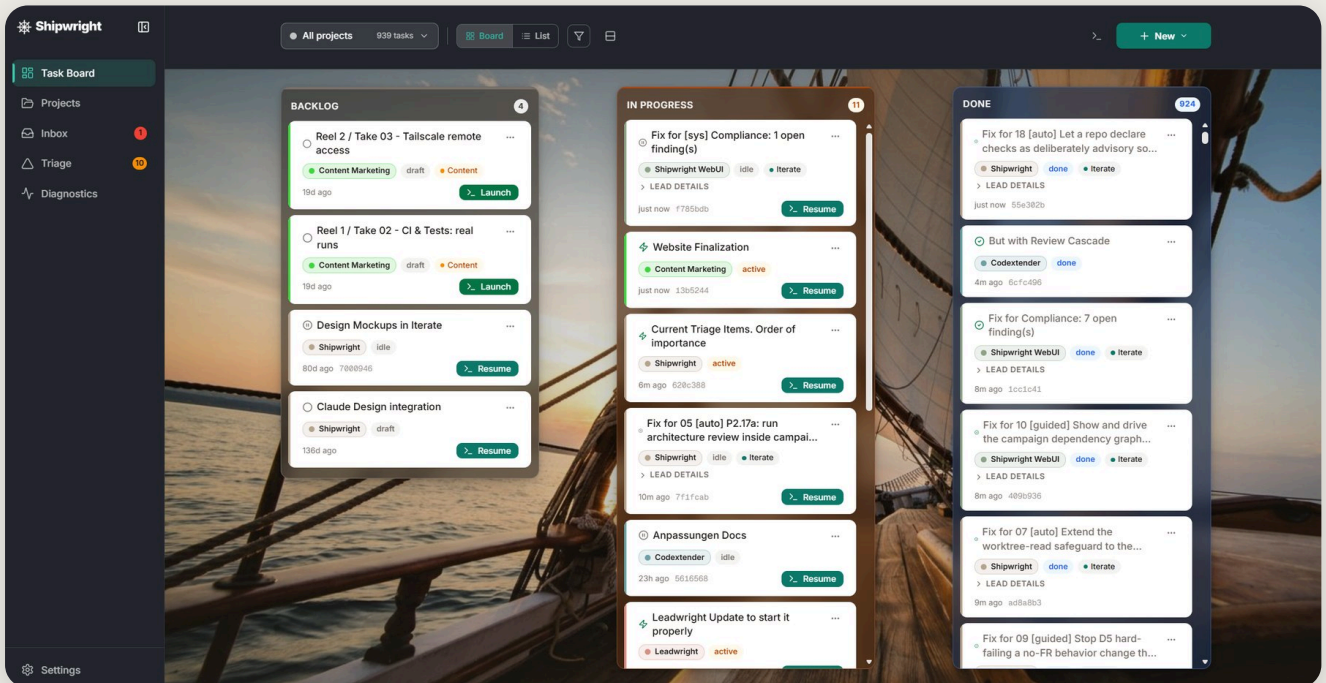


WHITEPAPER / OCTOBER 2026

Driving the AI SDLC with the Shipwright AI Code Factory.



Ship right, **not just fast.**

Contents

	Executive summary	3
	In short, the argument and five takeaways	
	One small change	4
	A small change, who this is for, the model in one picture	
01	AI builds fast, and nothing checks it against what you decided	6
02	The answer turned out to be a harness and a loop	11
03	Layer one: start with what is true	19
04	Layer two: make the rules something the system enforces	22
05	Layer three: put every change through the same loop	26
06	Layer four: make the result readable	31
07	The Command Center: loop engineering you can see	33
08	Your role changes before your tools do	37
09	Where this fits, and where it stops	40
10	A maturity model for control, and ten questions	42
	Closing	45
	Writing code is no longer the hard part	

APPENDIX

A	How the Control Grade is calculated	47
B	The change loop in thirteen steps	47
C	Timeline	48
D	Glossary	49
E	Sources	50

Executive summary

AI writes the code now. For many teams that part is solved, and it got cheap faster than anyone planned. What did not get solved is what comes after: knowing that the software is still true to what you decided, on every change, not just the first one.

This paper makes one argument: control is a process that can be built and measured.

Baseline

WHAT IS TRUE



Harness

WHAT IS ENFORCED



Loop

HOW EACH CHANGE RUNS



Evidence

WHAT CAN BE READ

FIVE THINGS TO TAKE AWAY

01

Speed and control are two axes.

Studies from 2025 and 2026 are mixed on how much faster AI makes teams. They agree on this much: AI-written code arrives with more issues per change, and teams report losing track of what they own.

02

The industry has converged on two ideas.

Harness engineering (the system around the agent) and loop engineering (the system that prompts the agent). The names are new. The practice is what Anthropic's own engineers describe.

03

A baseline is what makes a loop worth running.

A loop without a written record of what is true will iterate quickly on things nobody can check.

04

Automation only pays off with a loop and a harness.

If people check every AI step by hand, each role does more but also spends far more tokens, and the savings never arrive. If nobody checks, the process is worse than before.

05

Your role moves from typing to directing.

Intent, priorities, exceptions. The people stay accountable.

About this paper

Shipwright is an implementation of these four layers. It is open source and runs on Claude Code. Each chapter shows the idea, then how Shipwright applies it, including where it fell short. The paper ends with a maturity ladder and ten questions you can ask of any AI-assisted way of building software, whether it is Shipwright or not.

One small change

Earlier this year I needed a handful of mobile fixes in the Command Center, the web interface for Shipwright. A small change. I typed what I wanted, answered two questions, and went on with my day.

When I looked again, exactly what I wanted had happened, the way I wanted it. A short specification of the change existed. Tests had been written before the code and were green. The documentation was updated. The decision behind one of the fixes was recorded where the next session would find it. Every piece traced back to a requirement. A pull request had been opened, its checks were green, and it was merged.

None of that was luck, and I had not asked for any of it. The harness around the AI enforced each step, and the loop steered the change through them. That is the work I would otherwise have to remember to do by hand, on every change, forever.

That small change is the reason for this paper. It shows what the work looks like when control is built in. It also raises the question every technical leader eventually asks about AI-built software: **who checked it, and against what?**

There is a second question behind it, and it is about money. Automation only pays off when the checking is automated too. If a person controls every AI step, each role does more, but it also costs far more tokens, and the control, even with AI help, stays mostly with the human. If nobody controls anything, which is what often happens when people simply talk to an AI, the process ends up worse than the one it replaced. What you need is a defined approach: what is checked, by what, where, and when, and where a person really looks. That approach is a loop, held up by a harness.

Who this is for

This paper is written for people who want to understand how the software development lifecycle (SDLC) evolves with AI, and who want to see it on a real example, not on slides and paperwork. Some of you already use AI to build and want to regain control of what it builds. The paper shows what a working setup looks like, why it is designed that way, and what to ask for.

If you build prototypes and throwaways, the early levels of the ladder in chapter 10 are the right place to stay, and most of this paper is overkill.

The model in one picture

The answer is organized into four layers. They are a way of ordering ideas that other people named first. Birgitta Böckeler and Mitchell Hashimoto described the harness. Addy Osmani and Boris Cherny described the loop. Chapters 3 to 6 each take one layer.

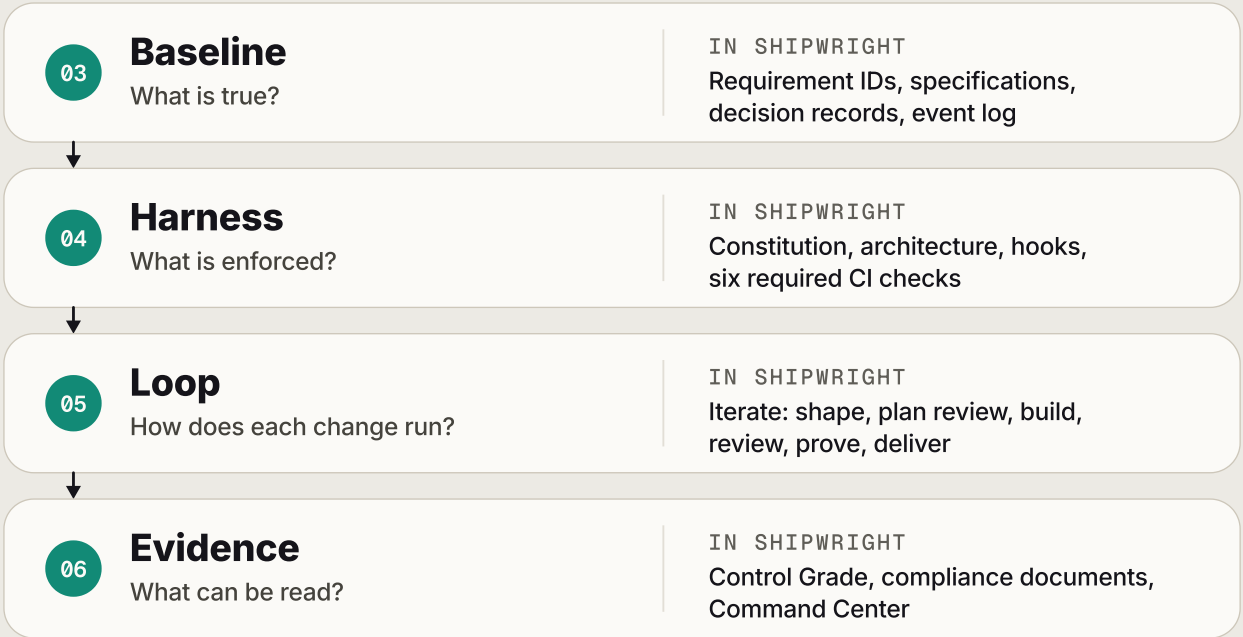


EXHIBIT 01 **Control is four layers, and each one answers a question that can be asked.**
Chapters 3 to 6 take one layer each.

AI builds fast, and nothing checks it against what you decided.

The speed arrived in 2025

On 2 February 2025 Andrej Karpathy wrote about "a new kind of coding I call 'vibe coding,' where you fully give in to the vibes, embrace exponentials, and forget that the code even exists." It spread because it matched what people were already doing. Three weeks later, on 24 February, Anthropic released Claude Code as a limited research preview, alongside Claude 3.7 Sonnet. It reached general availability on 22 May 2025.

The first versions were useful, not yet reliable. Boris Cherny, who started Claude Code at Anthropic, says in public interviews that it began on a small prototyping team and wrote only a fraction of his code at first. The steps that mattered came from the models. By November 2025, Simon Willison reports, the field had crossed a threshold: coding agents went from code that "would mostly work, but you had to pay very close attention to it" to doing what you told them "almost all of the time."

For a leader this was the best and the hardest news at once. A team could build in a week what used to take a quarter. Shortly after, the same team could no longer say, with the same confidence, what it had built.

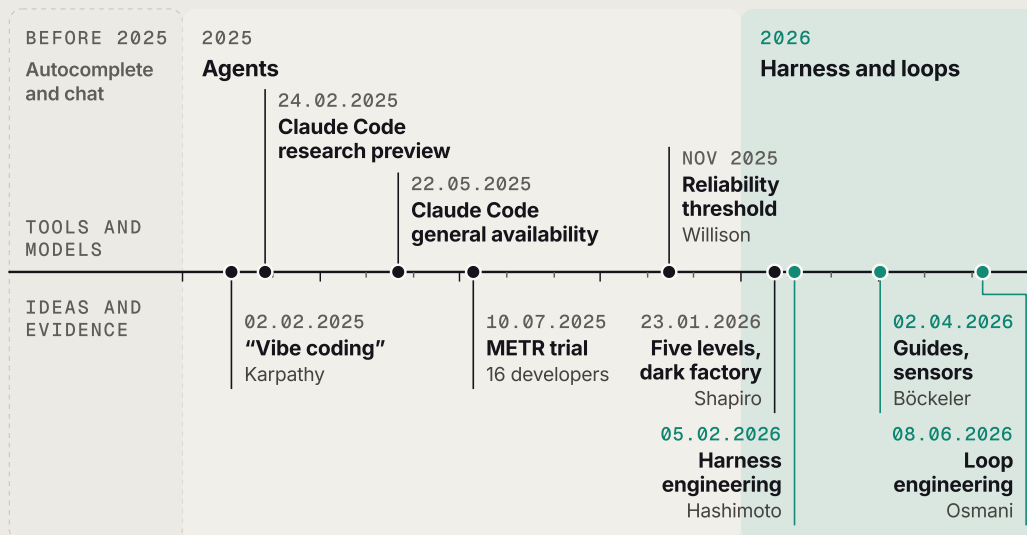


EXHIBIT 02 In sixteen months the field went from vibe coding to harnesses and loops.

Dates are first publication. Sources in Appendix E.

What the numbers say

The evidence on AI coding is mixed, and it deserves care. Three findings stand out. Two of the sources below date from 2025, the others from 2026.

Use is near universal. The May 2026 Google paper *The New SDLC With Vibe Coding* cites survey figures: 85 percent of professional developers regularly use AI coding agents, 51 percent daily, and about 41 percent of new code is AI-generated. The paper takes these numbers from industry surveys, so read them as indications.

