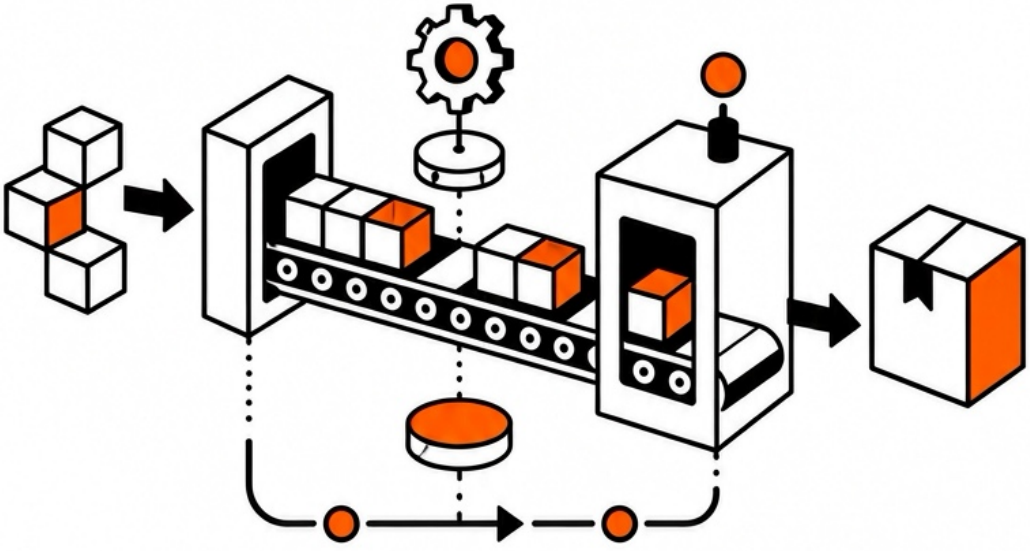
Principles of the Software

FACTORY MODEL



Javier Rayón

learnaipress.com

Contents

1	The New Factory Model of Old	2
2	Four Layers, Four Questions	7
3	The Loop: Goal and Stop Condition	10
4	The Harness: Guides, Sensors and Limits	13
5	The Graph: Precedence, Branches and Resumption	16
6	The Spec: Judgment Written Down Before Building	19
7	The Complete Factory Chain	22
8	Diagnosis and Reading Order	26

Principles of the Software Factory Model

Loop, harness, graph and spec-driven: the four layers of AI engineering

© 2026 Javier Rayón. All rights reserved.

Published by learnaipress.com

Free-distribution guide. You may share the complete, unmodified file. No part of this publication may be reproduced in part, altered, or within another work without the author's prior written permission.

Product and service names mentioned here are trademarks of their respective owners. This guide is not affiliated with or sponsored by Anthropic, OpenAI or any other company mentioned.

The principles collected here summarize the four volumes of the author's AI Engineering Series. Attributions and dates correspond to the sources cited in those volumes, verified as of September 2026. This information is provided without warranty of any kind, and the author is not liable for damages arising from its use.

First edition: September 2026.

SECTION 1

The New Factory Model of Old

Decades ago, car assembly plants solved the problem the software factory now inherits: producing at volume without a person holding each tool. Its parts map one to one onto the four layers of this guide, and that mapping helps explain why each layer exists.

At each station, a robot performs a single operation and repeats it until the part's measurement falls within tolerance. Around it are the grippers and jigs that let it work, the cage that limits how far the arm reaches, and the sensors that measure the part on its way out. The conveyor links the stations in a fixed order, with a door sub-assembly line that feeds in at a fixed point, a quality check where a person signs off, and a rework lane that sends a part back to an earlier station. The car's blueprint and the work order are posted at every station.

Layer	In the plant	What it decides
Loop	The robot at each station	When the part is done
Harness	Grippers, jigs, safety cage and sensors at the station	What the robot has to work with, and how far it reaches
Graph	The conveyor layout: sub-assembly lines, join points, quality check and rework lane	Which station comes after which
Spec	The car's blueprint and the work order	What gets built, and to what tolerances

The person works off the line: designing the layout, adjusting a station's tooling when its robot fails, and reviewing what comes off the end. That split, mechanics for the factory and judgment for the person, holds across all four layers.

The analogy helps to put the four concepts in place, and it fails in two places where software parts ways with car manufacturing. The plant's robot is deterministic and the agent isn't, so the harness checks, which in the plant only confirm quality, are in software how you find out whether the part is any good. And the plant builds thousands of

identical units while the software factory solves a different task every time, so the spec matters on every task, not just when the line is designed.

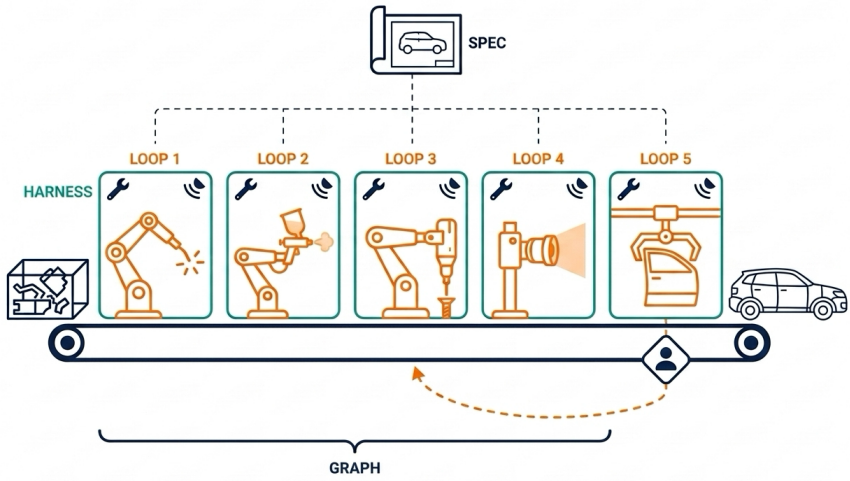


Figure 1.1: The same factory seen as a car plant: each layer has its piece on the line.

