not gotten faster along with it, and agents run them far more often than people do. Three properties of verification matter most. The first is speed: the agent, and often the engineer steering it, waits for every run, and more sessions in parallel fill only part of that time. The second is the context that comes with each result: the less the agent has to guess why a check failed, the less time and the fewer tokens it needs. The third is reliability: a false

failure, from a flaky test or a broken environment, is pure waste. At every point where verification runs, a false failure costs time, tokens, and compute, plus salaries when a person has to find out whether the failure is real.

Where there is a Release stage, agents fill each release with more changes. A larger release is harder to verify, and when it fails, the cause is harder to find. One reason organizations batch changes is that part of a release's cost is fixed. I expect releases to get smaller as automation lowers that part, while a person remains accountable for what goes to production.

This model adds two loops beside the stages. The verification health loop keeps verification fast and reliable. The eval loop tests the agents themselves whenever their model, their harness, or their outer harness changes. Where the organization controls such a change, the loop is a gate: it can stop a change with unintended effects before the change takes effect. A human review step can shrink only as far as the evals of the AI step that takes it over are trusted.

NOTES

1 DORA, *State of AI-assisted Software Development 2025*, p. 35, restating the 2024 report: "an estimated 1.5% reduction in software delivery throughput and an estimated 7.2% increase in software delivery instability for every 25% increase in AI adoption." The finding that AI adoption now improves throughput but still increases instability is on p. 4. DORA's 2026 report, *The ROI of AI-assisted Software Development*, restates it (pp. 19 and 25).

Introduction

Much of the talk about AI in the SDLC is about how much faster some parts have become, above all writing code. To turn that into faster delivery, the other parts have to become fast and efficient too.

Andrew Ng writes that with agentic coding, "deciding what to build is the new bottleneck, especially in early-stage projects."[2] For an ideal AI software factory, I think he is right. In such a factory, agents write and review the outputs of each stage, and verification checks the code. People decide what is worth building and direct the system, not each change.[3] In this model, a person still reviews a change wherever its risk or a regulation requires one. Most of today's SDLCs are not such factories yet. They are still limited by delivery, above all by verifying and reviewing what the agents build. Even in such a factory, delivery does not lose its importance: verification and review still carry the trust in the agents' work, and they cost something on every change.

This paper offers two things. The first is a model of the AI-native SDLC in the broad sense,[4] shown at a glance in Figure 1. The second is an in-depth look at verification across the stages, above all in Coding, Code review, and, where there is one, Release. These three stages largely decide whether an SDLC keeps pace with coding, because that is where the agents' code is verified and reviewed. Much of its cost, in tokens, salaries, and compute, arises there too. Review is a topic of its own, which this paper covers mainly where it meets verification. Verification is also the field I know best, and the one my company sells tooling for. That shaped where this paper goes deep, but not what it claims: the model holds without any product in it.

The model draws on many discussions with engineering leaders, on my background in continuous delivery, and on articles such as Anthropic's AI-native SDLC playbook. It shares the playbook's view of the AI-native SDLC. It differs in the stages and in what they do: it has eight instead of six, and verification runs across them instead of being a stage of its own. And this paper focuses on where the agents' work is verified, and goes much deeper there than the playbook. Where the paper needs a reference for how agents work, it uses Anthropic's, for two reasons: Claude Code is the harness I know best, and Anthropic has published

extensively about it and about the SDLC, work that many readers will know. My company has no partnership with Anthropic. Like many companies, it uses Anthropic's products.

Part of what sets an AI-native SDLC apart from a classical one is two loops beside the stages, and an effective AI-native SDLC requires them. The eval loop tests the agents that do the work in the stages. The verification health loop keeps verification fast and reliable: it watches all verification runs, and agents fix the drift it finds, such as flaky tests and slow builds. Anthropic's playbook covers both, but within its stages: it runs evals in its Test stage, and it watches the failure rate of the CI tests in its Maintain stage. This model gives each a loop of its own.

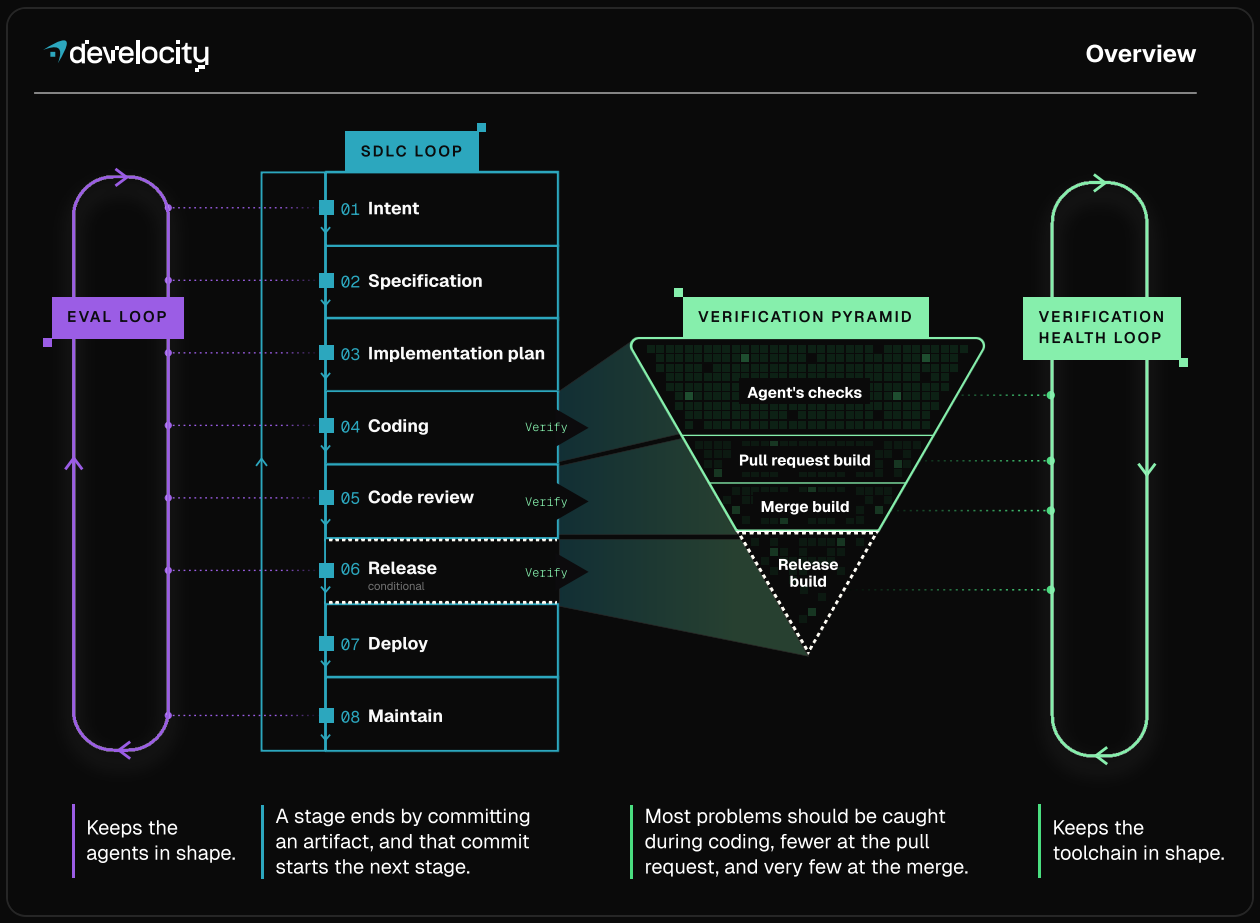


Figure 1. Overview of the model

This paper has six parts:

1. **The stages:** the eight stages of the model, and human review.
2. **Verification across the stages:** why verification matters more with agents, how verification drives cost and throughput, why verification is not a stage, the verification pyramid, and small batches in an AI-native SDLC.