Speed is not guaranteed. In a randomized trial published by METR in July 2025, 16 experienced open-source developers worked on 246 real issues. With AI tools they took 19 percent longer. They had expected to be 24 percent faster, and afterwards believed they had been 20 percent faster. In February 2026 METR published a follow-up with 57 developers and more than 800 tasks. It pointed the other way, with an estimated speedup of 18 percent for returning developers and 4 percent for new ones, but METR itself calls this only very weak evidence, because many developers withheld tasks they did not want to do without AI. METR adds that this selection effect probably understates the speedup. Nobody knows the true figure yet, and the people involved could not judge it by feel.

Quality moves the other way. CodeRabbit analyzed 470 pull requests in December 2025. The AI-co-authored ones carried 10.83 issues on average, against 6.45 for human-only ones, about 1.7 times as many,

with security issues up to 2.74 times higher. Veracode's 2026 report, published in July, found that the average security pass rate across models was 56 percent, barely changed from 55 percent in its first report. Roughly 44 percent of the code generation tasks introduced a risky vulnerability, while AI now authors about half of all committed code in adopting organizations. The DORA research program, in its 2026 report, associates AI adoption with higher individual effectiveness and with higher delivery instability.

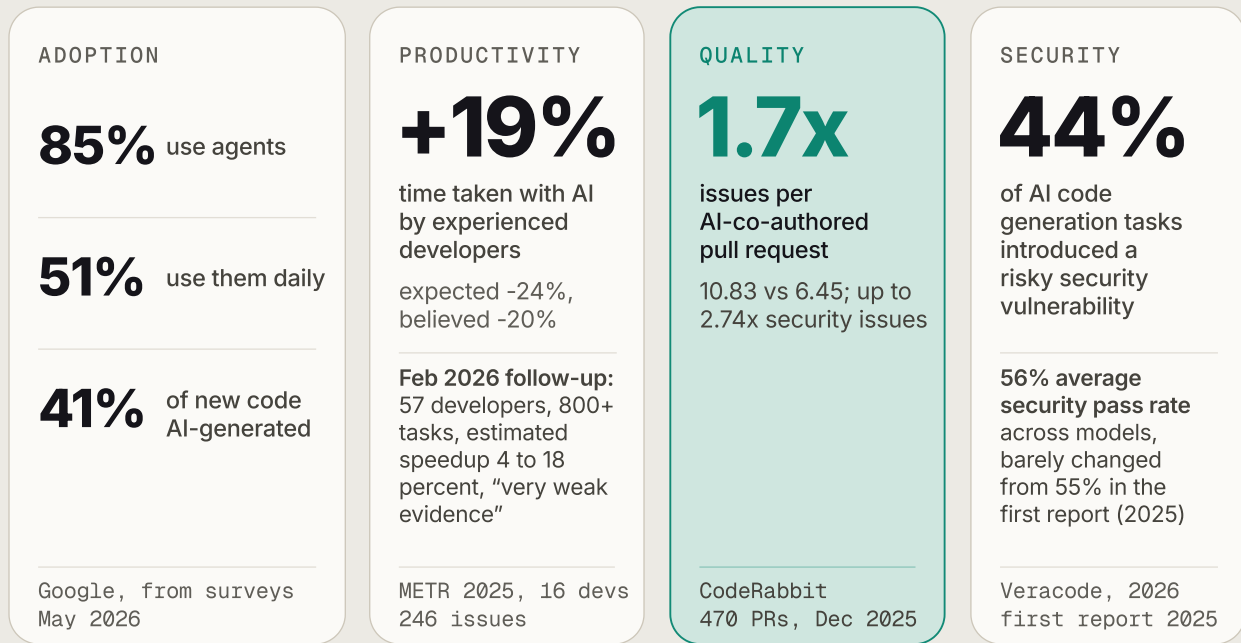


EXHIBIT
03

Use is near universal, speed is uncertain, and security does not improve.

Sources: Google (May 2026), METR (2025, update February 2026), CodeRabbit (December 2025), Veracode (July 2026).

None of this says that AI coding is a mistake. It says that speed and control are separate axes, and AI moved only one of them.

The part under the water

Developers describe the missing control in the same words again and again. One public example is a Hacker News thread titled "After Two Years of Vibecoding, I'm Back to Writing by Hand", in which developers describe losing their grip on code they let an AI write. They are not saying that AI is bad. They are saying that they lose the thread.

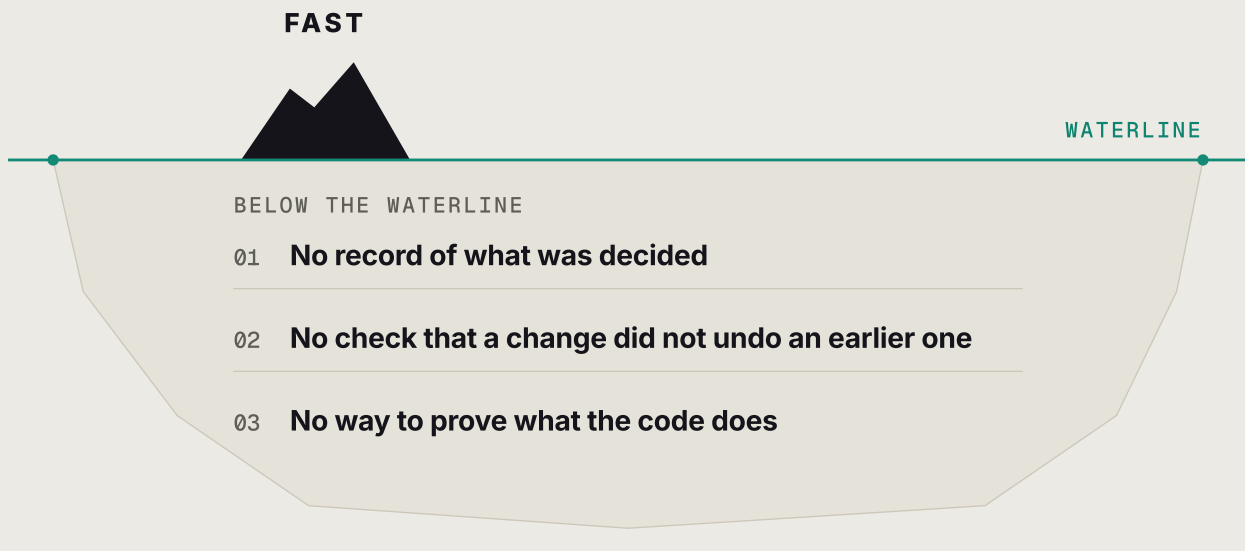


EXHIBIT Under the speed sits what nobody wrote down.
04

Three questions show where control goes missing:

- Could you prove what this software does today?
- Is every behavior covered by a test that runs?
- Did the last change quietly undo something you decided months ago?

If you cannot answer all three without asking the AI, you do not control the software. The AI knows what is in front of it right now. The decisions you made last quarter are only in front of it if somebody put them there, and nothing in a plain chat checks the result against them.

What this means for you

For anyone who leads or owns software, whether in technology, product, or delivery, the risk is accountability. You are answerable for software that nobody can fully explain, built at a speed that makes explaining it harder every week.

When code is cheap, it can be rewritten at will, so the code itself matters less. What has to survive each rewrite is the intent behind it, seen from the business: what the software is for, what it must always do, and what it must never do. If that intent is on record, a rewrite can be checked against it. If it lives in someone's head, every rewrite is a gamble, and the person who remembers is the only control.

The three questions above are a quick test. If any of them cannot be answered without asking the AI, the intent is not on record, and the baseline in chapter 3 is the place to start.

CASE

Why I built Shipwright

About a year and a half before I wrote this, a software architect I value very much recommended that I try Replit. He was amazed at what was possible with AI coding. I did, and I was hooked. Building software had become simple, and it taught me a lot about how a commercial product bridges the gap from a chat to deployed software. What I missed was everything that turns "it works" into "it's shipped right": requirements that say what you are building, traceability that shows you built that and nothing else, tests, decisions on record. I have spent my career on that gap, in environments such as a FINMA-approved digital asset exchange and private banking, where proving that a change is safe is part of the job. AI builds fast, and it skips what makes software safe to keep. I wanted that apparatus in the AI-driven SDLC too, especially since creating those artifacts no longer takes much work with the support of AI.

IN ONE LINE

AI moved speed. Nothing moved control with it. That gap is the subject of this paper.

The answer turned out to be a harness and a loop.

What changed after Claude Code launched

When Claude Code appeared in February 2025, the step that mattered was not a better answer. It was that the model could **take action**. It read files, ran commands, looked at the result, and went round again. A common short definition, from Simon Willison in September 2025, is that an agent runs tools in a loop to achieve a goal.

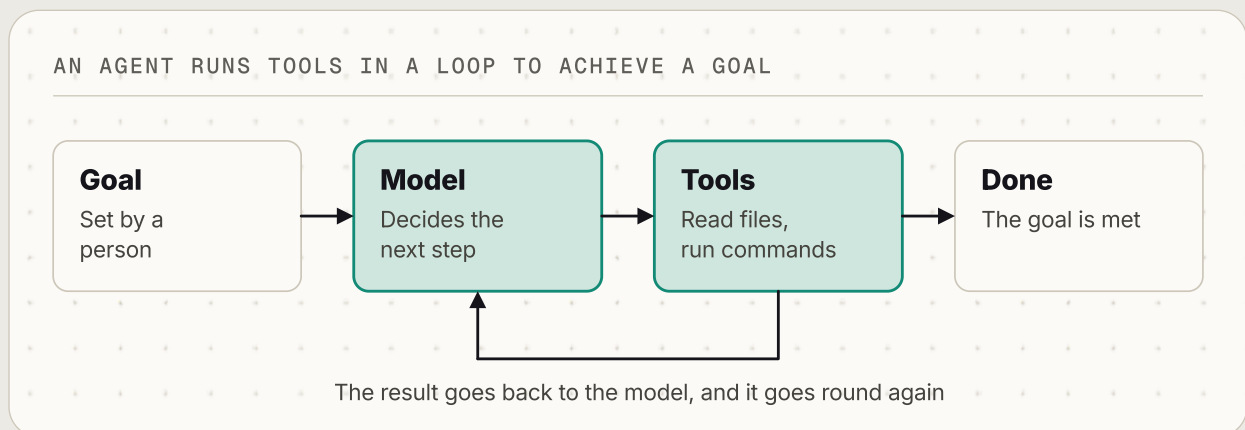


EXHIBIT 05

An agent is a model that uses tools in a loop until the goal is met.

Definition: Simon Willison (September 2025).

Two things followed, and they explain most of what the industry did next.

First, the loop got longer. Each model release let an agent work further before it needed a person, and over 2025 Claude Code gained the pieces that support long runs: hooks, subagents, skills, and plugins. Longer loops are where small mistakes compound. An agent that takes a wrong turn in minute two builds on it until minute forty.

Second, the work moved from a conversation to a system. When an agent works for an hour without you, the prompt you typed matters less than everything else that decides what it does: the rules it reads, the tools it can call, the checks that run after it acts.

The industry found its words in 2026

The first half of 2026 was a naming wave. Several people, working separately, described the same shift. Four terms came out of it: agentic engineering, dark factory, harness engineering, and loop engineering. Two of them, harness and loop, carry the rest of this paper. The other two describe where the person stands and how far the checking can move into the system. Each gets its own short section below.

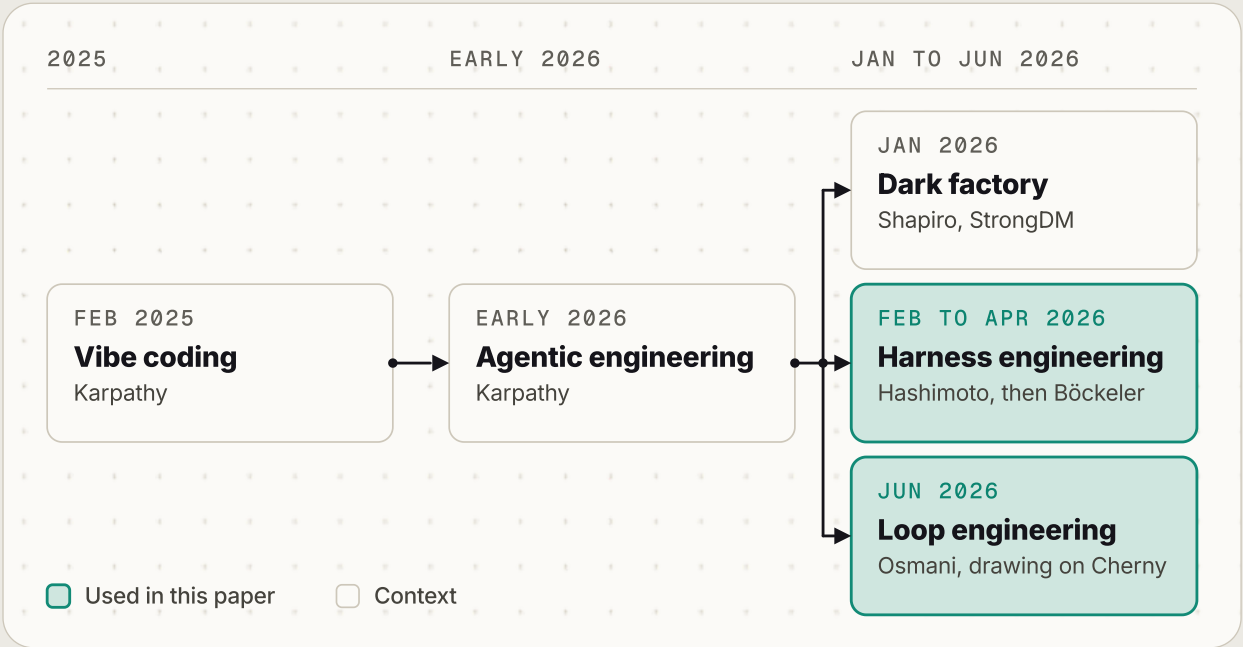


EXHIBIT 06 Five terms in sixteen months, and two of them carry this paper. Harness and loop are used throughout. The others are history.