1.1 From assistant to factory

A coding agent performs well while someone sits at the keyboard. You request a change, it reads the repository, implements it, runs the tests, and opens the pull request. The problem appears when the volume of work grows faster than the time you can spend in the session. Issues arrive at a pace no conversation absorbs, and the agent, however capable, still needs someone to tell it what to do next.

The first reaction is usually to improve the prompt, add instructions to the project's memory file, and try another model. Each of these measures helps for a few days, which makes the diagnosis look correct, and the limit reappears as soon as the load changes. The work needs organizing differently.

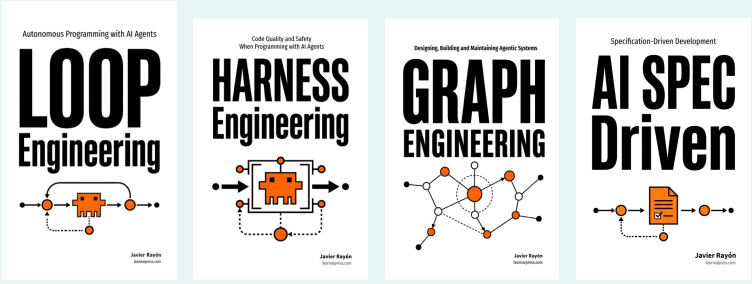
The **assistant** is the way of working almost everyone starts from. You write software and AI helps you write it faster, and you stay in command of every turn. It handles most of the day-to-day work, and stops scaling at the point described at the start of this subsection.

The **factory** works the other way around. A system produces software while you design it, feed it, and supervise it. Work enters on one side, as issues, continuous integration (CI) failures, or requests, moves through a chain that dispatches it, implements it,

verifies it, and records it, and exits the other side as a green pull request waiting for review, with no one driving each turn.

	Assistant	Factory
Who is in command of each turn	The person	The system
What gets designed	The prompt for each task	The chain that solves a class of tasks
When it improves	Within the session, by reading the response	Between runs, by reading what the system did with no one at the keyboard
What limits throughput	Session time	The capacity to review, with judgment, what comes out

The second way is called the **software factory model**, and this guide shortens it to factory model or the factory. The idea appears in *Loop Engineering*, the first volume of the AI Engineering Series, and the other three volumes build it piece by piece.



The AI Engineering Series. Four volumes, one per layer: Loop Engineering (Autonomous Programming with AI Agents), Harness Engineering (Code Quality and Safety When Programming with AI Agents), Graph Engineering (Designing, Building and Maintaining Agentic Systems) and AI Spec-Driven (Spec-Driven Development). Each can be read on its own. Available at learnaipress.com.

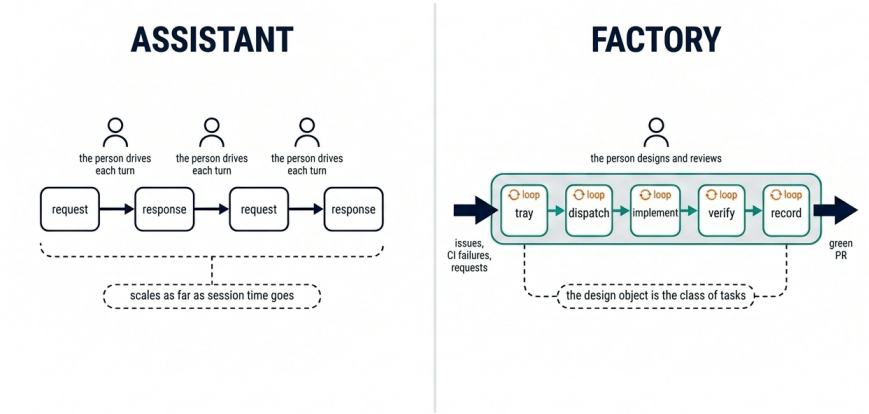


Figure 1.2: The assistant scales as far as session time reaches; the factory changes the object of design to the entire class of tasks.

1.2 What a factory contains

A software factory built with agents is made of many small loops chained together, each with its environment around it, arranged on a map of which step can follow which, and governed by a written description of what to build. The four answer different questions.

- The **loop** sets when the work counts as done, so that "done" can be checked by someone other than the agent.
- The **harness** establishes what the agent has to work with: instructions, tools, permissions, checks, and limits. Every gap it leaves gets filled with a guess.
- The **graph** determines what can happen after each step: what waits for what, what runs in parallel, who signs off, and where it resumes from. Left undeclared, the order comes from whatever the model generates each turn.
- The **spec** defines what to build. Without it, the factory produces, at full speed, something impeccably wrong.

In the factory model, the object of design becomes the entire class of tasks. You build the chain once, say the one that handles the weekly dependency update, and from there each dependency enters that chain without you stepping in for each one. Your work every Monday becomes reading the pull requests the factory produced and clearing the tray of what's left unresolved.