3. **Stage by stage:** Coding, Code review, Release, Deploy, and Maintain.
4. **The two loops:** the verification health loop and the eval loop.
5. **SDLC metrics**
6. **From delivery-limited to idea-limited**

NOTES

2 Andrew Ng, How to Get Through the Product Management Bottleneck, The Batch, 16 July 2025.

3 The term “software factory” comes from practitioners, not from a standard. StrongDM, which calls its own setup a software factory, states its rules as “Code must not be written by humans” and “Code must not be reviewed by humans” (Justin McCarthy, Software Factories and the Agentic Moment, February 2026). In Dan Shapiro’s five levels of AI coding, the highest, the “dark factory,” is “a black box that turns specs into software” (The Five Levels: from Spicy Autocomplete to the Dark Factory, January 2026).

4 Many product people distinguish the product development lifecycle (PDLC) from the SDLC. The software life cycle standard, ISO/IEC/IEEE 12207, defines the life cycle broadly, from conception through retirement, and this model takes the same broad view. Readers who think in terms of a PDLC will see Intent and Specification as product work, and the SDLC in their sense as starting with the implementation plan. Product management writers such as Teresa Torres call that later part delivery, as opposed to discovery.

PART TWO

The Model

Eight stages, two loops, and what verification costs.

01

The stages

The eight stages of the model, and human review.

Figure 2 shows the whole lifecycle in eight stages. A stage ends by committing one artifact, and that artifact starts the next stage.

Every stage has the same anatomy. Something produces the artifact, something checks it, someone or something accepts it, and the artifact is committed. When a check fails, the work goes back. In most stages today, an agent writes or helps write, an agent reviews, and a person reviews. In the stages that deal with code, much of the checking is verification.

This paper distinguishes two kinds of checks, verification and review, by what decides the result. In verification, a fixed rule decides it: the code compiles or it does not, and a test passes or it fails. Verification covers every check of that kind, such as the build, the tests, and static analysis. An agent can start a verification run, but it does not decide the result. In a review, a person or an agent judges the result. Some checks that look like verification are reviews: for example, when the agent runs the application and looks at how it behaves, or when a second agent tries to prove the first agent's result wrong. The word "verification" is sometimes used more broadly. Anthropic, for example, calls that second agent a verification subagent. This paper keeps the two apart, because a judgment can come out differently on the same change, while a rule applies the same logic on every run. A rule's results can still differ between runs on the same change, when the code under test, the test itself, or the environment is flaky.

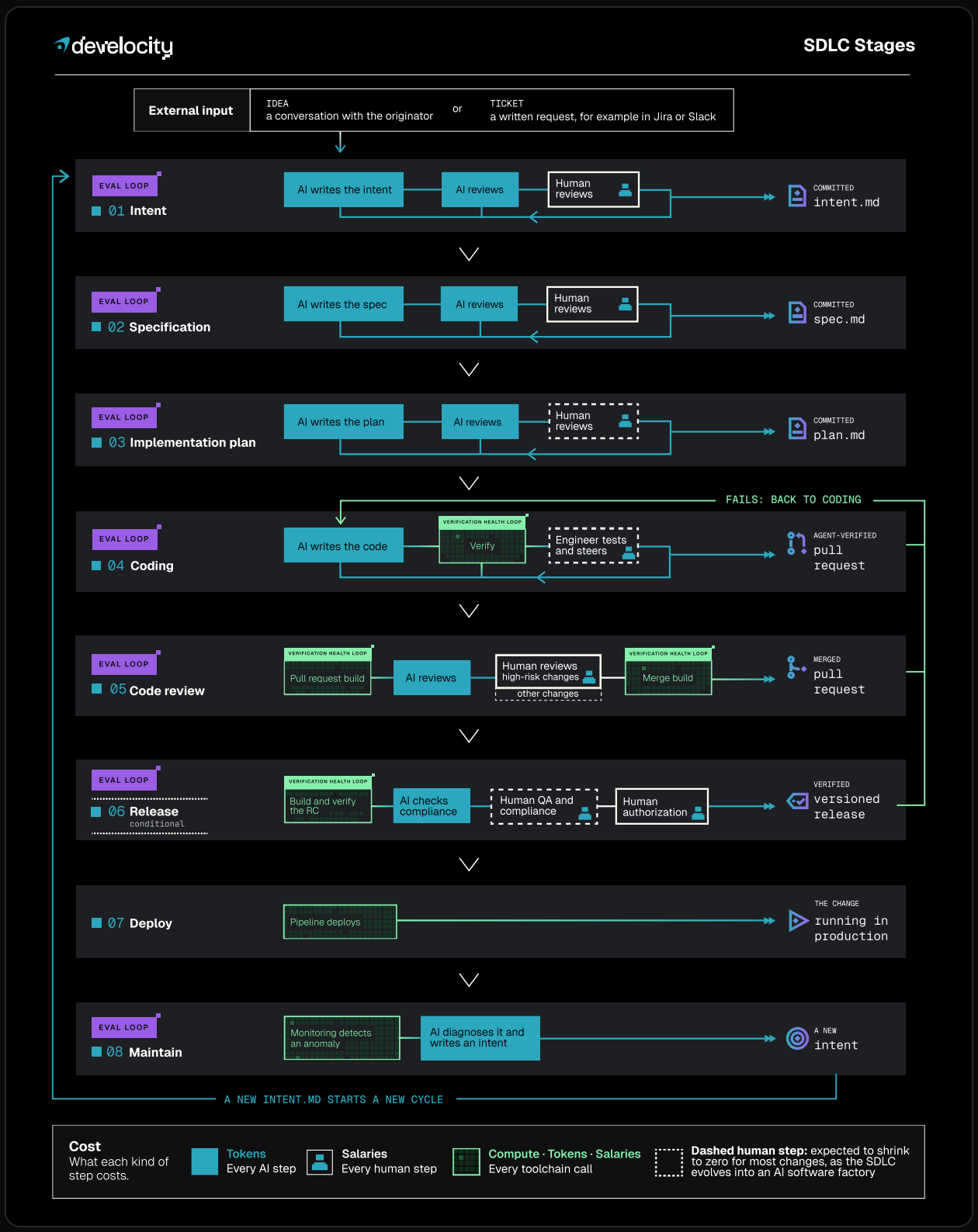


Figure 2. The stages of the AI-native SDLC

STAGE	WHAT HAPPENS
Intent	A request is shaped into what is wanted and why, and someone accepts it as worth doing
Specification	The product side decides the requirements and what counts as correct
Implementation plan	Engineering decides how the code will change: which files, in which order, proven by which tests
Coding	The agent writes the code and verifies it in a loop until the checks pass
Code review	The pull request build, reviewing agents, a person, and the merge build check the change before it lands
Release	Where releases are cut at chosen points, a release candidate is usually built again, verified further, and authorized
Deploy	The pipeline deploys the candidate, with a rollback path
Maintain	Monitoring detects an anomaly in production. An agent diagnoses it and writes the diagnosis and the proposed fix as a new intent

Depending on the type of change, some of the first three stages may be skipped. A change that needs no product decision, as with most bug fixes, skips Specification. Its intent is the bug report. Simple changes may go straight to Coding. More and more other changes skip the Implementation plan too, because the planning increasingly happens within the coding session: the agent plans as it works, or the engineer plans with it, for example in plan mode, and no plan is committed. That planning is part of Coding. For risky or complex changes, an implementation plan that someone accepts before the code is written still pays off: a wrong approach costs far less to correct in a plan than in code. A plan also lets a team use its strongest model for the planning and hand the coding to a cheaper one.