Harness

A **harness** is everything around the agent that steers it and checks its work: the rules it reads and the checks that run on what it did. Claude Code itself already comes with a harness: its tools and default rules. In Böckeler's formula, a model plus that harness is the agent. This paper uses the word harness for the second one, what a team adds around the agent. On 5 February 2026 Mitchell Hashimoto wrote about a practice he called harness engineering: whenever an agent

makes a mistake, you engineer a solution so that it never makes that mistake again. In April, Birgitta Böckeler gave it a structure on Martin Fowler's site: a harness holds **guides**, which steer before the agent acts, and **sensors**, which check afterwards.

One word carries more weight than the others: **enforced**. A harness is not a set of requests. If the only thing between the agent and a mistake is a sentence in a prompt, there is no harness yet, only hope. The Google paper *The New SDLC With Vibe Coding* (May 2026) says it in its own terms: a harness gives the agent "state, tool execution, feedback loops, and enforceable constraints", and it names hooks as "deterministic code that runs at specific lifecycle points", the place "for things the agent should never forget but often does". Böckeler's sensors are the same idea seen from the other side: they only help when a program runs them, every time. Control over AI depends on deterministic means, meaning code that runs and does not interpret. Chapter 4 shows how.

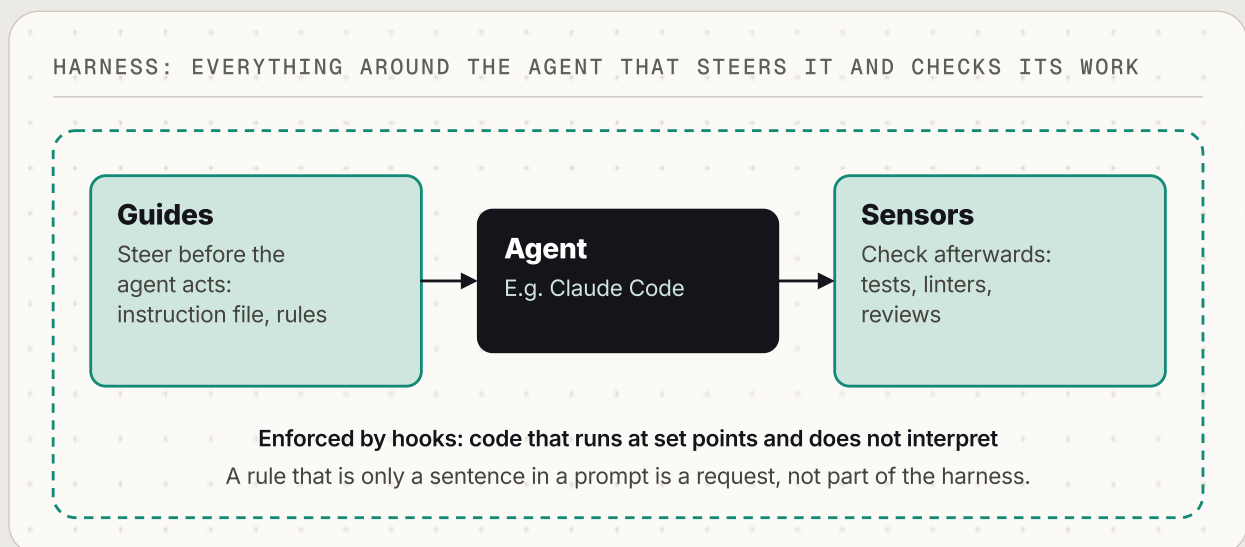


EXHIBIT 07 **A harness is the guides before and the sensors after, and enforcement makes it one.**
Guides and sensors: Birgitta Böckeler (April 2026). The drawing is the author's.

Does the harness matter that much? One data point from the Google paper: on the benchmark Terminal Bench 2.0, one team's agent moved from outside the top thirty to the top five by changing only the harness. The paper's own sentence is that most agent failures, examined honestly, are configuration failures.

Loop

A **loop** is a system that prompts the agent, checks the result, and feeds the outcome back, until a goal is met. In June 2026 Addy Osmani wrote about loops: instead of prompting an agent task by task, you define the intent and design a system that keeps prompting until it is met. He draws on Boris Cherny, who describes running loops instead of prompting.

The loop is where the harness gets used. In each round the sensors check the result and the guides steer the next one. What the person does changes with it: the person defines the intent and reads the outcome, and the system does the prompting. Chapter 5 follows one change through a loop from start to finish.

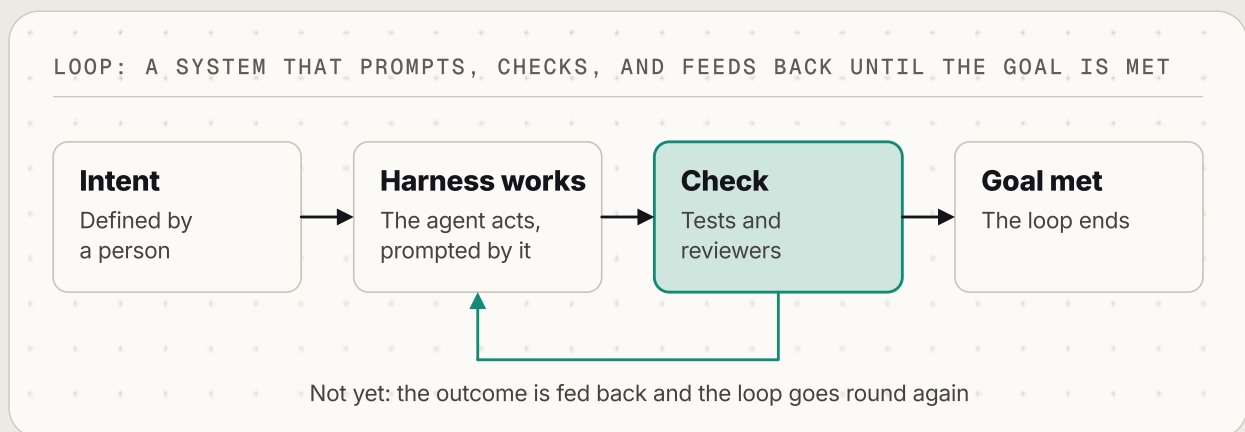


EXHIBIT
08

A loop prompts, checks, and feeds the outcome back until the goal is met.

Loop engineering: Addy Osmani (June 2026). The drawing is the author's.

Agentic engineering

In early 2026 Karpathy, who had coined vibe coding a year before, proposed a second term. Programming through AI agents, he wrote, was "increasingly becoming a default workflow for professionals, except with more oversight and scrutiny."

The two terms mark a difference in where the person stands. In vibe coding the person gives in to the vibes and forgets that the code even exists. In agentic engineering the person sets intent and priorities, agents do the work, and the person applies oversight: reading the evidence and deciding the exceptions. Chapter 8 returns to this role.

Dark factory

Dan Shapiro used the term in January 2026 for the far end of a scale of AI-assisted development. He called it dark because it is a place where "humans are neither needed nor welcome", a black box that turns specs into software. Simon Willison later described one real attempt, a small team at StrongDM working under two rules: code must not be written by humans, and code must not be reviewed by humans.

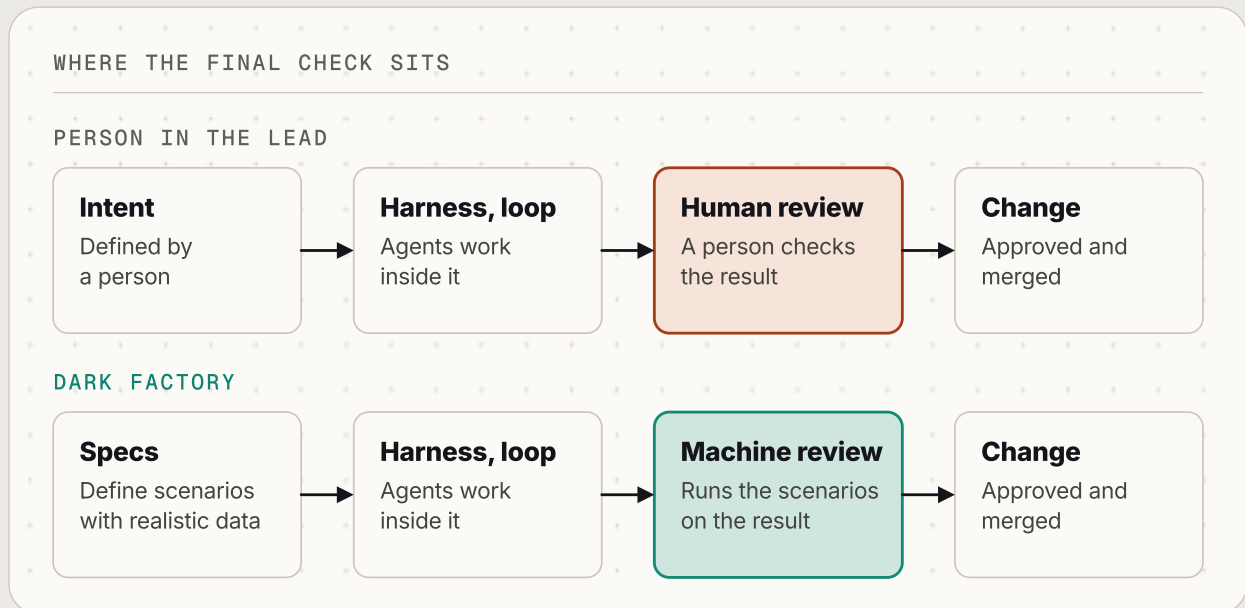


EXHIBIT 09

The question is where the final check sits: with a person or in the system.

Dark factory: Dan Shapiro (January 2026); StrongDM as described by Simon Willison. The drawing is the author's.

What replaces review is checking built into the system. The first is scenarios: end-to-end user stories with realistic data, kept away from the agents that build, the way a holdout set is kept away from a model in training. The second is a measure of satisfaction instead of pass or fail: of all the paths through all the scenarios, what share would probably satisfy the user.

The third is a behavioral clone of the third-party services the software depends on, such as Okta, Jira or Slack, so that thousands of scenarios can run per hour without hitting rate limits.

The other setup keeps people in the lead. A person defines the intent and checks the result, and agents work inside a harness and a loop. The difference is where the final check sits.

The two setups differ in what stands in for the review. With people in the lead, it is intent, a specification, and a human review. In a dark factory, it is intent, a specification, and scenarios with realistic data, and the review is done by machine. That is why a dark factory needs more in place: scenarios that exist as a maintained record, and standards for the third-party services they touch. The checking has moved from a person into the system, which is the question chapters 3 to 6 take up for every setup. Chapter 10 returns to this with Shapiro's five levels as a maturity ladder.

How Anthropic describes its own practice

The people who build the tool say much the same thing. Anthropic's public guidance on Claude Code reads like a short list of controls.

- **Give the AI a check it can run.** In Anthropic's words, that is "the difference between a session you watch and one you walk away from." Without one, "you become the verification loop."
- **Explore first, then plan, then code.**
- **Treat the project's instruction file like code.** Review it, prune it. Rules that must hold belong in hooks, because hooks are deterministic and instructions are advice.
- **Use a reviewer with a fresh context,** so that the agent doing the work is not the one grading it.

Anthropic also says how far this goes. As of May 2026, according to the company, more than 80 percent of the code merged into its own codebase is authored by Claude. Cherny describes a company where designers ship code too.

An experiment from Anthropic shows what holds such a setup together. Sixteen Claude agents, across nearly two thousand sessions, built a compiler of about 100'000 lines that can build Linux 6.9. The lesson Anthropic drew was about tests: the verifier has to be nearly perfect, or the agent will solve the wrong problem. The agents were fast. The tests kept them pointed at the right thing.