A long chain demands more verification. A factory multiplies whatever enters it, error included. If the dispatch stage classifies something wrong, the stages after it execute the wrong task to spec and deliver a correct pull request for a task that wasn't theirs. That's why designing a factory demands more judgment than writing a good prompt.

The next seven sections cover the four layers, the complete chain, and a procedure for diagnosing which layer is failing in a system of your own.

SECTION 2

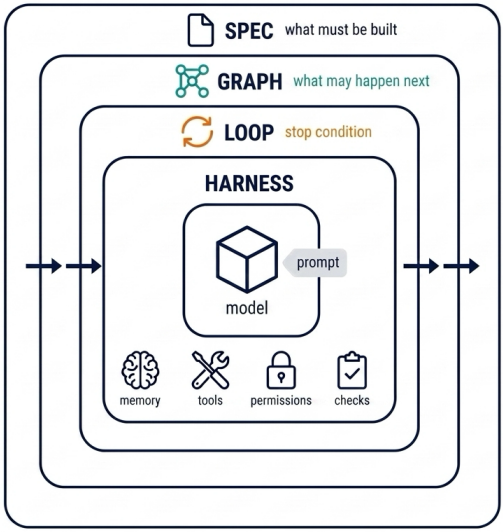
Four Layers, Four Questions

An agentic system can fail in several distinct places, and all of them produce the same symptom: the agent delivers something unusable. The environment you built for it can fail, or the loop whose stop condition should have detected that the work was done, or the order between steps you never declared, or the description of what needed to be built. From the outside the four look alike, so which layer is at fault ends up being a guess.

The four layers have names that have settled in practice, and each answers a question the others don't.

Layer	Question it answers	When it fails, what you see is…	Volume
Loop	When is the work done?	Almost works, but unevenly	*Loop Engineering*
Harness	What does the agent have to work with?	The agent can't operate	*Harness Engineering*
Graph	What can happen after each step?	No one can explain the whole system	*Graph Engineering*
Spec	What needs to be built?	Correct work on the wrong problem	*AI Spec-Driven*

Below the four sits the **prompt**, which determines what the model returns in a given response. It is the smallest layer, and the one with the least return on polishing, because a prompt gets iterated within a session and its improvements disappear when the session closes. The four layers in the table get iterated between runs, and their improvements accumulate.



layer	question it answers
loop	when is it done?
harness	what does the agent have to work with?
graph	what may happen next?
spec	what must be built?

Figure 2.1: The four layers surround the same model; each has its own question.

Prompt engineering and context engineering. Work on that bottom layer has been called prompt engineering and, once it grew to include which files, tools and memory enter the context window, context engineering. Calling it engineering overstates it: it comes down to a set of good practices for writing the request and choosing what goes into the context, and those practices are needed just the same inside any of the four layers. What gets designed, measured and accumulated lives in the layers in the table.

2.1 Three causes of confusion between layers

The first cause is naming. The four layers act on the same files in the same repository, and everything that happens there gets called "the agent". When something goes wrong, the subject of the sentence stays the same, so the failure ends up attributed to the model by elimination.

The second is that the word "loop" names three distinct things. There's the inner cycle of a turn, in which the model generates an action, the tool executes it, and the result returns to the context before the next step. There's the work loop with a stop condition, the one this guide means by loop. And there's the return between steps, the one that sends work back to an earlier step with the diagnosis attached, which belongs to the graph. Someone who says "I have a problem with the loop" could mean any of the three, and the fix differs in each case.

The third is partial improvement. Suppose an agent delivers an incomplete implementation and you react by expanding the repository's instructions. Results improve that week, the team considers it closed, and the failure returns with the next increase in workload. You've spent a week and still don't know what caused it.

2.2 How the layers fit together in one step of the chain

Take the "implement" step of a delivery chain. The graph specifies that it runs once per branch, with three branches at a time, in isolation, and that the work moves on to "verify". Inside the step there's a loop: the agent reads the spec, writes code, runs the tests, fixes the code, and runs the tests again until the stop condition is met. Underneath sits the harness, which provides the isolated working directory, the test command whose output serves as evidence, the scoped credential, and the telemetry. Above it sits the spec: what the agent reads at the start and what the verifier compares the result against at the end.

In other words, the graph runs inside the harness, loops live inside the graph's nodes, and the spec governs what enters the chain and what counts as valid output. When that fit breaks, the symptom shows up at a different level than the cause: an "implement" branch that never finishes blocks the "verify" step, and what you see from outside is a graph stalled at a node that has no problem of its own.

2.3 Where the four terms come from

The terms took hold in practice during 2026, outside the product documentation of coding agents, which explains part of the inconsistency in how they're used.

- **Harness.** Mitchell Hashimoto uses *harness engineering* in February 2026, in "My AI Adoption Journey": the price of every error you discover is the design work that stops it from happening again. On February 11, OpenAI publishes "Harness engineering: leveraging Codex in an agent-first world," by Ryan Lopopolo, with the diagnosis that gives the layer its name: the agent was capable and the environment was poorly specified.
- **Loop.** Addy Osmani pins down *loop engineering* on June 7, 2026, from a line by Boris Cherny, Claude Code lead at Anthropic, about having stopped passing prompts to write the loops that pass them.
- **Graph.** *Graph engineering* appears in mid-July 2026, and LangChain picks it up in its July 22 retrospective, where it describes the loop as a simple version of the graph.
- **Spec-driven.** Spec-driven development has tools of its own, such as GitHub's Spec Kit and AWS's Kiro, and it takes hold as an alternative to *vibe coding*.