This paper says little about Intent, Specification, and Implementation plan. Today, none of the three stages is where delivery is limited, although the quality of what they produce shapes how much work Coding and Code review need. What feeds Intent is a different matter: everything an organization does to decide what is worth building, which product people call discovery. That is creative work, and it is where a product's unique value comes from.

Human review

Human review is one of the big bottlenecks of today's SDLC. In an AI-native SDLC, I expect the human review steps, drawn as dashed lines in Figure 2, to shrink gradually as the SDLC evolves into an AI software factory, with better harnesses, more mature operations, and better models. The goal of such a factory is to shrink them to zero for most changes. Where an AI

step takes over a person's review, its evals have to be trusted first. Where the risk of a change or a regulation requires a person, a person stays in the loop. As the reviews shrink, people move from being in the loop to being on the loop, as Kief Morris puts it. When an agent's work is not right, a person in the loop fixes it, by editing it or by telling the agent what to correct. A person on the loop changes what produced it: the specifications, checks, and guidance that steer the agent, so that the agent gets it right the next time. People on the loop no longer look at each change. They look at the whole estate, across all projects, to see where the system works and where it needs attention. Accountability stays with named people, and records make it auditable: what was verified, and how, and which person or agent did what. The same records make the remaining human reviews faster and better founded.

02

Verification across the stages

The pyramid, and what each run costs.

Why verification matters more with agents

Verification is critical in any SDLC, and even more in an AI-native one. Its core is specification testing and regression testing. The two overlap: a specification test also becomes a regression test once the change that it was written for is merged. But a regression suite protects much more. It pins down the small choices inside the code, such as thresholds and rounding rules that nobody put into a specification.

Tests matter even more for agents than for people, because a test suite is a codebase's own memory. When something breaks, a test is often added so that the same failure cannot come back unnoticed. The test suite becomes a record of what broke and where the code is fragile. In most enterprise codebases, much of the rest of that knowledge lives only in the heads of experienced engineers, because years of decisions and incidents were never written down. An agent gets some of that knowledge from context files such as `AGENTS.md`, and in an AI-native SDLC, more and more of it will be captured in those files. But the agent may or may not follow its context files. The tests are not advice: if a change breaks one, the change fails. So on every change, an agent that runs them is held to what they record. Where a codebase has few tests, the agent has neither the memory nor the check. But an agent can help close that gap: before it changes a legacy system, it can record in tests what the code does today.

A smarter AI may introduce fewer bugs per change, but agents make far more changes. Say 1% of changes have a bug the tests miss. A team making 20 changes a week hits one every five weeks and treats it as an incident. At 200 changes a week, it hits two a week. If the volume keeps rising, bugs arrive faster than the team can deal with them, and delivery breaks down. It gets worse: changes made at the same time can interact. The more changes are in progress at once, the more likely it is that two of them, each safe on its own, break something together.

The number of missed bugs falls only if their rate per change falls faster than the volume of change rises. So far, the evidence points the other way. In its 2025 report, DORA restates its 2024 finding of an estimated 7.2% increase in software delivery instability for every 25% increase in AI adoption. It also finds that AI adoption now improves throughput but still increases instability.[1]

The checks that matter most are the ones that were not written for the change. An agent's own tests check what it foresaw, and an unintended side effect is exactly what it did not foresee. Agents can also game the checks they are given. So a change has to be judged by checks the agent did not write or rewrite for it, and the agent must not be able to change those checks unnoticed.

Verification also differs from review in kind: it runs the same logic against the code every time, while a review is a judgment. Satya Nadella, CEO of Microsoft, argues for the same principle in AI generally: to "surround non-deterministic models with strong, deterministic system design."[5] As human review shrinks, verification carries more of the trust in the agents' work.

I go much deeper into regression testing, with the evidence and its sources, in Why Regression Tests Will Remain Essential, Even with Smarter AI and Formal Verification, including who writes the tests a change is judged by, and who approves changes to them.

How verification drives cost and throughput

Agents multiply the number of runs. Before they report a task done, they run the build, the tests, and static analysis, as Anthropic recommends, and while they work, they run narrower checks. They also make far more changes than people do. Writing the code now takes a fraction of the time it used to, while the build and the tests have not gotten faster along with it. And there are more tests to run: agents add tests with their changes, and they can write tests for legacy code that never had them, so the test suites grow. So verification accounts for a growing share of the time an agent spends on a change.

A verification run costs tokens, salaries, and compute. The agent spends tokens reading the result and, when the run fails, finding the cause. Today, waiting can also cost tokens. Model providers cache the agent's context between requests, so that each request reads it at a fraction of the price. When a check outlasts the cache, the next request writes the whole context again, at several times the price of reading it. Harnesses can choose a longer cache lifetime, at a higher price for every write.

Salaries and compute are simpler. A failure the agent cannot resolve pulls a person back in, and the build and the tests use compute.

False failures are pure waste: a flaky test or a broken environment fails a correct change. Such a failure costs something at every point where verification runs. Take a flaky test. In the agent's loop, it costs iterations. If the agent takes the failure at face value, it changes code that is not broken. At the pull request, the flaky test costs a rerun, or a needless trip back to Coding, where the agent has to rebuild its context. In the merge queue, it throws out a correct change and holds up the changes behind it. And whenever nobody can tell whether a failure is real, a person has to find out.

Gene Kim, co-author of *Accelerate*, writes: "we need fast feedback loops—faster than ever—to match AI-accelerated code generation."[6] Long feedback loops limit agentic throughput in two ways.

First, an engineer can fill a wait only by steering another session, and parallel work has limits. An engineer can steer only a few sessions at once; Anthropic's tips for Claude Code suggest three to five. The work itself parallelizes only so far: tasks that depend on each other cannot run in parallel. And on a developer's machine, builds compete for memory: a very large build can use almost all of it, so several cannot run side by side. Remote environments avoid that. When the waits are long, the engineer cannot fill them with more sessions. The engineer is paid to wait, and the wait, not the engineer's ideas, sets the pace.

Second, a change that waits can fall behind. It waits for the agent's checks, the pull request build, review, and the merge. Meanwhile, the main branch may move, and then the agent has to reconcile its change with code it was not written against and verify it again. While the verification runs, the main branch may move once more. The slower the feedback loop, the longer a change waits, and the more often and the further it can fall behind. That problem is older than agents, but agents make it worse: a busy fleet moves the main branch more often. Slow feedback and a busy fleet compound, and the more changes fall behind, the fewer agents can usefully work in parallel.

An AI software factory has no engineer waiting for its agents, so the first limit falls away. The second limit tightens: the factory's larger fleet moves the main branch more often, and its agents compete for the same machines and CI capacity when they verify. Where the waits are long compared with writing the code, they keep the SDLC delivery-limited, with or without people steering the agents. I make these arguments in more depth in Your Toolchain Is Production Now and Your Developers Are Still Paying for Your Toolchain.