How the four terms map to the layers

The four terms ask one question from different sides: where does the checking sit, and what makes it hold? The rest of the paper takes that question apart in four layers. A harness that enforces is the second layer (chapter 4). A loop that every change goes through is the third (chapter 5). The checking needs something to check against, a

baseline that says what is true (chapter 3), and something to show for it, evidence that can be read (chapter 6). Agentic engineering names the role of the person in this setup (chapter 8), and the dark factory marks how far the checking can move into the system (chapter 10).

CASE

The harness I was missing

The gap I had felt in the browser builder turned out to be the harness. The model was good. What was missing around it was enforcement, a written record, checks, and a way to see state. I started to build exactly that, as plugins for Claude Code, first one for each phase of software delivery. Today there are fourteen of them, and one, `shipwright-grade`, measures how much control a repository can prove.

IN ONE LINE

The new vocabulary points at one idea. Build the system around the agent, and put every change through a loop.

THE ARGUMENT OF THIS PAPER

Control is a process that can be built and measured.

Four layers hold it up: a baseline that is checked, a harness that enforces, a loop that every change goes through, and evidence that can be read.

CHAPTERS 3 TO 6

Layer one: start with what is true.

• 1 BASELINE

2 HARNESS

3 LOOP

4 EVIDENCE

Question this layer answers: what is this product supposed to do, and where is that written down?

A record is not enough

Every serious AI setup keeps some written memory of the project: instruction files, specifications, notes from earlier sessions. That is good practice, and Anthropic's own guidance recommends it. A record helps the AI start each session with the right picture.

A record only helps while it stays true. A requirement you settled in spring can be undone in autumn by a change that nobody connected to it. The AI that made the change did not know. The one that reviewed it did not know either. Nothing in the process asked.

This is not a new problem. In the traditional world, specifications and architecture documents drift away from the code within months. Everyone knows it, and most teams live with it: the document is updated after the fact, if at all, because updating it is tedious and nobody gets thanked for it. Auditors and new colleagues then meet a record that describes a product that no longer exists.

AI changes this in two directions at once. Catching up on documentation becomes easier and cheaper than ever, since an agent can read the change and update the record in minutes. But the gap also opens faster. When an agent ships ten changes in a day, a record kept by habit falls behind by the afternoon, and the loss of control grows with every change that nobody traced. Whether the record stays true is no longer a matter of discipline. It has to be part of the process, which is why this layer comes first.

So the first layer is more than documentation. It is a **baseline**: a written statement of what the product is meant to do, linked to the code, the tests, and the decisions, and **checked again when it changes**. The baseline is not kept on the side. It is the source of truth the next session starts from.

What the baseline holds

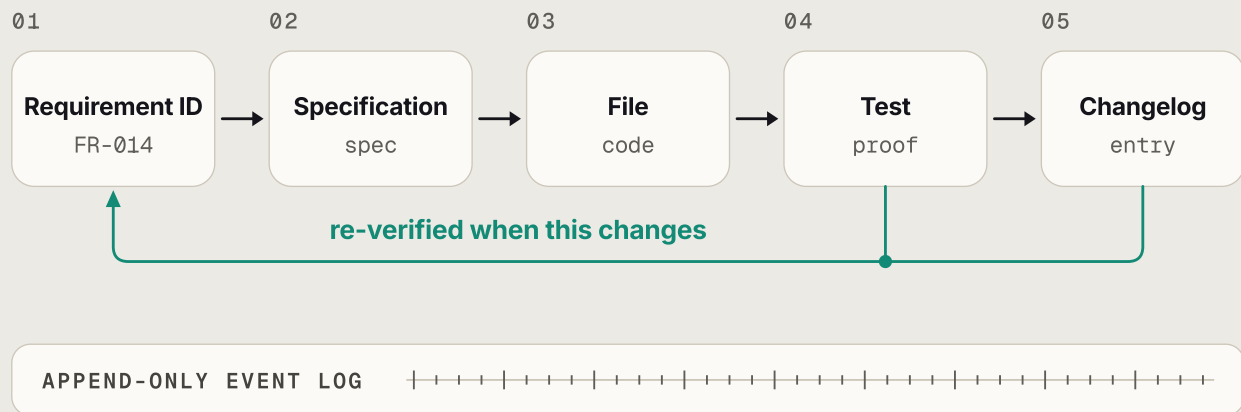


EXHIBIT 10 A requirement is only true while the chain behind it is re-checked.

In Shipwright the baseline has four parts:

- **Requirements with identifiers.** Each behavior has an ID and acceptance criteria, written so that a test can prove them.
- **An architecture document** that says how the product is built, updated by the process as part of each change and checked in review.
- **Decision records.** When a choice is made, the reason is recorded in the same pull request as the change.
- **An append-only event log.** Each step of the process writes an event to it. Reports such as the traceability matrix, the test evidence, and the change history are views of that log, regenerated when needed. A separate consistency audit reports where a view and the facts have drifted apart.

The log is a design choice with a reason. Summary documents written by hand rot, because they describe a project that has already moved on. A log that only grows, with views computed from it, gives the drift nowhere to hide.

Re-checking is the part that matters

Plenty of tools help an agent record what a product is. What takes extra work is keeping that record true as the product changes. In Shipwright two gates do this. When a change edits an acceptance criterion, the pull request has to name it, and the tests bound to it have to run again and pass before the merge. And a recorded change that affects behavior but names no requirement is refused.

The edge needs to be stated exactly. A change that alters code without touching any acceptance criterion does not trigger the first gate. That is why the baseline is one layer of four and not the whole answer. The result is called an **auto-re-verified baseline**, and it is what had to be built from scratch.

CASE

The first honest grade

On 27 June 2026 I ran Shipwright's own Control Grade on Shipwright. I had built the thing, so I expected a good result. It gave me a B, 88 out of 100, and put the biggest gap right at the top: my changes had quietly stopped tracing back to a requirement. Not all of them were still linked. Two more things it could not yet measure at all: whether changed behavior had been re-verified, and security. Those were the three gaps. Over the next days I closed them, and the grade climbed to an A. The number matters less than what it did: it handed me a list that showed where control had slipped, and I worked it down.

WHY IT LOOKS LIKE THIS

A baseline that exists only as prose will be contradicted in the first busy week. So the record is structured, with IDs a machine can follow, and a gate refuses a change that breaks it.

What this does not do

A baseline protects you against drift. It does not tell you whether you asked for the right thing. Whether the software serves the people who use it is still your call.

IN ONE LINE

A baseline is a record that gets checked. Without the checking it is documentation.

Layer two: make the rules something the system enforces.

1 BASELINE

• 2 HARNESS

3 LOOP

4 EVIDENCE

Question this layer answers: what happens when the AI ignores a rule?

Advice versus a gate

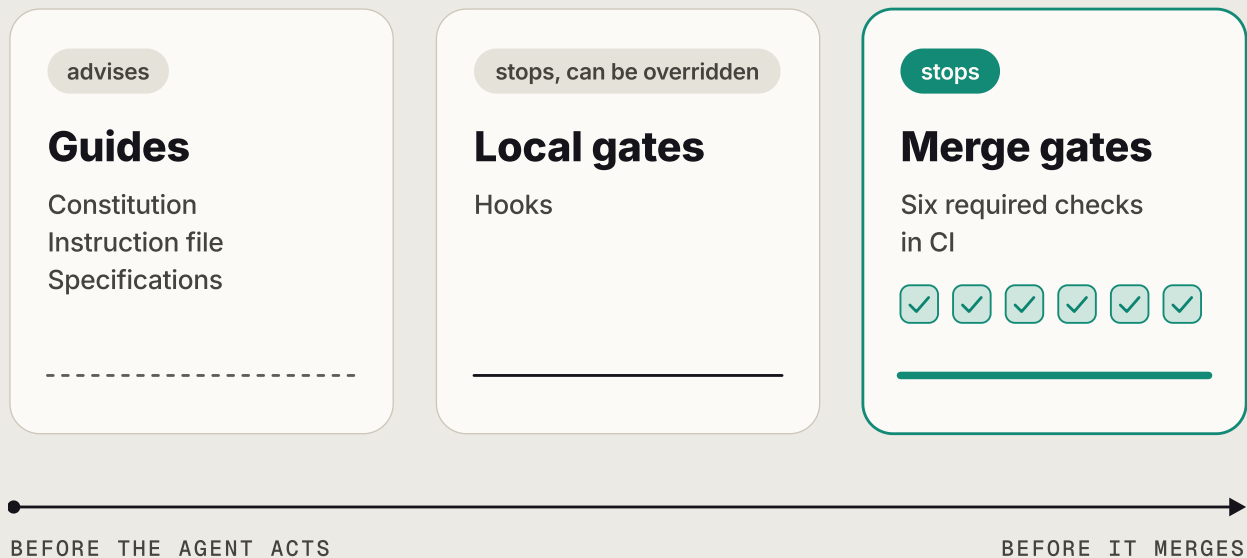
Böckeler describes the harness in two parts. The harness holds **guides**, which steer the agent before it acts, such as instruction files and rules. It also holds **sensors**, which check what the agent did, such as tests, linters, and reviews. Some checks are computational: a test passes or it does not. Others need judgment, so a second model gives an opinion.

The practical lesson is that an instruction is advice, and a **hook** that blocks an action is a gate. A hook is a small program that runs at a fixed point in the agent's work and can stop what is about to happen. Anthropic's guidance says the same: hooks are deterministic, instructions are not. Hooks are open to everyone who uses Claude Code, and other agent frameworks use them as well. What matters is what the hooks check, and how far across the lifecycle they reach.

Asking is not enough. For a rule that matters, it has to be certain that it was followed, and that is what enforcement means. Picture a machine in a cookie factory that fills the packs. Nobody there would think of asking the machine politely to put in the right number of cookies. Deterministic checks do it: a scale weighs every pack, a counter counts, a camera looks at the shape, and a pack that fails is blown off the belt. Nobody inspects every pack by hand, and still the pack that reaches the customer holds the right number of cookies, of the right

weight, looking the way they should. That is the job of the harness. It makes the checks deterministic, so that the loop can run without a person watching every step.

How Shipwright wires it



EXHIBIT

11

A hook reminds. A merge gate locks.

Local hooks stop an action and can mostly be overridden, a few are hard stops. The six required checks decide whether a change can merge.

Shipwright is a set of fourteen plugins for Claude Code. Seven of them are the phases that its orchestrator runs in order: **project, design, plan, build, test, changelog, deploy**. Iterate carries every change after the first build (chapter 5), and security, compliance, grading, and a few more run alongside. Each phase is its own plugin with its own hooks and tests, which keeps the surface of any one of them small.

The guides are a short instruction file and a **constitution** with three kinds of rules: what the AI always does, what it asks first, and what it never does. The sensors work at two levels. Architecture is covered too: a change that declares an effect on it is refused until the architecture document records it.

At the first level are hooks inside the working session. Two are hard stops: dangerous shell commands, and growth past the project's size baseline. Keeping edits inside an iterate's isolated worktree is enforced by a hook as well, but a person can release it. Most of the others stop the action but let a person continue. After the write they

flag a secret that landed in a file and warn about a destructive database migration. Compliance hooks stop a commit when requirement coverage falls below 80 percent, and block a deploy while critical security findings are open, and for these the override is recorded in a log. Some gates are soft on purpose, where a hard block would only annoy people without bringing the benefit of a hard stop, and a safety system that gets switched off protects nobody. The rules that must hold go to the place where they are hard: the merge gate.

At the second level are the **six required checks** on every pull request, which are the hard enforcement (chapter 5). A local hook is a reminder with teeth. The merge gate is the lock.

CASE

A rule that was only prose

Early on, my own process asked the AI to classify the effect of each change on the specification. It was a written rule. When I audited the history later, 27 out of 28 changes had never touched the specification. The decision record that followed puts it better than I can: a soft contract violated 27/28 times is not a contract. I moved the check into the system, so that it is no longer asked for. It is checked.

WHY IT LOOKS LIKE THIS

Instructions are read.
Gates are run. Where a rule matters, it lives in a gate.

The harness also costs something

Everything in an instruction file is sent with every turn, so a harness has a running cost, and it grows with the context an agent carries around times the number of turns. Context is treated as working capital. Shipwright keeps the entry file short, about two hundred lines, and follows the principle of **progressive disclosure**: the agent starts with a small set of always-loaded files and reads the rest only when the work needs it. Anthropic describes the same design for its Skills, where information is loaded in stages as needed, rather than all at once.

The decision history shows how this works in practice. Decisions are collected in one markdown log, with a much smaller index next to it that holds a title and a one-line summary per decision. The agent reads the index and opens a decision only when it matters. Karpathy described a close relative of this idea in April 2026 under the name LLM wiki: a set of interlinked markdown pages with an index, kept up to date by the model, instead of searching raw documents again on

every question. Two more habits keep the cost down. The first is compaction. The cost of a session is not the size of any one file. It grows with the number of turns times the context each turn carries, and a long session carries everything it has said and read, most of which is no longer relevant. Compacting replaces that history with a summary, so the later turns start lean. Claude Code can do this automatically at a context size that a team sets, and setting it is a practice Shipwright recommends. What matters is that the state of a run survives it. It is written to disk for all relevant artifacts: the plan, the review results, the decisions, and a session handoff with current state and next steps, written before compaction. A compacted or interrupted session picks up from there.