The AI Engineering Series arranges the four inside one framework and gives each of them its own volume.

SECTION 3

The Loop: Goal and Stop Condition

The agent writes “done” at the end of its response, and the next morning you discover that the test that was supposed to prove it had spent a week disabled. That is a common loop failure, and it has a costlier version: retries with no cap, the system repeating the same error all night, and a bill that has grown without anyone able to say at what point.

A **loop** is a recursive goal with a verifiable stop condition. What sets it apart from a burst of prompts is that it pursues an end until something checkable confirms the end has been reached. Without a measurable stop condition, what you have is drift that consumes tokens with nothing to tell it when to stop.

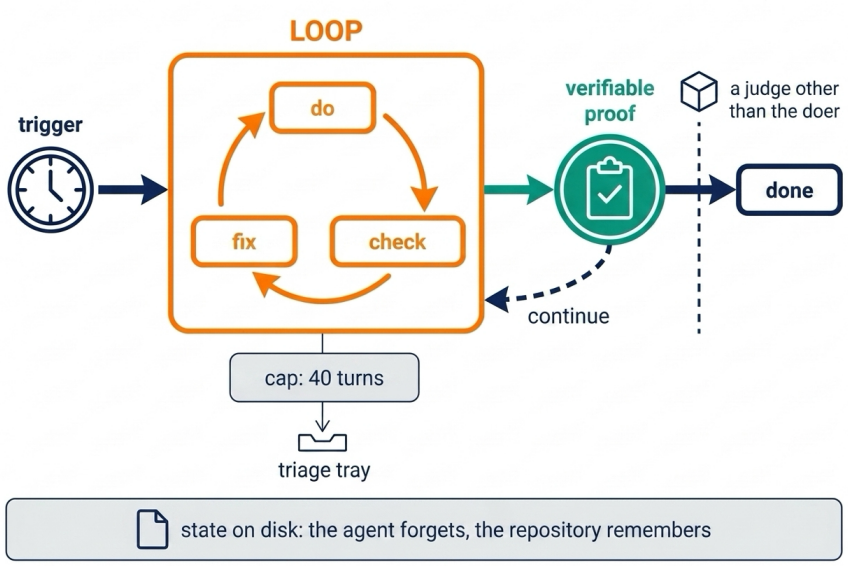


Figure 3.1: A loop: the trigger, the do-and-check cycle, the checkable proof judged by another, the cap, and the state on disk.

3.1 Four decisions made in advance

A well-built loop has four things written down before it starts.

- **What triggers it.** A CI failure, a new issue, a clock that launches it every night, or a person at the terminal. That recurring trigger is what separates a loop from a one-off run.
- **What proves it's done.** A passing test, a schema that validates, an approved review. Something you can check by running it, not the agent's own say-so.
- **Who judges that proof.** Someone other than whoever did the work. The model that wrote the code tends to confirm its own work when it reviews it, so review goes to a different agent, with different instructions and, when possible, a different model.
- **How many rounds it gets, and what happens when they run out.** A number, and a destination for whatever work is still unresolved.

Of the four, the third decision pays off most. Separating the doer from the verifier is what lets you walk away while the loop works, because its "done" comes to mean something.

3.2 State on disk

An agent keeps nothing between one run and the next. That's why loop memory lives outside the conversation, in a state file or a board that tracks what's done and what's pending: state persists in the repository, not in the agent.

The same holds for intent. Every session starts cold, and the agent fills any gap you leave with a guess, so project conventions are written once, in a *skill* read on every run.

3.3 An example: CI failure triage

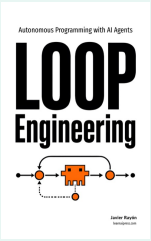
Take a loop that reviews the previous day's CI failures every morning. The naive version hands it a long prompt: review the failures, separate real ones from flaky tests, and propose a fix for the real ones. It half works. Some days the agent can't find the logs; other days it classifies a flaky test as a real failure because the team's flaky-test history isn't available to it.

The loop version leaves that sentence unchanged and writes down the four decisions:

```
# illustrative
trigger: every day at 7:00 AM
stop: "every failure from the day before is classified as flaky
       or real, and the real ones have an open ticket"
judge: a second agent, with verifier instructions
cap: 40 turns; whatever isn't resolved goes to the triage tray
```

The prompt can now be a single line. The stop condition is checkable, because every failure ends up classified with a ticket or in the tray with a reason. And what the loop records about flaky tests accumulates in a file that grows, instead of getting lost with the conversation.

"Done" is still a claim. The loop declaring a task finished doesn't mean it's correctly resolved, however much a verifier backs it up. Confirming that the code works remains your responsibility, and your review capacity is the real limit: no additional agent running in parallel increases the number of pull requests you can read with judgment.



Loop Engineering. Autonomous Programming with AI Agents. First volume of the series: it develops this layer across ten chapters, from the agent's inner cycle to governing unattended loops, and its final chapter presents the factory idea this guide summarizes. Available at learnaipress.com.

SECTION 4

The Harness: Guides, Sensors and Limits

You ask an agent to run the tests. Its context doesn't say that your repository's actual command is `make test`, so it tries `npm test`, doesn't find the script, and reports the suite as broken. The model did something reasonable with what it had in front of it. What was missing was one fact. That's a harness failure, and in practice that kind is the most common.