Why verification is not a stage

Verification is not a stage of its own. It is part of the stages: it runs at several points inside them. A separate, manual test phase after coding is exactly what continuous delivery replaced. DORA's guidance says: "Perform all types of testing continuously throughout the software delivery lifecycle."

Verification also does not fit how this model defines a stage. A stage ends by committing one artifact, and that artifact starts the next stage. Verification commits no artifact that starts another stage. It produces evidence about the artifact of the stage it runs in.

The verification pyramid

A verification result holds only for one state of the code in one environment. When the code changes, or the environment changes, the code has to be verified again. The environment counts because it is part of the test fixture: the operating system, the toolchain, the dependencies, the credentials, the data. That is why verification runs at three points: during coding, at the pull request, and at the merge. At the pull request, the code may be exactly what the agent verified during coding, but the environment is different. It is a CI runner, a clean environment that is usually closer to production than the agent's workstation or sandbox. At the merge, the code is different, because the change is now combined with everything that landed after the pull request was last pushed. The environment may also be hardened further. Where there is a Release stage, verification runs again there.

So even if the checks were identical at the three points, they would still have to run at each one. They are often not identical, for a practical reason. In an ideal world, an agent would get instant, exhaustive, free feedback, and the same verification would run everywhere. In the real one, speed, frequency, comprehensiveness, and cost compete, and the agent's checks, the pull request build, and the merge build each balance them differently. The result is a verification pyramid, the counterpart of the testing pyramid, and closely related to Humble and Farley's deployment pipeline, which also runs the fast checks first and the slower ones later.

The pyramid follows a single change on its way to production. How many changes go through it at the same time is a question of throughput, discussed above.

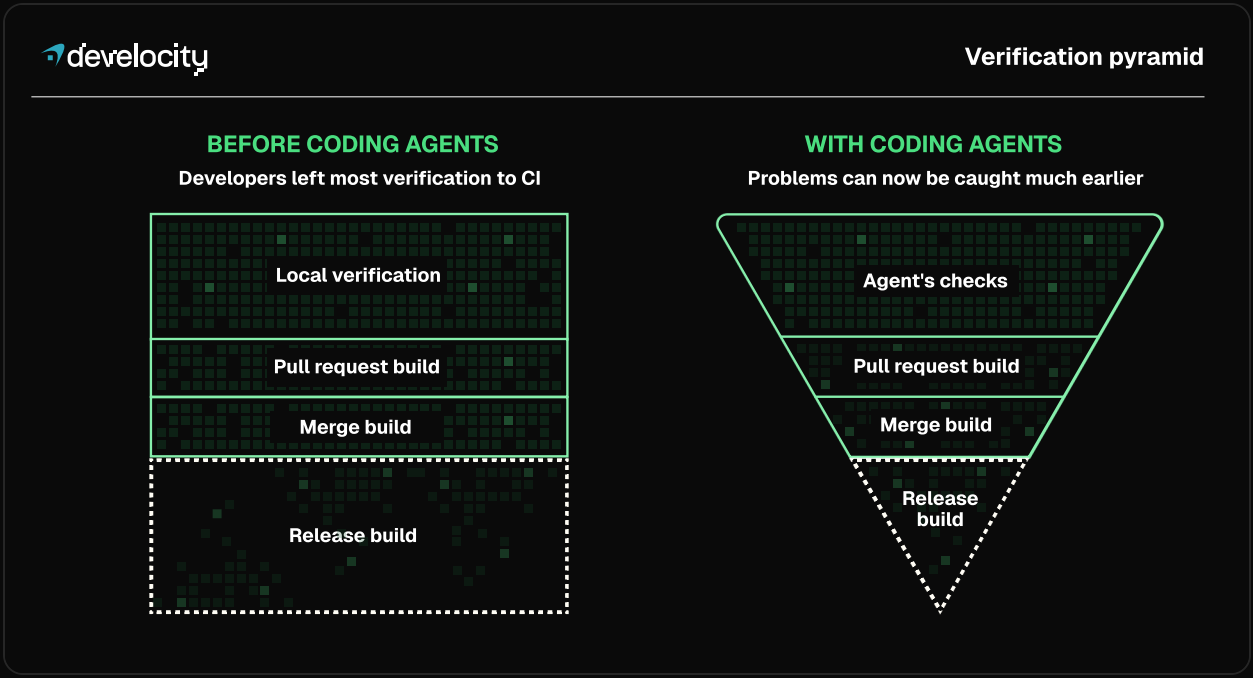


Figure 3. The verification pyramid

During coding, verification runs before anything is pushed, on a workstation or in a cloud sandbox, wherever the agent works. Within its loop, the agent runs narrow checks, such as the tests closest to the change, after every iteration. Before it reports a task done, it runs the complete local verification, so a pull request that implements several tasks gets it several times. Verification during coding runs more often than at any later point, so it should be as fast as possible. It does not have to be exhaustive, but it has to be broad enough to catch most problems.

At the pull request, the pull request build runs in CI on every push and gates the merge. It repeats the agent's checks in the CI environment, and adds the ones too expensive to run during coding, or impossible to run in the agent's environment. The pull request build still has to be fast. The reviewers wait for the pull request build, and every fix for a review finding runs through the build again, so a slow build becomes a bottleneck for review. It also costs more: more CI time, and more reruns when the main branch moves under a change that waits.

At the merge, before the change lands, the merge build tests it against the current main branch and any changes that will land ahead of it. Where a pull request has to be up to date with the main branch before it merges, the pull request build also plays the role of the merge build, as long as the main branch has not moved. Many teams run broader checks here than at the pull request. Here thoroughness wins over speed. A problem this build misses lands on the main branch. Every later change builds on top of it, and the more changes there are, the harder it is to trace the problem back to the change that caused it. Speed is still critical, because every approved change waits for the merge build. And the slower it is, the more

often it has to run again, because the main branch has moved or a change ahead in the queue has failed.

A well-configured pull request workflow has all three. Many organizations verify further after the change has landed, often in more than one step: with checks too slow to gate a merge, or in a staging environment. Those checks cannot gate the merge, because the change has landed. What they verify depends very much on the industry and the organization. These later checks continue the pyramid upward: each further step runs less often, costs more, and finds problems later.

Verification at these three points forms a pyramid in two respects.

The first is how often verification runs at each point. Take one change, from coding to the merge. During coding, the agent verifies its work after every iteration of its task. The pull request build runs once for every push. The merge build runs once for every attempt to merge. With human developers, the frequencies often did not form a pyramid at all. Many developers rarely ran the complete local verification, because it was slow, and left it to CI. Coding agents change that. Anthropic's playbook recommends telling them to run the build, the tests, and lint before reporting any task complete. The base of the pyramid is now wide by default.

The second is where problems are caught. DORA describes Mike Cohn's ideal test automation pyramid as one "where most of the errors are caught using unit testing," and gives the reason: "You should find errors with the fastest category of test." The same holds for verification. Most problems should be caught during coding, fewer at the pull request, and very few at the merge. Frequent verification during coding makes this possible but does not guarantee it: the agent's verification also has to be broad enough to catch most of what the pull request and the merge would otherwise catch.

Where a problem is caught decides what it costs. A problem caught during coding costs the agent one more attempt. The same problem caught at the pull request costs more: a CI run, the trip back to Coding, and the time the agent needs to rebuild the context it had when it made the change. Any review already spent on the change is wasted too, in tokens and in a person's time. Caught at the merge, the problem also blocks every change that is waiting to merge. When one of the three points misses a problem, the cost of that problem does not go away. It moves to a later point, where it is higher.