The second is the choice of model. A team can set one model for the work of building, which is the default, and separate models for plan review and code review. Two reasons to do so. A reviewer from a different model than the builder is not marking its own work (chapter 5), and a lighter, cheaper model can do the routine building while the strongest one is kept for review. The roles run as separate agents and not as a model switch inside one session, because a switch throws away the cached context and the next turn pays for it again.

So the harness has two jobs. It makes the rules that matter certain, and it stays light enough to carry through a long run. Both serve the same purpose: an agent that can work for an hour without a person, because what it needs is at hand and what must hold is checked. The next layer puts every change through that harness in the same loop.

IN ONE LINE

Enforce the rules that matter, and keep the harness light enough for a long run.

Layer three: put every change through the same loop.

1 BASELINE

2 HARNESS

• 3 LOOP

4 EVIDENCE

Question this layer answers: what happens between "I want this changed" and "it is merged"?

From prompts to loops

"I don't prompt Claude anymore. I have loops running..."

BORIS CHERNY, CREATOR OF CLAUDE CODE, QUOTED BY ADDY OSMANI, JUNE 2026

When people say "agentic," they often mean a long conversation. The practice that carries teams further is different. Instead of prompting an agent task by task, a team designs a system that prompts it, and the system keeps going until the checks say the change is done. Osmani ends his piece on the risk: build the loop like someone who intends to stay the engineer, not only the person who presses go.

A loop is work, plus checks, plus feedback. The harness from chapter 4 supplies the rules and the checks, and the loop decides when they run: in a fixed order, on every change, with no way to skip a step quietly. A loop without a harness only iterates faster in the wrong direction. A harness without a loop is a set of checks that nobody is required to pass.

One change, in six stages

The loop in Shipwright is called **iterate**. It runs for every change after the first build. This is what was used for the mobile fixes described in the introduction, in six stages. The full thirteen steps are in Appendix B.

The teal stages are gates: the loop does not go on while one of them fails. Changes come in three kinds: a new feature, a change to existing behavior, and a bug fix. A bug is first reproduced and traced to its root cause, and a failing test pins it before anything is fixed.

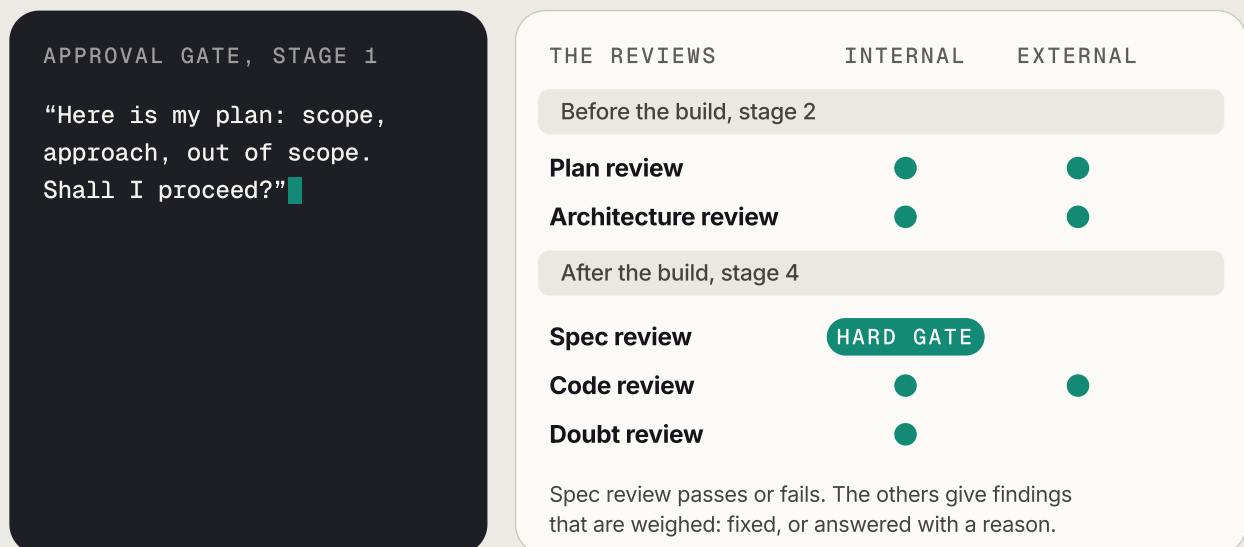
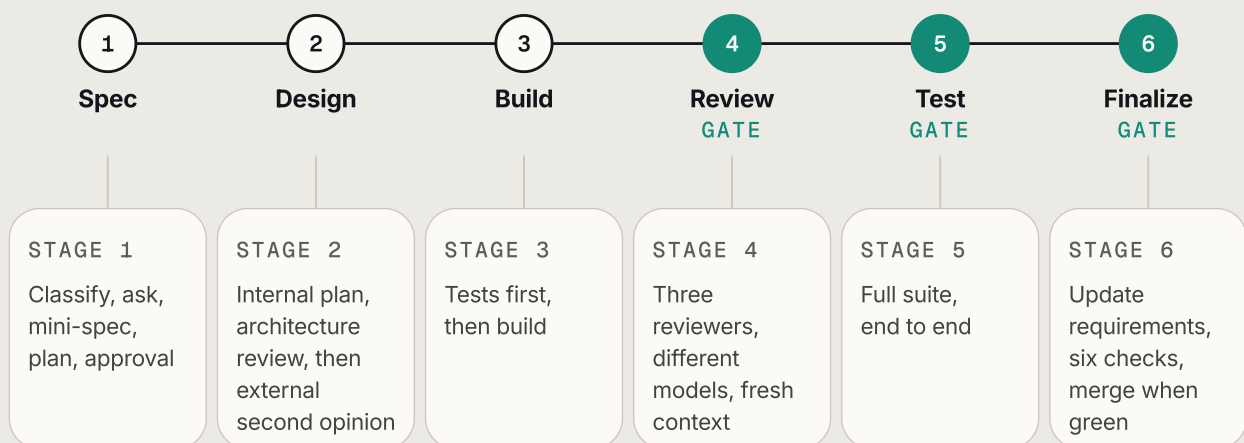


EXHIBIT
12

Every change takes the same route, and the gates sit where mistakes are expensive.

Full steps in Appendix B.

1. **Shape it.** First the change gets its own **worktree**, an isolated copy of the repository, and a hook blocks edits anywhere else. That is what lets several changes run side by side without their work mixing. It also checks that the main branch is green. Then the change is sized as trivial, small, medium, or large, and that decides how much of the process runs: a one-line fix skips the interview and the outside reviews, a medium feature gets all of it, large work is handed to the full pipeline, and every change ends with the same finalization. Risk flags such as login, billing, or database migrations set a minimum, so such a change is never trivial. From medium upward the system asks one to three questions, writes a short specification of the change, and states its plan to the user: scope, approach, and what stays untouched. It also records how the change touches the requirements: one is added, modified, removed, or none is affected. Nothing is built until the user approves the plan or corrects it.
2. **Have the plan reviewed, before anything is built.** Once the plan is approved, it is reviewed while nothing exists yet. Two internal reviewers on a strong model go first. The plan review checks the plan against the specification for security, performance, and architecture problems. The architecture review sees only a short brief and the specification, never the plan, and asks whether the change should be built at all. Then two models from other vendors repeat both reviews on the same material and give a second opinion. They may disagree with the internal review and with each other. If this external review cannot run, for example because a key is missing, the process stops and asks.
3. **Build it, tests first.** A failing test comes first, then the code that makes it pass. For a change to existing behavior, the existing tests are adjusted first. The agent then reads its own diff and confirms that every behavior it added is either tested or has a written reason why not.
4. **Have the result reviewed.** Each reviewer starts with a fresh context, so none of them inherits the author's assumptions. The specification reviewer comes first and is a hard gate: does the change do what the spec says? It passes or fails, and on a fail the change is fixed and reviewed again before anything else happens. A code reviewer follows. For risky changes, such as migrations, concurrency, or irreversible operations, a third reviewer follows whose only job is to disprove the work. These last two do not return a yes or no. They return findings, and a finding is a judgment that has to be weighed: the author fixes it, or answers it with a reason why not.

5. **Prove it.** From medium upward the project's entire test suite runs again, not just the new tests, and smaller changes run the tests related to them. Any failure is fixed before the loop goes on. From medium upward the application is then started and the new behavior is checked from the outside, end to end. A check that was written but not run counts as no test.
6. **Deliver it.** First the baseline is brought up to date: the affected requirement, or a new requirement with a new unique ID, plus the architecture document, the decision record, the changelog entry, and the test results. The checks that CI will run are first run locally, then the change goes to GitHub as a pull request, where six required checks run in CI. When all six are green, the pull request merges by itself. If one fails, the agent reads the logs, fixes the problem, and pushes again. The change is done when it is merged and green, not when it is sent. Last, a short reflection records what was learned, for the next run to read. This is how the harness improves: a mistake that happened once is written down so that it does not happen again.

The six required checks run in CI on every pull request:

- **Code analysis, twice.** GitHub's CodeQL scans the Python and the JavaScript and TypeScript code, one run each, for security and quality problems.
- **Lint and tests.** A linter checks style and common errors, and the project's whole test suite runs on a clean machine.
- **Security scan.** It looks for unsafe code patterns, dependencies with known vulnerabilities, and secrets that were committed by mistake.
- **Anti-bloat.** Files may not grow past a size limit, measured against a frozen baseline.
- **Independent review.** A separate reviewer reads the pull request without the author's context.

A check that cannot run does not count as a pass. This is called **fail-closed**.

The review cascade of stage 4 always runs from medium upward, and the code is also reviewed by models from other vendors, in the same way as the plan in stage 2. For smaller changes it runs when a risk flag is set or the change is larger than 100 lines. The result of every review is recorded, and the loop does not finish while one is still open: a review that did not run has to say why.

Most of the quality comes from the front. A finding in the plan review costs minutes, and the same finding after the build costs the build. That is why the plan is reviewed by internal reviewers and by other vendors' models before a line of code exists, and the result is reviewed again afterwards. One internal reviewer never sees the plan on purpose, because a reviewer who works inside the plan's own frame tends to confirm it.

What went wrong once

CASE

The change that should not have been built

One of my iterates passed three rounds of review, and about 25 findings were fixed along the way. Nobody asked whether it should have been built at all. Reviewers who work inside the plan's own frame do not ask that. When I gave the same work to two reviewing models as a plain brief, without the reasons behind the plan, both rejected it. Since then an architecture review runs before the external review and asks that question on every plan and every medium-or-larger change. It has no trigger that the author can forget to set, because a trigger the author sets fails on exactly the changes that most need the question.

WHY IT LOOKS LIKE THIS

A self-review confirms what the author already believes. A reviewer needs a different context and, for the changes that matter, a different model family.

What the loop gives back

Every stage costs time that a plain prompt would skip. A change like this used to take days. Now it takes minutes, and the loop adds a few minutes more than an unchecked prompt would. But because it is a loop, nobody sits next to it and watches. A change is handed over, runs on its own, and is finished at some point, and what comes back is a merged change with the evidence behind it. A few extra minutes are worth it to keep control.

This is where the two layers meet. The harness decides what must hold, and the loop makes sure it is checked on every change, in the same way, whoever or whatever made the change.

IN ONE LINE

A few extra minutes per change, and what comes out is what you wanted: cleanly engineered, tested, and backed by evidence.

Layer four: make the result readable.

1 BASELINE

2 HARNESS

3 LOOP

• 4 EVIDENCE

Question this layer answers: how do you know, without asking the AI, that the software is under control and still does what the business intends?

A number that shows the dark spots

Looking at a repository, can you tell how much control the team has? The **Control Grade** is one answer. It scores a repository from A to F, zero to a hundred, on **how much control it can prove**, not on how good the code is. It asks whether you can follow what the software does, trace it to what you asked for, and trust that it has not drifted.

It combines seven dimensions with fixed weights.

The method is modeled on the OpenSSF Scorecard, which uses a weighted average and leaves out any check that cannot be measured. An unmeasurable dimension does not count as zero, so you never get an unfair F for something the tool could not see.

Two guards keep the number honest. An average can hide a dark spot, so the headline is capped at 89 when a control the repository is set up to measure is dark. It is capped at 49 when one of the load-bearing parts, test health, change traceability, or security, collapses below half. Whether a team ships mostly features or mostly maintenance is deliberately not part of the grade. It measures control, not what kind of work you did that month.