The **harness** is everything around the model call that you control: the instructions it reads on startup, the tools it can invoke, the permissions that decide what it does alone and what it stops to ask about, the checks that return signal to it, and the limits it can't cross. The model is the same for everyone who rents it. The harness belongs to each team, it's made of deterministic pieces, and it's what accumulates over time.

4.1 Every mistake becomes engineering

Mitchell Hashimoto formulated the discipline's foundational rule: every time the agent makes a mistake, you invest the time in engineering so that mistake cannot happen again. A mistake you only fix is a cost. A mistake you turn into a rule, a linter, a test, or a hook is an investment.

In practice, that changes what a project memory file is: each line exists because a specific mistake motivated it, and a line that prevents nothing is noise competing for context.

> **The rule of two corrections.** If you've corrected the same thing twice in review, the third time it becomes a linter or a memory line. Choose the linter when the rule is mechanically checkable, and the memory line when it calls for judgment.

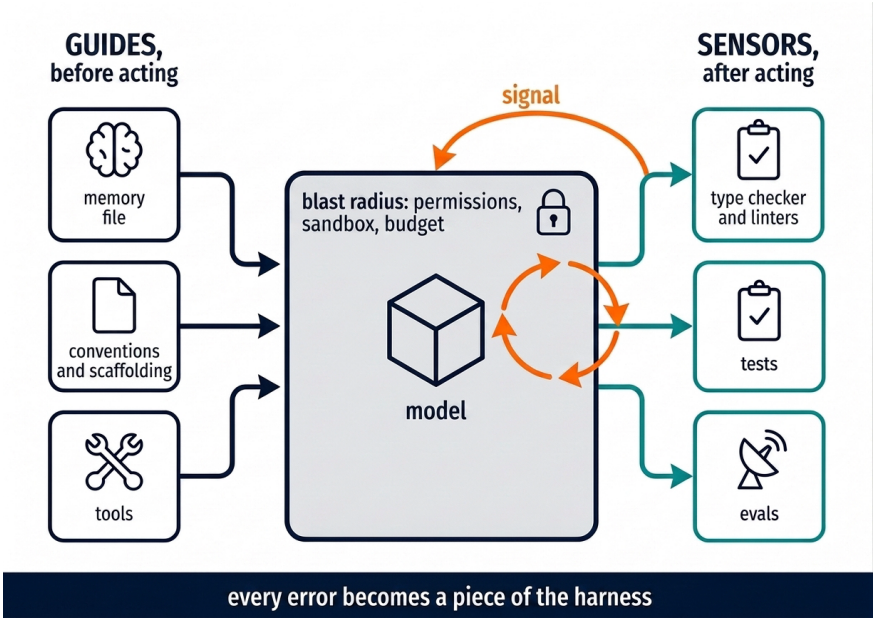


Figure 4.1: Guides orient before acting, sensors return signal after, and the blast radius is set around both.

4.2 Guides and sensors: orienting first, measuring after

Guides orient before generation: architecture documentation, written conventions, the scaffolding that fixes the structure so the agent only fills it in. **Sensors** return signal after: linters, the type checker, tests, architecture validators, model-based reviewers. Guides stop the harness from detecting the same mistake over and over, and sensors are what checks that the written rules are followed.

Sensors are ordered by cost. A type checker responds in seconds and is right about what it checks; a model-based reviewer takes time, costs money, and is right only some of the time. The harness arranges its controls from the cheapest and most deterministic to the most expensive and inferential. And the repository's code determines how much each one catches: strict typing, clear module boundaries, and reproducible builds multiply what each control catches. Preparing the repository is the harness's first step.

When an agent does the programming, tests change role: they become the signal the agent receives on each iteration, determining whether it continues or corrects. That calls for fast suites, failure messages that explain, and coverage that doesn't depend on whoever wrote the code, because the agent that wrote it tends to write tests that confirm it. **Evals** sit at a higher tier than tests: representative tasks with a checkable success criterion, run against every change to the harness, the prompt, or the model. Without that suite, there's no way to know whether a change to the harness, the prompt, or the model made the result worse.

4.3 The blast radius is decided beforehand

All of this assumes the agent will make mistakes. Permissions, process isolation, restricted network access, secrets kept out of reach, and budget caps define how much it can break when it does. They're sized to what the mistake can cost, and they grow with autonomy: work you approve turn by turn can run on your machine, whereas anything that runs unattended overnight goes in a sandbox with restricted network access.

What the agent verifies on its own machine is courtesy. What the CI pipeline verifies is contract, because it runs outside the agent's reach and the agent can't alter its result. The harness is also the only place where what you wrote is separated from what came inside an issue, a dependency's README, or a tool's response, because the model reads a single sequence in which nothing marks where each fragment came from. Every input is an instruction until proven otherwise, and that separation happens outside the model.

4.4 What the harness doesn't decide

The harness externalizes the mechanics of verification. The judgment about what to build, what to accept, and what risk to take stays yours, and a perfect harness in service of absent judgment only produces verified garbage. And a harness that only its author understands will degrade; to prevent that, it is versioned with the code, and each rule carries the mistake that motivated it right beside it.



Harness Engineering. Code Quality and Safety When Programming with AI Agents. Second volume of the series: it develops this layer across fifteen chapters, from the anatomy of the harness to the factory assembled end to end, covering tests, evals, permissions, hooks, CI, observability, and untrusted input along the way. It's the practical continuation of Loop Engineering: where that one describes the idea, this one implements it. Available at learnaipress.com.