Small batches in an AI-native SDLC

Working in small batches is one of the principles of continuous delivery. DORA found that small batches amplify AI's positive effect on what it calls "product performance," and its advice to AI-assisted teams is to reduce the size of work items.[7]

Agents change two things: a large change is now cheap to write, and an agent can review it quickly. But a large change is still more expensive to verify. When a test fails, the cause is harder to find, so the agent needs more iterations to fix it, and a large change is more likely to fail and go back to Coding. A large change also still costs more in human review, and it is harder to recover from when it fails in production. In an AI software factory with little or no human review, small batches would matter less, because the cost of human review would shrink. For the other costs, we are only beginning to understand how much the size of a change matters when an agent does the work.

Verification pushes toward small batches only if the tests catch something. With weak tests, the agent's loop passes a large change as easily as a small one, and the cost arrives later.

In this model, a pull request implements a plan, or part of one. So small batches have to be planned. Where the planning is part of Coding, the agent's context files have to ask for small batches.

NOTES

1 DORA, *State of AI-assisted Software Development 2025*, p. 35, restating the 2024 report: "an estimated 1.5% reduction in software delivery throughput and an estimated 7.2% increase in software delivery instability for every 25% increase in AI adoption." The finding that AI adoption now improves throughput but still increases instability is on p. 4. DORA's 2026 report, *The ROI of AI-assisted Software Development*, restates it (pp. 19 and 25).

5 Satya Nadella, Models as Insider Risks in the Super Intelligence Era, essay on X, October 2026.

6 Gene Kim's foreword to DORA's *State of AI-assisted Software Development 2025*, p. 9.

7 DORA, *State of AI-assisted Software Development 2025*, pp. 58–64. Working in small batches is one of the seven capabilities in its AI Capabilities Model. The survey found that small batches amplify AI's positive effect on product performance and turn AI's neutral effect on friction into a reduction in friction. The advice card is titled "Reduce the size of work items." DORA measures batch size by the lines of code in a change, the number of changes in a release, and how long the work assigned in a single task takes.

03

Stage by stage

Where verification and review happen.

Coding

Coding in an AI-native SDLC is more than writing code. A language model that generates code in a single pass produces unverified code, just as a person typing it does. Coding is the model working in a loop with its harness: it writes code, runs the build and the tests, and reads the results. Anthropic's playbook says: "Claude iterates until the check passes, so what reaches the engineer has already passed it." Verification is not a step after coding. It is part of what coding means.

Running the build and the tests before a change leaves the developer's hands has been part of continuous integration since Martin Fowler and Matthew Foemmel described the practice in 2000. Many organizations never got there, and shifting verification left stayed a cultural struggle. With coding agents, a single instruction makes this the default but does not enforce it.[8] I think this default is a major step toward continuous delivery.

Figure 4 shows a coding loop built from Anthropic's recommendations for Claude Code. The agent works one task at a time through a plan: the implementation plan, or, where that stage was skipped, a plan made within the session. It does a task in iterations. In each iteration, it edits one or more files, and each edit is formatted and checked for compile and lint errors in that file. At the end of the iteration, the agent runs the tests closest to the change. If they fail, the next iteration fixes the problem. If they pass, the next iteration continues with the task. How many edits an iteration makes, and which tests it runs, is up to the agent. Before the agent reports the task done, it runs the complete local verification: everything the team has designated to run before a change leaves the developer's machine, such as the build, the tests, and static analysis. The closer the agent's environment is to CI, the more the agent's passing run predicts the pull request build.

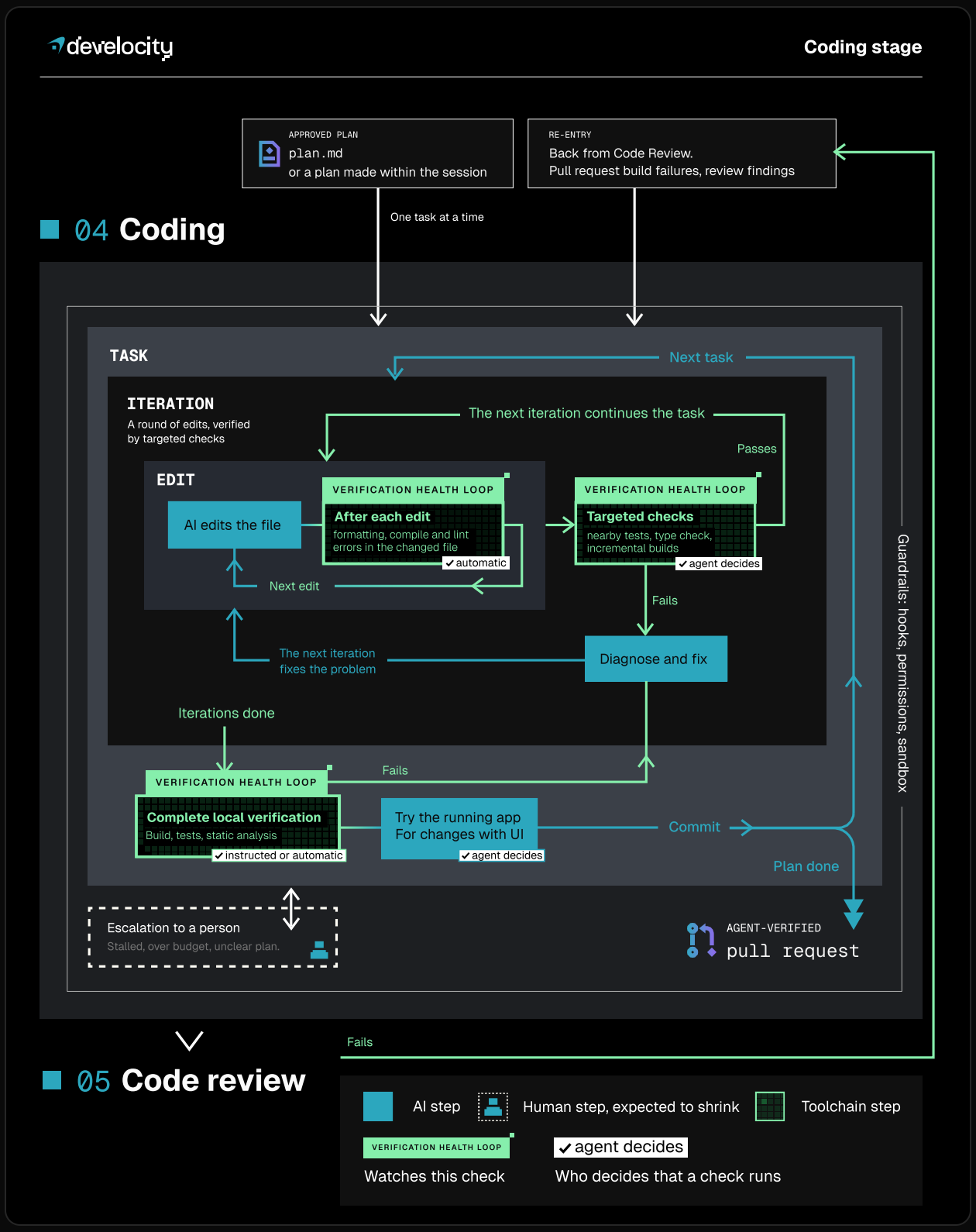


Figure 4. Inside Coding

People work the same way. The IDE checks the code as they type, they run some tests while they work, and, ideally, they run the complete local verification before a change leaves their machine.