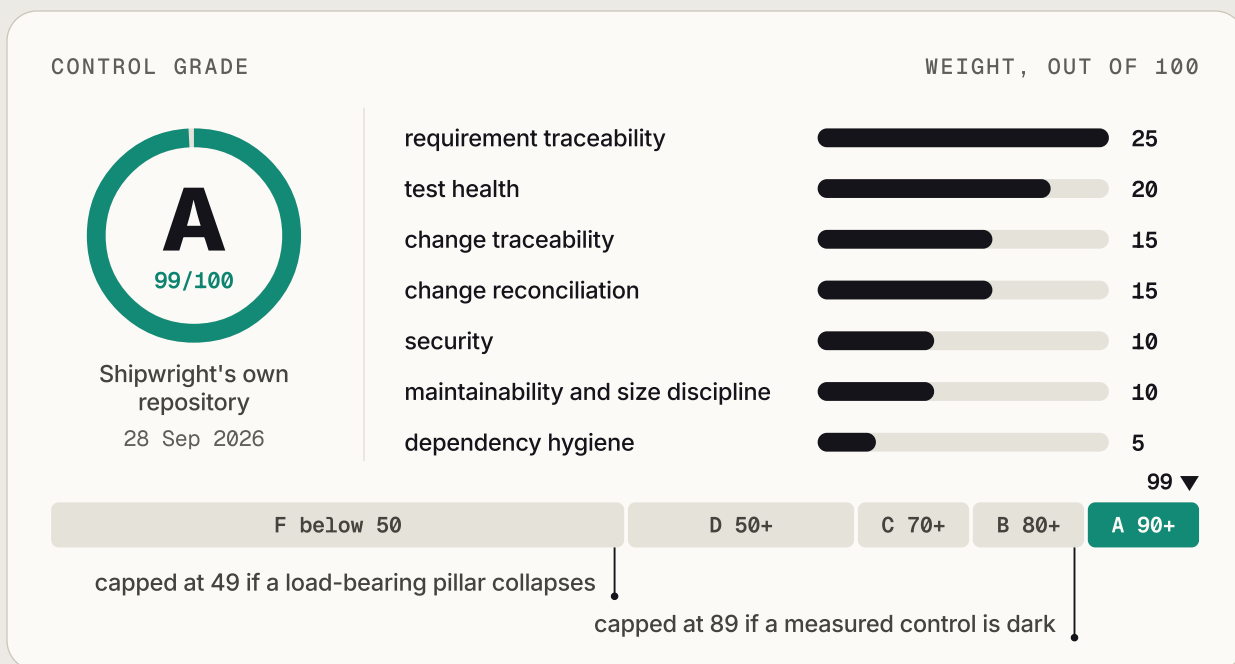


EXHIBIT
13

One grade, seven dimensions, and caps that stop an average from hiding a dark spot.
Shipwright's own repository, as of 28 September 2026. Weights sum to 100.

A repository graded from outside, without Shipwright's own records, can reach at most a B. That is a heuristic and is labeled as one.

Shipwright's own repository stands at an A, 99 out of 100, as of 28 September 2026. One caution belongs next to that number: it is the tool grading its own repository. Read it as a demonstration of the method, not as independent proof. The grader is open source, so you can run it on your own repository and see what it says.

Documents that come with the grade

The grade is computed from the records the work leaves behind: the event log from chapter 3 and the repository. The same records produce compliance documents as a by-product, regenerated whenever they are needed: a requirement traceability matrix, test evidence, a software bill of materials, a change history, a dashboard, and an audit report. They exist because the log exists, so an audit request needs little extra work. The aim of the setup is control, and compliance is a side effect of it.

IN ONE LINE

Evidence you have to ask for is not evidence, so build it to be read.

The Command Center: loop engineering you can see.

Shipwright's plugins are the harness and the loop. The **Command Center** is where you watch and steer that loop: a local web app that makes the work moving through the harness visible and steerable, while the terminal and the files stay authoritative. A loop that runs without you needs a place where you can see where each change stands, step in on an exception, and decide. That is what turns a loop from something you launch into something you steer.

CASE

Why I built it

Working in VS Code, I eventually lost the overview: several windows, and several tabs in each. I also needed one simple place where the CI and the loops could bring something to my attention, which became the triage. And I wanted to see at a glance when an iterate needs something from me. That became the inbox.

One command installs and starts it:

```
npx @svenroth-ai/shipwright@latest
```

The Command Center is optional. Every Shipwright plugin works without it, in VS Code or in the terminal where Claude Code runs.

How it works

The Command Center embeds a real terminal, built on the same components as the one in VS Code, with a file tree and a smart document viewer beside it. That puts files, terminal, and previews in one place. It does not wrap Claude Code: a change starts by running Claude Code in that terminal with the standard start command. That is also why it needs no API key of its own: it runs on the standard Claude Code subscription.

The board, the inbox, and triage

The task board shows every piece of work across all your projects, from backlog to done, with filters by project and a list view. An iterate appears as a card with its state. Above the columns, lanes show running pipelines with their phases and campaigns with their progress. The board is not limited to code: for example, a content marketing project runs on the same board.

Two lists sit beside it. The **inbox** collects every "Claude needs permission" prompt from all projects in one place, and one click takes you to the terminal where you answer it. The **triage** list holds findings from Shipwright's quality, security, and compliance hooks and from CI, ready to be promoted into tasks.

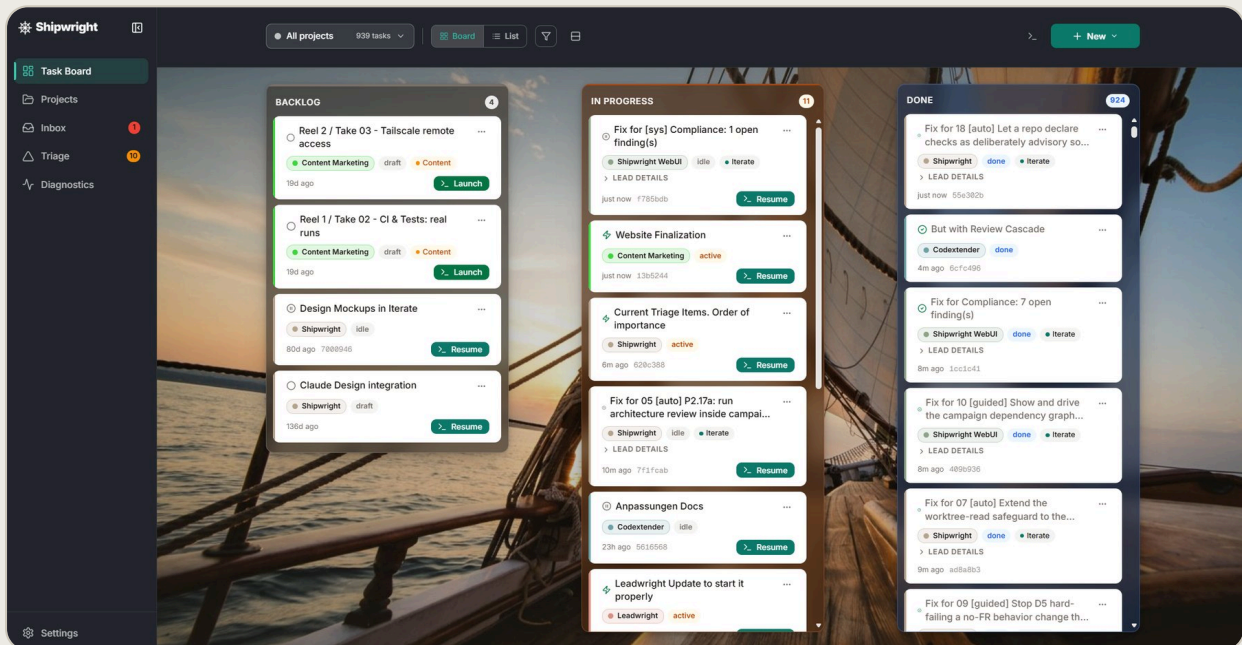


EXHIBIT
14

Every piece of work sits on one board, from backlog to done.

The Command Center task board. Screenshot from Shipwright's own work.

A task, from start to finish

A change starts from the **+ New** menu: a single phase, a full pipeline, an iterate on a finished project, or a plain Claude session. A dialog takes a title, an optional description that becomes the first prompt, and the autonomy: guided, where the AI asks before each major step, or autonomous. It shows the exact command before anything runs. The task then goes to the backlog, or, on **Launch**, into progress.

Open a task and a switch at the top offers two views. **Mission** is a simple view of what happens in the terminal: the session as a readable transcript, like the one in the Claude Code desktop app, without the terminal's control output. It updates live, so there is no need to scroll through terminal output. **Files & Terminal** is the work itself. The terminal is on one side, started with the prepared command, and a file tree and a smart document viewer sit beside it. The viewer opens any file in the project: Markdown renders, code is highlighted, images and short videos play inline, and diffs are marked. Markdown files can be edited in place, with a diff shown before saving and a conflict prompt if the file changed on disk in the meantime.

The terminal hosts only an allowlisted shell, and the click on Launch is what authorizes it. A diagnostics page shows the CLI version, the session count, and the health of the watcher.

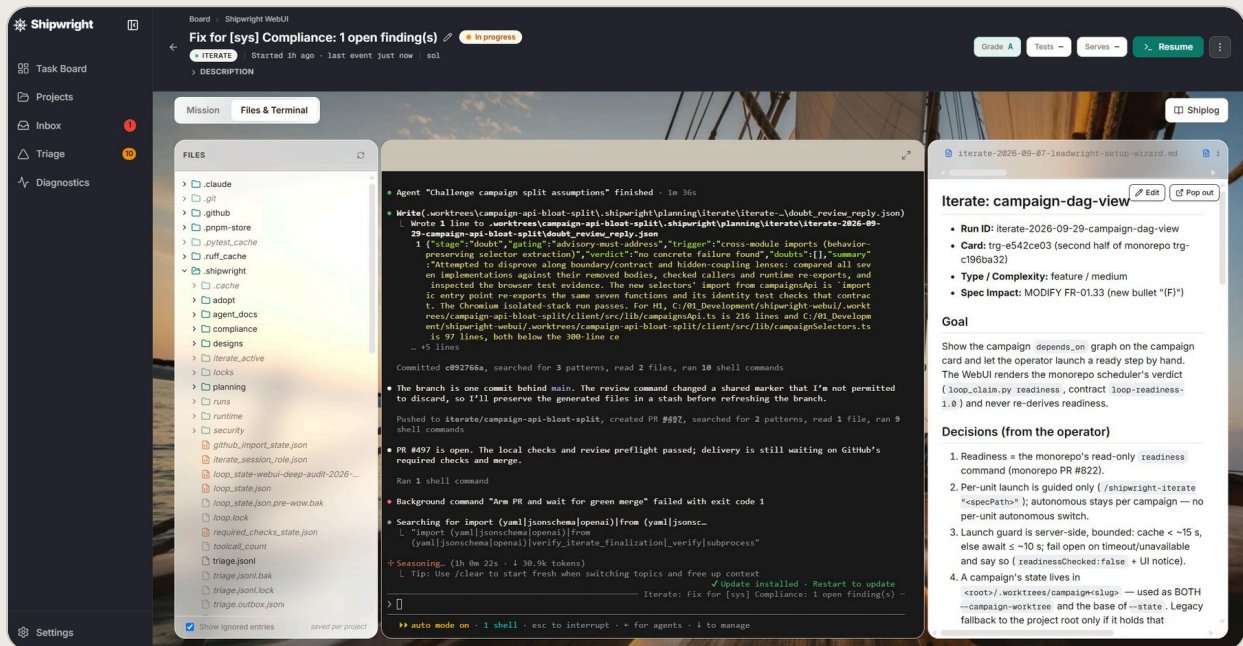


EXHIBIT
15

A task opens onto the work itself: the file tree, the terminal, and a viewer for any document.

The Command Center task view. Screenshot from Shipwright's own work.

The Ship's Log: a project's home between runs

Each project has a **Ship's Log**. At the top is the Control Grade, with the dimensions behind it. Below, every run is an entry in the logbook, with the requirements it touched and, where recorded, the outcome of its review, test, and security gates. On the right are the project documents, grouped as specifications, agent documents, and compliance. A box at the top prepares the next change, which you then launch from its task page. Everything else in the Ship's Log is read-only: it shows the evidence and does not change it. It is where you land when you come back to a project after days away: you see the grade, the last runs, and the documents, without opening a terminal.