SECTION 5

The Graph: Precedence, Branches and Resumption

An issue enters the system with five requirements listed, and the step that implements it opens a pull request that builds, passes CI, and covers two of the five. The agent had the repository, the instructions, and access to the issue tracker, so the harness is ruled out. The stop condition existed. What failed is that the step that wrote the spec recorded the five requirements in a field the transition to the implementation step didn't carry. The agent received the issue title and the two requirements that fit in it, and filled the other three with guesses.

That failure belongs to the **graph**, the layer that decides where the work goes when a step finishes. It can go to the next specialist, go back because review rejected it, stop to wait for a person's approval, or go nowhere because the work is already done. Someone has to make that decision, and if you haven't written it down, the model generates an unvalidated transition on the fly.

5.1 Six decisions that don't fit in the other layers

- **Precedence.** Which step waits on which. Deployment doesn't start until review is approved.
- **Concurrency.** Which pieces of work are independent. Documentation and tests can run at the same time; two agents editing the same module can't.
- **Wait.** Where the branches you opened come back together, and what you do with the one that fails or the one that takes ten times longer than the others.
- **Sign-off.** Which step doesn't run without a person's approval. Usually the one that can't be undone.
- **Return.** Where the work goes back to when verification rejects it, which is almost never the beginning.
- **Resumption.** From which point a run that stopped partway resumes, so it doesn't repeat what was already done.

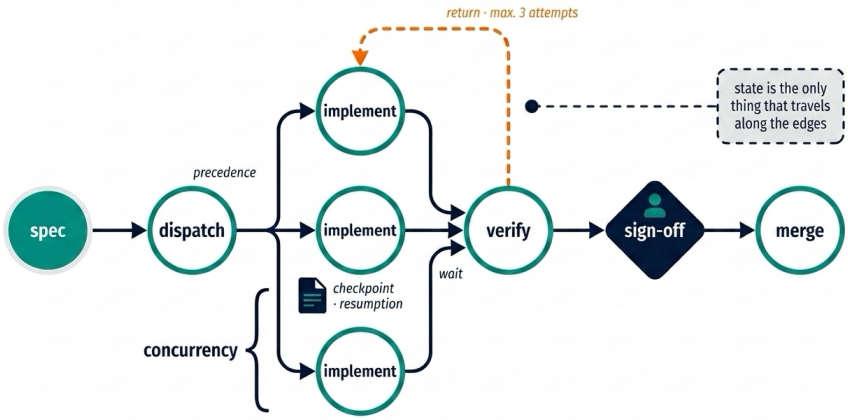


Figure 5.1: A delivery graph with the six decisions marked: precedence, concurrency, wait, sign-off, return and resumption.

The vocabulary for writing these decisions fits in three words. **Nodes** are the steps: a deterministic function, a call to the model, a specialist subagent, or a person. **Edges** are the allowed transitions. **State** is the only thing that travels between them, and it's declared as a contract: which field each node writes and which field it reads.

> **The failure blamed on the model.** Almost every failure in a multi-agent system that looks like "the model lost track" is an edge that didn't carry the data. Before touching a prompt, check which field the step read and which step wrote it.

5.2 When a graph is justified

Real branching, approvals, handoffs between specialists, parallel work, and recovery paths justify a graph. An agent with four tools and a decent stop condition doesn't. A premature graph is easy to spot in the file: six nodes and four state fields for a process a single agent could have solved. A month in, every change forces an edit to the topology, and the team starts working around it to skip the process.

When a graph is justified, the type of each node follows from what the step requires. What can be checked with code goes in a deterministic function; what requires semantic judgment goes in the model; whatever can't be undone goes to a person. Putting a

model where a conditional would have done the job is a common mistake, and one of the most expensive to fix.

5.3 Capped cycles and resumable systems

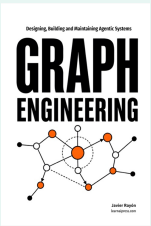
Production graphs have cycles. Retrying, asking for clarification, re-validating, and waiting for a person are legitimate loops, which is why a pure acyclic graph doesn't describe a real system. The rule is that every cycle carries a cap, a counter, and a destination on exhaustion. Of the three, the one most often forgotten is the third: setting a cap of three attempts without specifying where the unresolved work goes leaves the system stalled in a place nobody declared.

Routing, like the loop's stop condition, is decided with evidence. The conditional edge fires on a passing test, a schema that validates, an eval score, or a human sign-off. Routing because "the node says it's done" reproduces the loop's worst habit in the topology.

And a system that doesn't resume is a demo. Processes crash, credentials expire, and approvals take three days. Checkpoints, idempotency in every node with external effects, and resuming from the last valid state are what separate an operable graph from a long script. For the same reason, a human node carries a deadline and an expiry policy.

5.4 What the graph needs from the harness and the loop

The graph runs inside the harness, and needs from it per-node identity, per-branch budget, and correlated traces, because what isn't attributed by node goes ungoverned. From each node it needs a loop with a stop condition, because a step without one blocks everyone who depends on it. Structure replaces oversight, not judgment: what deserves to go through the graph is still your call.



Graph Engineering. Designing, Building and Maintaining Agentic Systems with Claude Code. Third volume of the series: this book develops this layer across twelve chapters, from node boundaries and the state schema to durability, human gates and per-node cost. It includes graphspec, the open source tool that declares, draws and validates the topology from a YAML file. Available at learnaipress.com.