Every check ends with the agent reading the result, and a success is no exception. The agent scrutinizes the output and decides whether the output shows what it needs, or whether to check further. For example, it needs to see that the tests passed after its last change. That decision is the model's judgment, not a fixed rule. It varies between models and situations, and it is not always right. The toolchain can help by making the output say what the agent needs to know, in as few tokens as possible: what ran, what was skipped and why, what passed only on a retry. The clearer and shorter the output, the better the judgment, and the less it costs.

When a tool reports a failure, the agent gathers context to find the root cause. The more it has to guess about the cause, the more it has to gather, and the more iterations it may need to fix the problem. Flaky tests are the worst case. Unless the agent knows that a test is flaky, it cannot tell whether its own change caused the failure. It may chase a failure it did not cause, or dismiss a real one as flaky. One failure does not show whether a test is flaky. That takes the results of multiple runs. The verification health loop, described later, observes those results and gives the agent what it finds as context.

Three levers make the coding loop faster and cheaper. The first is the quality of the context, and there are two kinds. The context the agent works from (the intent, specification, and plan, and what it knows about the codebase) drives how much time and how many tokens the agent needs to write the right code. The context that comes with each result drives how much time and how many tokens the agent needs to understand the result and to find the cause of a failure. The second is the speed of verification: fast builds and tests that run only what the change affects shorten every wait. An agent that runs verification 30 times and waits two minutes each time spends an hour waiting. At 20 seconds per run, it waits 10 minutes. The third is the reliability of the result: a failure the change did not cause wastes time and tokens.

Each lever has a measure that shows where things stand, and the first has one for each kind of context. For the context the agent works from, it is the time and tokens the agent needs to write the right code. For the context that comes with each result, it is the time and tokens the agent spends understanding results and finding the causes of failures. For the speed of verification, it is the wait per verification run. For the reliability of the result, it is the time and tokens wasted on failures the change did not cause.

Code review

Code review is where the organization checks what its agents produced, before that work merges. It checks with the pull request build, with reviewing agents and the policy they apply, and with a person, who judges whether the change matches the intent, and how risky it is. Its last check is the merge build. Its input is a pull request, opened or updated. Its output is a merged pull request: the change is approved, it has passed the merge build, and it sits on the main branch. Code review is a stage of its own because the builder and the judge should be separate.

Human review is a major bottleneck here, because the amount of code to review grows with what the agents produce. Which changes need human review, and how to review them well, is one of the most important questions in an AI-native SDLC, and one of the most complex. It is a topic of its own, and this paper covers it mainly where it meets verification.

The order of the checks in Code review matters. The pull request build comes first, for three reasons. First, the agent's own verification is not enough. Which checks it runs is up to the agent: it can skip part of the verification, or report a task as done although a check failed. Hooks can guard against that, but not in every situation. The pull request build does not rely on the verification the agent ran. It runs the verification again, in CI. Second, the pull request build usually runs more extensive checks than the agent runs during coding. Third, review is costly. Human review costs a person's time, and agentic review costs tokens and time. None of that should be spent on a change that cannot be merged.

An agent can make the verification pass by weakening a test. Most weakening shows only in the diff, such as a loosened assertion or a changed expected value. That is why a change to an existing test needs approval from someone other than the agent, who checks that the intent calls for it. A skipped, disabled, or deleted test is easier to catch: the pull request build can flag it.

For agentic review, the approaches vary. Many review tools start when the pull request opens,[9] so a team that wants the build first has to set up that order itself. Some teams make the review wait for the build.[10] Others choose a compromise: they wait for the checks, but only up to a time limit, and then review without the build's results.[11] The conflict is between time and tokens. A review that starts at once comes early and may be wasted. A review that waits for a slow build comes late. The faster the pull request build, the smaller the conflict. The human reviewer should come last in any case, when the build results and the agentic reviews are in.[12]

The build also produces more than a verdict. It reports what it does not enforce: warnings, skipped tests, tests that passed only on a retry. One review pass should cover those, and it cannot start before the build has finished.

While a pull request is in review, other pull requests merge, and the main branch moves. Two changes can merge without a conflict and still break the build or the tests together. So an approved pull request has to be verified again, on top of the current main branch. That is the job of the merge build. A merge queue runs it on the approved changes before they land.[13] Teams with fewer pull requests can instead require a pull request to be up to date with the main branch before it merges. Then the pull request build also plays the role of the merge build. If the main branch has not moved since the pull request was built and reviewed, nothing has to run again. If it has moved, the pull request has to be brought up to date and built again, and that build is the merge build. The more changes land, the more often that happens, which is why teams with many pull requests use a merge queue.[14]

Every failed build and every review finding that needs a change sends the work back to Coding. Every round trip between the two stages costs tokens, salaries, and compute, and it takes time. Two things decide what the round trips cost: how effectively the two stages verify, and how effective agentic review and human review are. Whatever else an organization gets right, an AI-native SDLC does not pay off without effective verification and effective review.

Release

In the deployment pipeline of continuous delivery, as DORA puts it, “every change creates a release candidate.” Humble and Farley’s rule is to build the candidate’s binaries only once, when the change is committed to the main line, so that what goes to production is what was tested.[15] In a pull request workflow, a change reaches the main branch at the merge, so under that rule, the candidate comes from the merge build.

In continuous deployment, there is no Release stage: the binary from the merge build is deployable, and Deploy takes it straight to production. Other organizations cut releases from the main branch at chosen points. A candidate then often carries many merged changes, and because any commit can be chosen, the main branch should never be broken. Such a candidate goes through a stage of its own, Release,[16] where it is usually built again, then verified further and authorized before it goes to production. Figure 2 marks Release as conditional: the stage exists only where an organization cuts releases at chosen points and checks them further. Most regulated organizations do, and so do many large organizations without such regulation. Where the Release stage exists, every change passes through it.

Building the candidate again departs from Humble and Farley’s rule, and the departure has a price: the builds in Code review were evidence about the code, not about the candidate, so the release build repeats their checks on the new binary. From the release build on, the same binary goes through every check and to production.

In Release, the candidate goes through the checks that cannot gate every merge: long-running test suites, security and compliance checks, and a deployment to an environment that is close enough to production to meet compliance requirements. It gets a version and a record of which changes it contains, how it was built, what was verified, and who approved it. In this model, rule-based checks verify compliance wherever a rule can decide, an agent reviews the questions no rule can decide, and people do QA, review compliance, and authorize the release. I expect the human QA and compliance reviews to shrink as automated acceptance and compliance checks mature. The authorization stays where regulation requires a named person. What is verified here differs more between organizations than anything else in the pipeline.

In many organizations today, Release is a major bottleneck. Its human QA and compliance reviews take time, and its verification is extensive and slow. Agents make it worse. Each candidate now carries more changes, so there is more to test, review, and document, and a failed check is harder to trace to its cause. AI-generated code also gets more scrutiny. And however fast a change was written and reviewed, it waits for the next release.