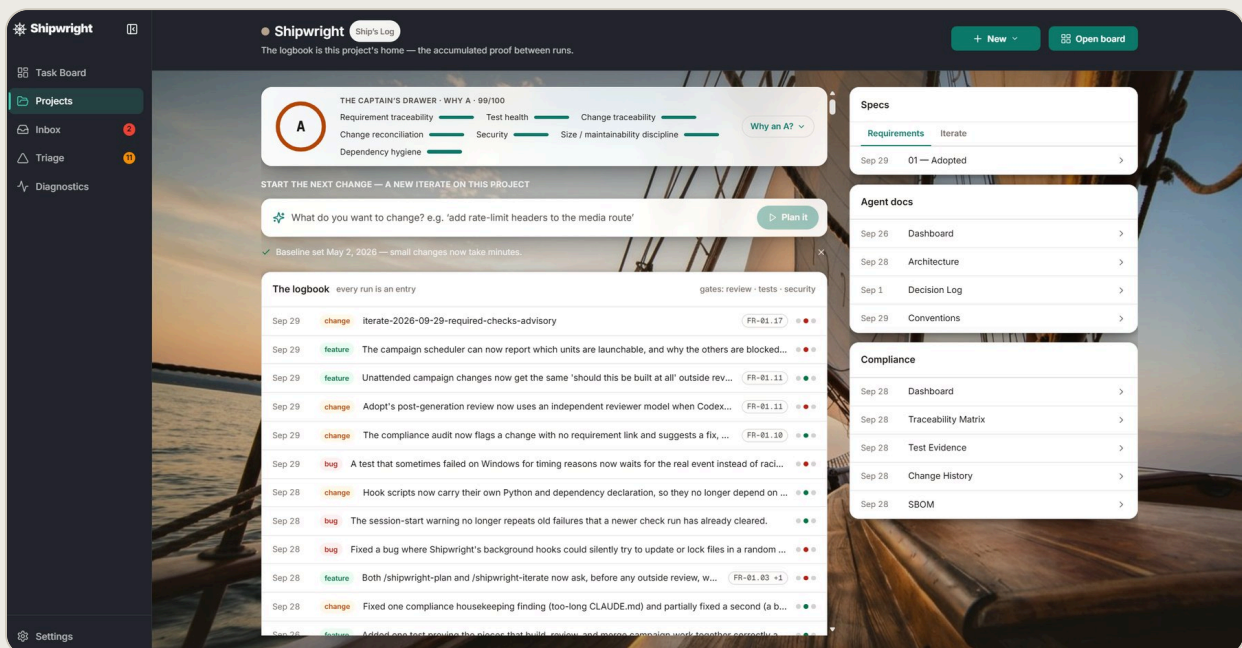


EXHIBIT
16

Each project has a logbook: the grade, the reasons, and every run as an entry.

The Command Center Ship's Log, read-only. Shipwright's own repository, 29 September 2026.

IN ONE LINE

The interface shows the loop, and the terminal and the files stay the source of truth.

Your role changes before your tools do.

Question this chapter answers: if the AI does the typing, what is left for the human, and how does the role develop?

Five kinds of builders

Boris Cherny, the creator of Claude Code, described five archetypes for AI-native teams, first in a post on X at the end of June 2026 and then in more depth on Bloomberg's Odd Lots podcast in July. He argues that the old split into engineering, product, design, and data science stops describing what people do. What replaces it is five archetypes tied to a stage in a product's life, not to a title.

The five are the Prototyper, who comes up with ideas and iterates fast; the Builder, who turns a validated idea into a product; the Sweeper, who cleans up and removes what should not have shipped; the Grower, whom he also calls a scaler, who pushes a product that has found its market further; and the Maintainer, who keeps a mature system secure, reliable, and fast. The exhibit below shows them.

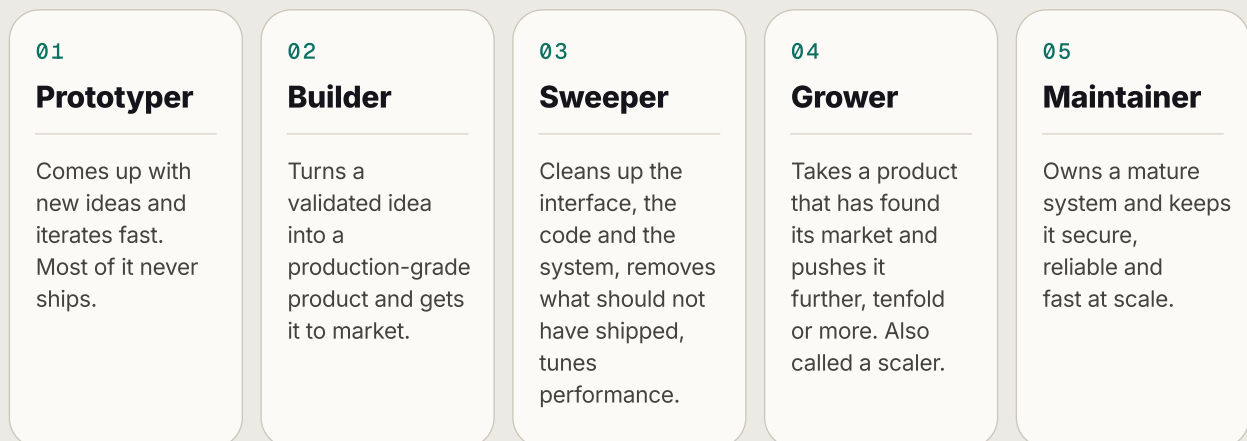


EXHIBIT
17

Five archetypes for AI-native teams, tied to the stage of a product.

Archetypes: Boris Cherny, June and July 2026.

What changes, and what stays

When code stops being the bottleneck, a team no longer has to be cut by function. An idea used to pass through several hand-offs, from product to design to engineering to whoever ran deployment, and every hand-off lost something. A team also had to cover every stage with the people it had. A builder was often the maintainer too, and a whole organization handed everything downstream to one team because that was the only way to cover it, not because it matched anyone's talent.

With a clean harness for each stage of the work, that changes. Anyone can use a harness, because the rules, the checks, and the gates are in the system and do not depend on who runs them. What cannot be handed to the system is responsibility. Someone has to own the harness of a stage: decide which checks this kind of work needs, how strict they should be, and where a hard stop would only get in the way. A throwaway prototype and a system that handles other people's money need different harnesses, and someone has to keep evolving them.

This is where Cherny's archetypes fit. Read as ways of taking responsibility and not as job titles, each one belongs to a stage of a product's life and brings its own strength: iterating fast for the Prototyper, production-grade work for the Builder, pushing a product further for the Grower, cleaning up for the Sweeper, keeping a system secure and reliable for the Maintainer. The link to the harness is this paper's own reading. It means a team can organize around who is good at which stage, and the mindset to keep the harness fit for that stage counts for more than the function on a business card.

What stays is steering.

From relay to steering

A second shift happens inside one person's day. In an early setup the human is the relay between agents: passing results along, running checks, asking for status, answering the same questions again. It is busy, and it does not scale. It is also where the cost of AI quietly grows, because every hand-off costs tokens and attention.

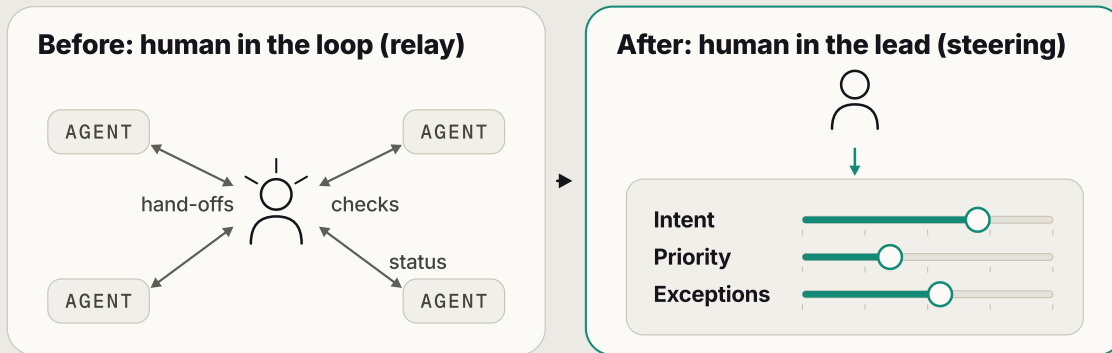


EXHIBIT 18

The human moves from the relay in the middle to the lead above the board.

Adapted from Cherny and Osmani; the drawing is the author's.

As the setup matures, the human steers. A board shows the work. The person sets **intent** and **priority** and handles **exceptions**. Routine hand-offs move into the system. That is what the Command Center is for.

The short version is **human in the lead, not human in the loop**.

Shapiro's level 3 describes the person as "the human in the loop", the full-time reviewer of what the AI wrote. That role does not scale, and it puts control where the least capacity is. In the lead, the person decides what gets built, in which order, and what counts as an exception, and reads the evidence. The loop runs without the person standing in the middle of it. The AI does the work, and the person decides that it should be done, and whether it was done right.

What comes next

A companion project, Leadwright, takes the steering role one step further. Autonomous agents are woken on a schedule, claim the next task in their domain from the Command Center's backlog, and work within budgets and authority limits that a person sets in an org chart. It is in active development and not yet released. It is mentioned because it shows where the human role is heading: from doing the work, to assigning it, to writing the charter and role description of agents that pick it up and deliver on their own.

IN ONE LINE

People stay accountable, and what changes is where they stand: in the lead, not in the loop.

Where this fits, and where it stops.

The landscape, described by audience

Many readers of this paper will know other frameworks that put structure around an AI agent. This section places Shipwright among them: why it exists, and what it is for. It does not compare them. Each of these efforts has a clear purpose of its own, and its authors describe it best, so their documentation is the place to look.

- **agent-skills** by Addy Osmani, **Superpowers** by Jesse Vincent, and **Matt Pocock's skills** are collections of skills and workflows for developers. They are meant to be operated by people who read and shape the code themselves.
- **spec-kit**, GitHub's open-source toolkit for spec-driven development, gives an agent a structured, written specification to work from.
- Browser-based builders such as **Replit** take a sentence to a running application with the whole stack included.

Shipwright is built around the combination of the two ideas from chapters 2 to 5: a **harness** that holds the rules and the checks, and a **loop** that runs them on every change, from the first plan to the merged pull request. The practices of a careful engineering team live in the system, so shipping with AI does not depend on knowing every one of them. That is what makes it usable for **tech-savvy users** who want to ship in a controlled way. They keep the business intent in their own hands, and the baseline is re-verified against it after every change. The Control Grade and the Command Center show at any time where things stand.

For now, Shipwright is built and supported on Claude Code only, and that is a sequencing choice. The most visible reason is the hooks: they enforce steps across the whole lifecycle, and that is easiest to make reliable end to end on one platform first. A port also has to cover how the plugins are packaged and installed, how reviewers and

other agents are started, which instruction files an agent reads, how the Command Center starts and resumes sessions, and the transcript and cost tracking. What does not change is the core. The Python scripts, the artifacts, and the phase logic run the same on any platform.

A second platform still came into view, for a practical reason: usage limits. Shipwright lets so much work run in parallel that the limits of a single plan become the bottleneck. A Codex path, Codex Light, has been started, but porting everything will take time. To speed things up, [Codextender](#) was created. This separate project does not depend on Shipwright. It lets Claude Code sessions run on the models of a Codex plan through a local LiteLLM proxy, which doubles the available limits with little effort. It was built by the author of this paper.

The limits

Where a tool stops matters as much as what it does. These are the limits of Shipwright, and of what this paper claims.

- **It does not make AI build faster, and it does not replace your judgment.** The loop adds minutes to every change. Whether the software does the right thing for the people who use it stays a human decision.
- **Behavior is the least covered area.** Tests and reviews check the code against the requirements. Judging the non-deterministic parts of an agent's output takes a formal evaluation harness, which runs the agent on a fixed set of examples and measures how often it gets them right. That is not included yet.
- **The reviewers are models, and models are sometimes wrong.** The reviewer behind the CI pull-request gate was replaced after it repeatedly returned confident blocking verdicts that were made up. Independent review lowers the risk of a missed problem. It does not remove it.
- **A grade from the outside is an estimate.** It tops out at a B, because the full grade needs Shipwright's own records.
- **The compliance records are a by-product.** They are useful to an auditor, but they are not a certification.

IN ONE LINE

Say what a tool is for, and say where it stops, because both help the person deciding.

A maturity model for control, and ten questions.

The five levels, seen from the side of control

Dan Shapiro's ladder from January 2026, published as the five levels but numbered 0 to 5, runs from "spicy autocomplete" up to the dark factory. They describe how much of the work the AI does and what your job becomes. The Google paper *The New SDLC With Vibe Coding* draws a similar line, a spectrum from vibe coding to agentic engineering. Shapiro's levels can be read as the finer version of that spectrum, and they serve as the ladder here, because they are well known and they say clearly where the human stands.

What this paper adds is the control that each level needs. The further the human steps back, the more the control has to move out of their head and into the system. At level 5 the factory is dark: nobody watches the work, so nothing can be controlled by looking. That is why control is not an extra at the top of the ladder. It is what makes the top of the ladder possible.

the further the human steps back, the more control must move into the system

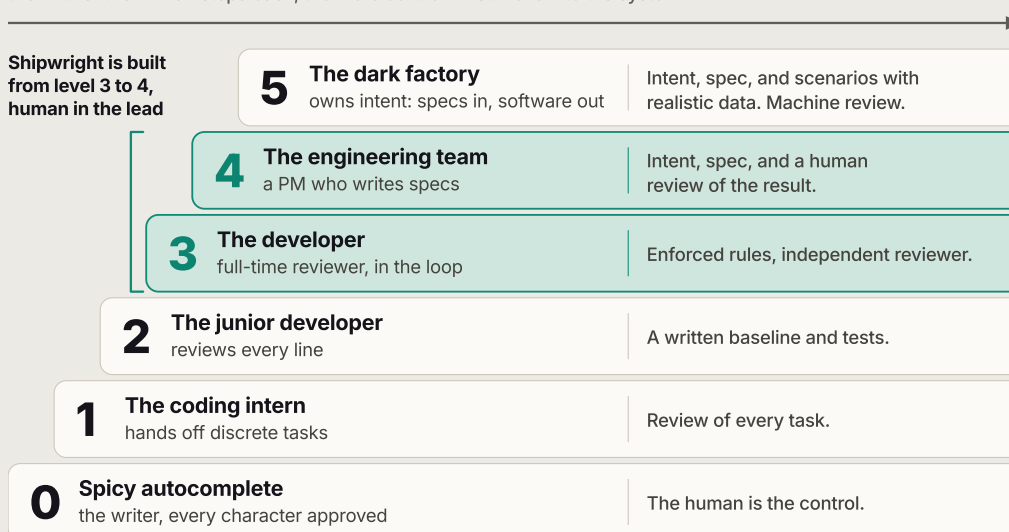


EXHIBIT
19

The further the human steps back, the more control has to move into the system.