SECTION 6

The Spec: Judgment Written Down Before Building

Thirty screens waved through in a week leave thirty decisions nobody fully made. Each one arrived within two minutes, compiled, and looked close enough to what was asked for, so it got approved and work moved to the next one. By the end of the week, the confirmation email goes out twice because an agent added a retry nobody asked for, and the form asks for a phone number because another agent copied it from an example sitting in its context.

This way of working is called *vibe coding*. Someone holds the intent in their head, releases it piece by piece through the conversation, and approves turn by turn. It works while the task is small and someone is sitting in front of it, and it stops working the moment the agent runs for hours unsupervised, because every gap in judgment it runs into gets filled with a plausible guess. In a factory, that way of working isn't even an option: no one is there to approve every turn.

6.1 Judgment moves to the front

The alternative moves judgment to the front. Before the agent writes anything, the decisions that shape the outcome get written into a document the agent receives as input, and the result is checked against that document afterward. That is spec-driven development, and doing it with an agent involved has a method of its own.

As long as writing the code was the expensive part, judgment got applied by typing, decision by decision, and the upfront document was an approximation to be corrected later. With an agent that writes the code, the upfront document becomes the expensive and decisive part. An agent can generate ten versions of the same screen in the time it takes you to explain which one you wanted. What's scarce is the explanation.

AI automates generation and leaves the decision where it was, because it lacks the context: that this service goes down on Mondays, that the team doesn't know this tech-

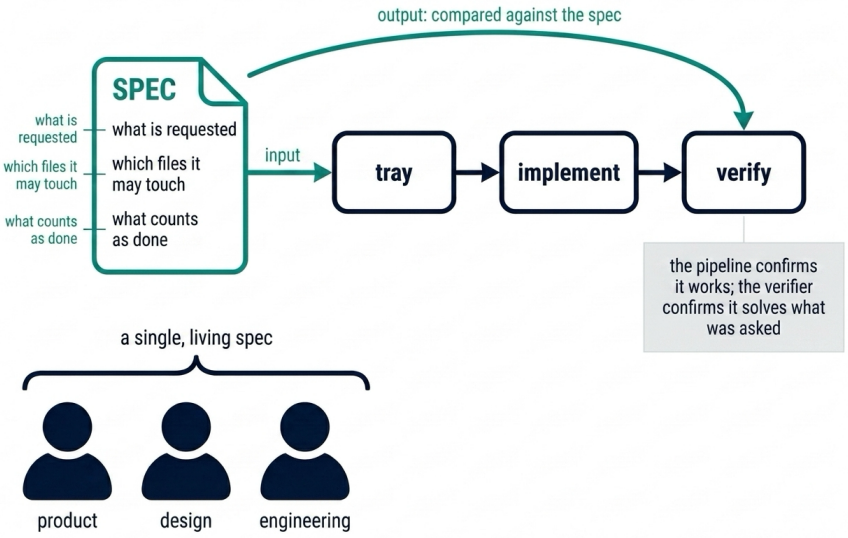


Figure 6.1: The spec enters the tray with the task and reappears on the way out, when the verifier checks the result against it.

nology well, that this user cares more about speed than features. That information lives in your head and in the team's conversations, and when it's missing, the model fills the gap with an output as fluent as the right one, and hard to tell apart from it without the spec.

6.2 Three properties of a spec for agents

- **It is single.** Product, design, and engineering judgment coexist in one artifact, and their contradictions get resolved before reaching the agent, instead of being split across three documents that force the agent to choose.
- **It is executable by an agent.** It's written with the level of precision needed by someone who won't ask questions. Every decision that shapes the outcome is in it, and whatever is left open is left open on purpose.
- **It is living.** It changes when the product changes, with a record of what was decided and why. A spec that still describes the first day's version governs nothing by the third week.

Keeping that up by hand isn't viable. The heavy part, which consists of drafting, auditing gaps, detecting contradictions, and propagating a change through the whole document, is done by the AI itself, with prompts and rules.

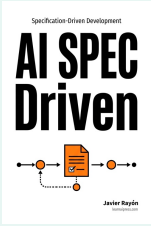
6.3 The spec inside the factory

In the factory chain, the spec shows up in two places. On the way in, because every task that enters the tray needs a minimal spec.

> **The minimal spec for a task.** What's being asked, which files it's reasonable to touch, and what success criterion the verifier will check later. A task missing the third item is a half-formed idea that someone has to make concrete before putting it into the tray.

And on the way out, because the pipeline confirms the code works according to its own sensors, and that's different from confirming it solves the task the spec asked for. If the task asked for date validation without adding new dependencies, the pipeline confirms the tests pass. The verifier checks that the solution didn't pull in an external library and that the pull request didn't touch any module the spec didn't mention.

This layer becomes more important with every version of the model, because the better the model executes, the more expensive it is to have asked it for the wrong thing.



AI Spec-Driven. Spec-Driven Development for Engineers, Product Managers and Product Designers. Fourth volume of the series: it develops this layer across fifteen chapters, covering the anatomy of a living spec, how to start one from scratch, how the three kinds of judgment merge into it, how to audit it until it's complete and consistent, how to keep it up to date, and how to direct and review the agent from it. The book is prompts and rules, with no code in it. Available at learnaipress.com.