I expect releases to get smaller. Part of what a release costs does not depend on how many changes it carries: the long-running checks run in full, and the QA and compliance reviews take place whether the release carries one change or fifty. That fixed part is one reason it pays to batch changes into releases. As compliance and acceptance checks are automated and verification gets faster, the fixed part falls, and releases can carry fewer changes and come more often. Agents make smaller releases more pressing: they fill each release with more changes. A larger release is harder to verify, and when it fails, the cause is harder to find. The number to watch is how many changes a release carries, which DORA uses as one measure of batch size.[7]

Authorization is different: automation can prepare the decision, but a person remains accountable for it. Regulators ask that changes be authorized,[17] and a release is one way to provide that, but not the only one.[18] The more often an organization releases, the more often a person has to authorize. Beyond some frequency, approving each release no longer scales, and a different control is needed: a person approves the rules and checks that decide which changes may go to production. Pre-approved standard changes already work this way.

Deploy

Deploy does what its name says. Its input is a verified binary, together with a record of how it was produced. Where there is a Release stage, that binary is the candidate that Release verified and authorized. Without one, it is the candidate from the merge build. The stage contains the gate before the deployment, the deployment itself, and the rollback path. Its output is the change running in production, which is where Maintain begins.

Deploy is the only stage without an AI step. Anthropic's playbook draws the same line. Its governing principle is that "the agent may act up to the production gate and cannot pass it."

Maintain

Maintain is where the SDLC watches what it has shipped. What it finds becomes a new intent, next to the ideas and tickets that start the SDLC from outside. That closes the loop. Where a bug reached production and a test could have caught it, the intent also calls for a test that catches it, so that the same failure cannot come back unnoticed. I follow Anthropic's playbook here. Three principles define its version of Maintain. First, people do not have to start the work: an alert, a ticket, or a schedule triggers an agent, which diagnoses the problem and writes an intent, and people triage and review the intent. Second, detection stays deterministic: a script, not a language model, decides that something is wrong. Third, the agent acts only through approved routes, such as a pull request into Code review.

Maintain differs from the other stages. It does not take one change from Deploy and turn it into something else. It watches everything that runs in production, and it starts new work. The latest change still gets the closest attention, because it is the most likely source of new instability. Maintain is the last stage because it watches the software at its final destination, production. Its output, a new intent, starts another cycle.

NOTES

8 Anthropic's playbook recommends this instruction. Its example `CLAUDE.md` lists the commands for the build, the tests, and lint, and says: "Run all three before reporting any task complete, and paste the output."

9 Two examples. Sourcery "reviews a pull request as soon as it opens, and again when you push new commits." In Anthropic's Code Review, "each repository has a setting for when reviews run: once when a pull request opens, after every push, or only on request." The documentation of GitHub Copilot code review and of Cursor Bugbot describes the same kind of trigger. None of these pages mentions waiting for the build. The four pages were checked in October 2026.

10 The pydantic-ai project is one example. Its review workflow starts when CI has completed. A comment in the workflow gives the reason: the project had measured "4.1 runs per PR, 96% of cancelled runs emitting nothing," and "there is no point reviewing a diff that does not compile or pass its tests."

11 CodeRabbit's documentation says that it "waits for GitHub Checks to finish and then reads their output." In August 2025 its configuration schema had a setting for this wait: "Default 90 seconds, max 15 minutes (900000ms)." The setting is no longer in the current schema, so I do not know whether a limit applies today.

12 This is an old norm for human review. GitLab's code review guidelines tell the author of a merge request: "Request maintainer reviews only when tests pass." Chromium's contributing guide says: "Reviewers expect to review code that compiles and passes tests."

13 GitHub's documentation: "The merge queue will ensure the pull request's changes pass all required status checks when applied to the latest version of the target branch and any pull requests already in the queue." GitLab calls the same mechanism a merge train.

14 GitHub's documentation on protected branches describes the rule and names its price: "More builds may be required, as you'll need to bring the head branch up to date after other collaborators update the target branch."

15 Humble and Farley, *Continuous Delivery* (2010), chapter 5, "Only Build Your Binaries Once": "you should only build your binaries once, during the commit stage of the build," because "the binaries that get deployed into production should be exactly the same as those that went through the acceptance test process." The chapter is the publisher's free sample.

16 A word of caution about the name "Release". GitHub's description of the DevOps lifecycle has a Release phase in the same place, with packaging, a version number, and final quality checks. Humble and Farley use the word differently. In their pipeline, the release is the last step: it delivers the system to users. In this model, that step belongs to Deploy.

7 DORA, *State of AI-assisted Software Development 2025*, pp. 58–64. Working in small batches is one of the seven capabilities in its AI Capabilities Model. The survey found that small batches amplify AI's positive effect on product performance and turn AI's neutral effect on friction into a reduction in friction. The advice card is titled "Reduce the size of work items." DORA measures batch size by the lines of code in a change, the number of changes in a release, and how long the work assigned in a single task takes.

17 For example, Article 9(4) of the EU's Digital Operational Resilience Act requires, in point (e), "that all changes to ICT systems are recorded, tested, assessed, approved, implemented and verified in a controlled manner," and adds that "the ICT change management process shall be approved by appropriate lines of management." The act, also abbreviated DORA, is unrelated to DORA, the research program cited in this paper.

18 Some banks meet their requirements for authorizing changes without batching them into releases. In 2022, Monzo, a UK bank, deployed to production more than 100 times a day, with "one review from an engineer on the owning team" as the minimum approval for a change.

04

The two loops

Keeping verification healthy and agents tested.

The verification health loop

Anthropic's playbook applies the approach of Maintain to the delivery system as well, for example, by tracking the failure rate of the CI tests and pull request cycle time. In its words, this "shows the harness works for process metrics as well as production ones." That is an important idea: in both uses, the approach observes what has already happened and reacts to drift. But the two uses look after different things: one the software running in production, the other the delivery system. They also look in different places. Maintain observes the software after Deploy, which is why it comes last in the SDLC loop. The verification runs happen earlier, in Coding, Code review, and Release, and a loop that watches them cannot sit after Deploy. In my model, the verification health loop is therefore a loop of its own, outside the SDLC loop. It maintains the toolchain that verifies the software.

The toolchain drifts. Tests become flaky, builds slow down, and new types of failures appear that the change under test did not cause. Each verification run tells the agent whether its change passed. All runs together tell the loop whether verification itself is healthy. Across all repositories, the verification health loop watches five measures: flaky tests, other failures the change under test did not cause, failures caught later than they could have been, the duration of verification runs, and tests taken out of the gate (skipped, disabled, or quarantined), with how long they have been out. Each one points to a fix in the toolchain. Like Maintain, the loop reacts to what it observes. Maintain does not look for features the product lacks, and the loop does not look for behavior that the tests do not verify. Which tests a change needs is decided in the stages: Specification decides what counts as correct, the Implementation plan names the tests that prove it, and Code review judges them. Where a defect reached production and a test could have caught it, the intent that Maintain writes requires a test that catches it. The change fail rate shows whether more defects are reaching production.

The verification health loop itself sits outside the SDLC loop, but its fixes do not. An agent diagnoses the drift and hands Coding a task with the evidence attached, and the fix goes through Code review like any other change. That works because much of the toolchain is code. Which tests run where is defined in the build and the CI configuration, so moving a test earlier, to catch a failure sooner, is a code change. The same goes for changes to the build and test configuration that make them run faster, and for quarantining a flaky test or fixing it. A quarantine takes a test out of the gate, so it needs an owner and an expiry date. Sometimes the cause of the flakiness is in the product, not in the test, and then the fix is a change to the product.