Levels and names: Dan Shapiro (January 2026). The control column is the author's.

Shipwright is built for the step from level 3 to level 4, with the human in the lead. It is not a dark factory, but it covers part of the control that level 5 depends on. What is still missing is on the testing side. The end-to-end specs of the test plugin catch regressions, but the builder can see them. A dark factory needs scenarios with realistic data that the builder cannot reach, run repeatedly against stand-ins for third-party services, with the pass rate as a merge condition.

Ten questions for any AI-assisted way of building

- 01 **At which level do you work today, and at which level does your work need to be?** A prototype can stay at level 1. A product other people depend on cannot.
- 02 **Where does a person really look, and where does nobody?** If nobody controls, the process is worse than without AI.
- 03 **Is there a baseline,** a written record of what the product should do, checked again after every change?
- 04 **Does every change trace to a requirement,** and a test?
- 05 **Is there a regression test suite that is kept up to date?** Are its tests run again when a requirement changes, and the whole suite before something counts as done?
- 06 **Are the important rules enforced by the system,** or only requested in prose?
- 07 **Is the reviewer a different one from the author,** with a fresh context, before the build as well as after it?
- 08 **When a check cannot run, does the change stop?**
- 09 **Do plans, decisions, and lessons survive the session?** They belong in files the next session reads, not in a context that gets compacted.
- 10 **Can you, or someone else, check the state without asking the AI,** and does the measure make a dark spot visible?

What to do on Monday

Pick one question you cannot answer, and answer it first. Often the earliest gap is in questions 3 to 5: no written baseline, no link from change to requirement, no re-run of tests when a requirement changes. Close those before you add anything else. If you are not sure where you stand, run the Control Grade on a repository you care about and read the list it gives you. If questions 8 to 10 have good answers, the control no longer depends on a person watching.

Writing code is no longer the hard part

What is hard now is keeping the software true to what you decided, on every single change. Three points are worth carrying out of this paper.

- **Control is a process that can be built and measured.** Four layers: a baseline that is checked, a harness that enforces, a loop that every change goes through, and evidence you can read.
- **The people stay accountable, and they lead.** They stop typing and relaying, and start directing: intent, priority, exceptions.
- **What you cannot measure you will not keep.** Put a number on control, and let it show you the dark spots.

The Google paper *The New SDLC With Vibe Coding* sums up these three points in a single sentence:

Generation is solved. Verification, judgment, and direction are the new craft.

OSMANI, SABOO, KARTAKIS, GOOGLE, MAY 2026

Shipwright is open source and on GitHub. Check the project page for more information.

github.com/svenroth-ai/shipwright
svenroth.ai/shipwright

Ship right, **not just fast.**

Reference material

A **How the Control Grade is calculated**

B **The change loop in thirteen steps**

C **Timeline**

D **Glossary**

E **Sources**

The argument ends with the conclusion. What follows is reference material: the grade formula, the loop step by step, a timeline, a glossary, and the sources.

How the Control Grade is calculated

- Seven dimensions with weights that sum to 100: requirement traceability 25, test health 20, change traceability 15, change reconciliation 15, security 10, maintainability and size discipline 10, dependency hygiene 5.
- The grade is a weighted average. A dimension that cannot be measured is left out of the denominator. If nothing can be measured, the result is "not gradeable", never an F.
- Bands: A from 90, B from 80, C from 70, D from 50, F below 50.
- Caps: 89 if a control the repository is set up to measure is dark; 49 if test health, change traceability, or security collapses below half.
- A repository graded from outside the framework tops out at B.
- The method follows the OpenSSF Scorecard: weighted checks that add up to a single score.

The change loop in thirteen steps

1. **Classify.** The change gets its own worktree and the main branch is checked to be green. Then it is sized as trivial, small, medium, or large, and risk flags set a minimum. Large work is handed to the full pipeline.
2. **Interview.** From medium upward the system asks one to three targeted questions. Facts that can be found in the code are looked up, not asked. Anything not answered is marked as assumed.
3. **Mini-spec.** From medium upward the change gets its own short specification: goal, acceptance criteria, which requirements it adds, modifies, or removes (or none), and which boundaries it touches.
4. **Approval gate, then plan review.** From medium upward, before any code, the system states the plan, and a person approves or corrects it. Then internal plan and architecture reviewers look at it, followed by an external second opinion from other vendors (if that cannot run, the process stops and asks).
5. **Design check.** Only if the interface is affected.
6. **Build, tests first.** A failing test comes first. For a change to existing behavior, the existing tests are adjusted first.
7. **Self-review and confidence check.** The agent reads its own diff. Every testable behavior is tested, or the reason it is not is written down.
8. **Review cascade.** Specification reviewer (hard gate), code reviewer, and for risky changes a reviewer biased to disprove, whose findings are weighed. From medium upward, models from other vendors review the code too, and for smaller changes they do when a risk flag is set or the change is larger than 100 lines. Every review is recorded.

9. **Full verification.** From medium upward the entire test suite runs again, and smaller changes run the tests related to them. Any failure is fixed before the loop goes on.
10. **End-to-end run.** From medium upward the application is started and the feature is run.
11. **Finalize.** The affected or new requirement, architecture document, decision record, changelog entry, and test results are updated, the CI checks run locally first, and a short reflection is recorded. The commit follows a standard format.
12. **Six required checks.** Pull-request gates that decide whether a change can merge.
13. **Merge.** Auto-merge is armed. If a check fails, the agent reads the logs, fixes, and pushes again. Done means merged and green.

APPENDIX C

Timeline

DATE	EVENT
02.02.2025	Karpathy's post that names vibe coding
24.02.2025	Claude Code released as a limited research preview, with Claude 3.7 Sonnet
22.05.2025	Claude Code generally available
10.07.2025	METR randomized trial
November 2025	Reliability threshold crossed (Willison)
23.01.2026	Shapiro's five levels, up to the dark factory
04.02.2026	Karpathy proposes agentic engineering
05.02.2026	Hashimoto describes harness engineering
02.04.2026	Böckeler on guides and sensors
May 2026	Google paper: The New SDLC With Vibe Coding
08.06.2026	Osmani on loop engineering
27.06.2026	First Control Grade of Shipwright's own repository: B, 88

Glossary

Agentic engineering: Karpathy's name for professional software work in which AI agents write the code and a person steers, specifies, and verifies it.

Baseline: The written statement of what a product is meant to do, linked to code, tests, and decisions, and checked when it changes.

Command Center: Shipwright's local web interface: a board, an inbox, an embedded terminal, and the Ship's Log.

Constitution: A short list of rules for the AI: what it always does, what it asks first, and what it never does.

Control Grade: A score from A to F for how much control a repository can prove.

Dark factory: A setup in which machines also do the review, using scenarios with realistic data, so that no person reads the code.

Event log: The record of what happened in a project, written by every run. The Control Grade and the compliance documents are computed from it.

Fail-closed: If a check cannot run, the work stops. A missing result does not count as a pass.

Gate: A check that has to pass before work moves on.

Guide / sensor: A guide steers the agent before it acts. A sensor checks afterwards.

Harness: Everything around the agent that steers it and checks its work: rules and checks.

Harness engineering: Designing everything around the agent that steers it and checks its work.

Hook: A small program that runs at a fixed point in the agent's work and can stop an action.

Human in the lead: A setup in which a person sets intent and priority and handles exceptions, while the loop runs without them standing in the middle.

Iterate: Shipwright's loop for changes to a product that already exists.

Loop: A system that prompts the agent, checks the result, and feeds back, until a goal is met.

Loop engineering: Designing the system that prompts the agent and keeps it going until the checks say a change is done.

Progressive disclosure: Loading information in stages, as the work needs it, instead of all at once.

Ship's Log: A project's home in the Command Center: the Control Grade, every run as an entry, and the project documents.

Token: The unit in which AI models are measured and priced, roughly a short word or word fragment.

Triage: A list of findings from hooks and CI that lie outside the current change and can be promoted into tasks.

Worktree: An isolated working copy of a repository, so that parallel changes do not mix.

Sources

- 1 Karpathy, vibe coding post, 02.02.2025. <https://x.com/karpathy/status/1886192184808149383>
- 2 Karpathy, agentic engineering post, 04.02.2026. <https://x.com/karpathy/status/2019137879310836075>
- 3 Anthropic, Claude 3.7 Sonnet and Claude Code preview, 24.02.2025. <https://www.anthropic.com/news/claude-3-7-sonnet>
- 4 Anthropic, Claude 4 and Claude Code general availability, 22.05.2025. <https://www.anthropic.com/news/claude-4>
- 5 Willison on the November 2025 threshold, 02.04.2026. <https://simonwillison.net/2026/Apr/2/lennys-podcast/>
- 6 Osmani, Saboo, Kartakis, The New SDLC With Vibe Coding, Google, May 2026. <https://www.kaggle.com/whitepaper-the-new-SDLC-with-vibe-coding>
- 7 METR, early-2025 developer study, 10.07.2025. <https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/>
- 8 METR, uplift update, 24.02.2026. <https://metr.org/blog/2026-02-24-uplift-update/>
- 9 CodeRabbit, State of AI vs human code generation, 17.12.2025. <https://www.coderabbit.ai/blog/state-of-ai-vs-human-code-generation-report>
- 10 Veracode, GenAI code security report, 2025. <https://www.veracode.com/blog/genai-code-security-report/>
- 11 Hacker News thread, "After Two Years of Vibecoding, I'm Back to Writing by Hand". <https://news.ycombinator.com/item?id=46765460>
- 12 Shapiro, The five levels, 23.01.2026. <https://www.danshapiro.com/blog/2026/01/the-five-levels-from-spicy-autocomplete-to-the-software-factory/>
- 13 Willison on StrongDM's software factory, 07.02.2026. <https://simonwillison.net/2026/Feb/7/software-factory/>
- 14 Hashimoto, My AI Adoption Journey, 05.02.2026. <https://mitchellh.com/writing/my-ai-adoption-journey>
- 15 Böckeler, Harness engineering, 02.04.2026. <https://martinfowler.com/articles/harness-engineering.html>
- 16 Osmani, Loop Engineering, 08.06.2026. <https://addyo.substack.com/p/loop-engineering>
- 17 Cherny, five archetypes for AI-native teams, X, end of June 2026. <https://x.com/bcherny/status/2071379474277613732> - and Odd Lots, Bloomberg, 20.07.2026. https://www.youtube.com/watch?v=7C_IHWkHKmU
- 18 Anthropic, Claude Code best practices. <https://code.claude.com/docs/en/best-practices>
- 19 Anthropic Institute, recursive self-improvement, May 2026. <https://www.anthropic.com/institute/recursive-self-improvement>
- 20 Anthropic Engineering, building a C compiler with agent teams. <https://www.anthropic.com/engineering/building-c-compiler>
- 21 Veracode, 2026 GenAI Code Security Report, 28.07.2026. <https://www.veracode.com/blog/2026-genai-code-security-report-ai-risk/>
- 22 DORA, 2026 report on the ROI of AI-assisted development, 11.05.2026, as summarized by InfoQ. <https://www.infoq.com/news/2026/05/dora-roi-ai-assisted-dev-report/>
- 23 Anthropic, Agent Skills overview (progressive disclosure). <https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview>

-
- 24 Karpathy, LLM wiki, April 2026. <https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f>
 - 25 OpenSSF Scorecard. <https://scorecard.dev>
 - 26 Shipwright. <https://github.com/svenroth-ai/shipwright>
 - 27 Codextender. <https://github.com/svenroth-ai/codextender>
 - 28 Willison, definition of an agent, 18.09.2025. <https://simonwillison.net/2025/Sep/18/agents/>
 - 29 Osmani, agent-skills. <https://github.com/addyosmani/agent-skills>
 - 30 Vincent, Superpowers. <https://github.com/obra/superpowers>
 - 31 Pocock, skills. <https://github.com/mattpocock/skills>
 - 32 GitHub, spec-kit. <https://github.com/github/spec-kit>
 - 33 Replit. <https://replit.com>

WHITEPAPER / OCTOBER 2026

Driving the AI SDLC with the Shipwright AI Code Factory.

CODE

github.com/svenroth-ai/shipwright

PROJECT PAGE

svenroth.ai/shipwright

INSTALL

```
npm install @svenroth-ai/shipwright@latest
```

Ship right, **not just fast.**

SVENROTH.AI