Not every fix is a code change, though. An overloaded build cluster, an unstable network, or a dependency repository that goes down is a problem for the people who operate the toolchain. For those, the loop opens an issue for operations, with the same evidence attached.

The loop's findings also go to the agents, as context for diagnosing failures.

Broader measures, such as the time a pull request takes to merge, are not for this loop but for the people who run the SDLC. They do not point to a fix in the toolchain. The section on SDLC metrics covers them.

The eval loop

The agents in the stages are themselves a system that changes. An agent is a model and its harness. Around it, a team builds what Birgitta Böckeler calls the outer harness: everything that steers the agent and checks its work, such as `AGENTS.md`, skills, subagents, hooks, settings, and MCP servers. Changing the model, the harness, or the outer harness changes how the agent behaves; that is the point of the change. But it can also have effects nobody intended. The eval loop tests the agents that do the work in the stages whenever their model, their harness, or their outer harness changes. Removing an instruction that a new model no longer needs is such a change too.

Changes to the model and the harness mostly come from their vendors. Changes to the outer harness come from the team, and the best ones come from what actually happened in the stages: where engineers had to steer an agent, which review findings keep coming back, which changes made which tests fail, which pull requests were sent back to Coding, and which incidents reached production. Each points to something the agents did not know or did not do. A change to the outer harness that addresses it keeps the agents from doing the same wrong work again, which saves tokens and the time people would otherwise spend catching it. These failures also become eval cases, so the eval loop can check that the change works, the way a regression test checks a fix in the code.

Each stage with AI has its own evals, because each stage produces a different kind of output, and each kind needs its own way of judging it.

For Coding, an eval gives the agent a real task from the past, for example, the task in which a bug was once introduced. It checks the result the way the pipeline would: the code builds, the tests pass, nothing else changed, and the bug is not there. No team can let the agents redo its whole codebase every time their configuration changes. So the evals are a sample, and their value depends on how representative the sample is. The best cases come from real failures: a bug an agent introduced, a change that Code review sent back, or an incident in production. The records of failed pull request builds are a good place to look, because they show which change failed and why.

For Code review and every other stage with a review step, an eval gives the reviewing agent an artifact with a defect planted on purpose, and checks that the agent finds it. It also gives the agent artifacts without defects, and checks that the agent reports no defect.

The eval loop depends on verification as much as Coding does. An eval for the Coding stage runs the agent on a coding task and checks its result with the build and the tests. While the agent works on the task, it runs the build and the tests in its own loop, as it does in Coding. The results of evals already vary from run to run, because the model's output varies. A flaky test adds variation that has nothing to do with the agent, and it can hide the effect the eval is meant to find. A slow build makes every eval run more expensive, and the more an eval run costs, the harder it is to run on every change to the outer harness. So the verification health loop serves the eval loop too.

The verification health loop and the eval loop both look after the system that makes the software, but they differ in what triggers them and in what they read. The verification health loop observes all verification runs and reacts when results leave their accepted range. It sees drift only once the drift has started, so it cannot stop the first failures. But the earlier it catches the drift, the smaller the problem stays. The eval loop runs tests, and any change to the model, the harness, or the outer harness triggers it. Where the organization controls such a change, the loop is a gate: the change is reviewed before it takes effect, and the evals give the review its evidence.

The evals limit how far the AI software factory can go. A human review step can shrink only as far as the evals of the AI step that takes it over are trusted.

05

SDLC metrics

Where the SDLC delivers and struggles.

SDLC metrics are for the people who run the SDLC, the people Kief Morris places on the loop. They build and manage the system that does the work, rather than inspecting what it produces. They use SDLC metrics for three things: to judge whether the SDLC delivers, to see where it struggles, and to check whether a change they made to it helped. Unlike the findings of the verification health loop and the eval loop, SDLC metrics do not trigger a fix. They lead to decisions about the system: review capacity, the risk policy, the size of plans, the investment in verification.

SDLC metrics come at two levels of detail. The first is delivery outcomes, and the established measures are DORA's. Change lead time, deployment frequency, and failed deployment recovery time measure throughput. Change fail rate and deployment rework rate measure instability. They show whether the SDLC delivers, but not where it struggles.

The second level shows what happens on the way to production: how often the pull request build passes on the first try, how often a change goes back from Code review to Coding, how long a pull request takes to merge, how long a change waits between the merge and production, how much of an engineer's day goes to waiting rather than steering, and the cost per change in tokens, salaries, and compute. These measures point to an area, such as a stage or a handover, but not to a fix. A longer time to merge can mean that reviews take too long, that the pull request build is slow, or that changes go back to Coding too often. It is the job of the people on the loop to find out which. None of these measures is specific to agents, but agents make them matter more: they move the bottleneck to verification and review, and they add tokens to the cost.

SDLC metrics also show some of the problems the verification health loop and the eval loop look for, but later. Poor verification health eventually shows up in the time to merge and the change fail rate, mixed with every other cause. And SDLC metrics matter more as the SDLC

moves toward an AI software factory. As the human review steps shrink, people move from being in the loop to being on the loop, and SDLC metrics become their main view of the system.

PART THREE

The Destination

Where the bottleneck moves when delivery keeps pace.

06

From delivery-limited to idea-limited

Ship as fast as you can decide.

Today, most SDLCs are delivery-limited: the bottleneck is verifying and reviewing what the agents build, and, where there is a Release stage, getting changes through it. So Coding, Code review, and Release largely decide how fast an organization can ship, and much of the SDLC's cost arises there. That is where agents and engineers do most of the work on a change, and where verification and review decide how many rounds that work takes. Every round costs tokens, salaries, and compute.

An AI-native SDLC is meant to move the bottleneck: its stages verify and review the agents' work effectively, and two loops beside them keep the toolchain and the agents in shape. Getting there takes decisions about review capacity, the risk policy, the size of plans, and the investment in verification. As human review shrinks, as far as the evals allow, verification carries more of the trust in the agents' work. Every change has to pass verification, so its speed and reliability limit how fast changes get through.

When Coding, Code review, and Release run smoothly, the bottleneck moves to where Andrew Ng sees it: deciding what to build, the work that feeds Intent. An organization still pays for every change in tokens, salaries, and compute, so efficient verification and review lower that cost even when they no longer limit the speed. The SDLC is then idea-limited: an organization can ship as fast as it can decide what is worth building.

Acknowledgments

Many thanks to Piotr Jagielski, Rahel Lüthy, Jan Mittelman, and Etienne Studer, whose reviews improved this paper in many places; to Harriet Clayton, who designed the diagrams; and to Peter Vo, the paper's editor.

About the author

Hans Dockter is the founder and CEO of Develocity, formerly Gradle, Inc. He created the Gradle Build Tool, one of the world's most widely used build systems. Before that, he led build and continuous delivery projects for major enterprises and held software engineering roles at Volkswagen and several startups. He also created the JBoss-IDE and co-founded a consulting company with Eric Evans focused on Domain-Driven Design. Hans is a Java Champion and holds a master's degree in physics from the University of Göttingen.

ABOUT DEVELOCITY

The model in this paper is independent of any product. Develocity helps teams strengthen the verification it depends on: faster verification runs, fewer false failures, clearer results, faster troubleshooting, and a verification health loop that acts on drift, across Gradle, Maven, Bazel, sbt, npm, and Python builds.

Learn more about Develocity:
develocity.ai

develocity

Develocity®, Gradle®, and Build Scan® are registered trademarks of Gradle, Inc. © 2026 Gradle, Inc.