SECTION 7

The Complete Factory Chain

You can have all four layers built and still not dare leave the system alone for eight hours, because nobody has connected them yet. This section assembles the complete chain and marks which layer holds up each link. It is the same chain that *Loop Engineering* describes as an idea and that *Harness Engineering* builds piece by piece in its final chapter.

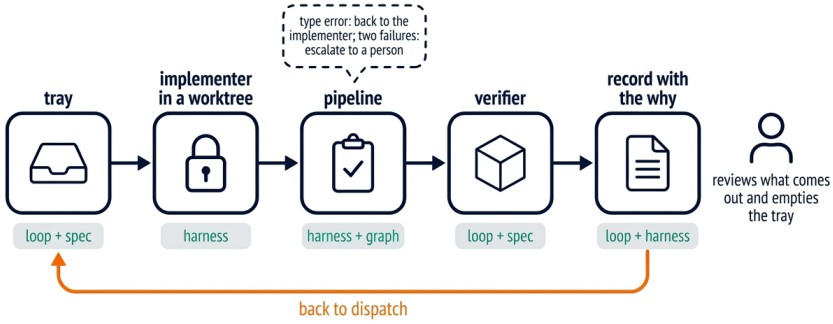


Figure 7.1: The complete chain: dispatch, isolated implementation, pipeline, verification against the spec, and record, with the layer responsible for each link.

7.1 A tray for dispatching work

Without a mechanism that dispatches work, every new task depends on someone remembering to launch it, and that is no longer a factory. The tray is a prioritized backlog

where each entry carries, besides the description, a criterion for what counts as done. A loop walks it, pulls the next task that meets its entry conditions, and assigns it to an agent that implements it. Each entry needs the minimal spec from the previous section, and that entry requirement is what lets the rest of the chain run without anyone watching. Responsible layer: the loop dispatches; the spec sets what enters.

7.2 An implementer in an isolated working directory

The first agent in the chain works in its own working directory, isolated according to the permissions and sandbox profile that fits an unattended task. That isolation is what limits the damage when the agent fails. Inside, the implementer has the harness around it: the repository set up to be governed, the project memory file with the conventions the work always needs, and the tests that return signal on every iteration. In practice, an implementer working without that environment produces a diff that looks much like one from an implementer that has it. The difference shows up three steps later, when that diff goes through controls that ignore explanations. Responsible layer: the harness.

7.3 The pipeline as authority

The implementer finishes and opens a pull request. What it ran in its own directory gave an early diagnosis; the real contract is imposed by the pipeline, where the same sensors run again with authority, followed by the expensive controls that only make sense after integration.

The failure policy determines what happens when something doesn't pass. A typecheck or linter failure returns to the implementer with the full log and no human involved, because it's the kind of error the agent fixes with the signal it already has. A failure that repeats after two attempts, or one in a control that requires judgment, escalates to a person, because retrying blindly burns budget without improving the result. Responsible layer: the harness sets the controls; the graph writes the return, the cap, and the escalation.

7.4 A verifier against the spec

The agent that implemented a task tends to approve its own result: it leaves the same gaps in the review that it left in the code. That's why the factory adds a layer a normal human workflow doesn't need: a second agent that checks the result against the original spec, with a narrower permission scope, because it only needs to read the diff, run the cheap sensors, and compare.

Something must differ between implementer and verifier, even if it is only the system prompt or the model version. If the two share model and prompt, they also share the

same systematic failures. It's enough for the verifier to receive the task from the spec, without the implementer's intermediate output in its context. When it rejects a result, the task returns to the tray with the reason noted. Responsible layer: the loop separates the doer from the verifier; the spec is the standard.

7.5 A record with the why

If nobody records why each decision was made, the next task repeats the previous one's diagnosis. Before the tray selects the next task, the result gets written to a shared place: what was done, what alternative was rejected and why, what obstacle stopped progress if there was one. That same entry feeds the telemetry: task, model, turns, tokens, cost, pipeline result, verifier verdict, and the code's final destination. With that record built up over time, the factory starts to show trends, such as cost per merged pull request or which type of task generates the most rejections. Responsible layer: the loop keeps the state on disk; the harness measures.

That's the complete chain. Dispatch, implementation with a harness, verification by pipeline, verification by spec, a record with the why, and back to dispatch with what was learned already noted down.

7.6 Three rules for any factory

The factory produces more than one person can review. That's both its purpose and its danger, and three rules follow from it.

- **What you can't verify, don't automate.** Before adding a stage, ask who will confirm that its "done" is true, and how. A loop whose output nobody knows how to judge is debt that arrives faster.
- **The factory records the why as well as the what.** A pull request without the reasoning behind it is opaque. That note is what makes the factory's output reviewable.
- **You keep the judgment and hand off the mechanics.** What you hand to the factory is the repetitive, verifiable work. You keep the decision of what to build, what trade-off to accept, and what risk not to take.

> **Same factory, two outcomes.** Two engineers can set up the same factory and get opposite results. One uses it to move faster on a subject they already master: they read every pull request, understand every change, and the factory saves them the typing while they keep the judgment. The other uses it to skip that subject, and accepts whatever comes out as long as it passed verification. The factory produces equally well for both. Over the medium term, one accumulates a system and an understanding of it, and the other accumulates code they don't understand.

The OpenAI team behind the “Harness engineering” post cited in section 2 built a product of around one million lines in about five months without writing them by hand, and it didn’t have a different model from yours. It had a better harness, built by turning every early failure into a control that catches it the next time, and a way of working in which the engineer designed the environment, specified the intent, and built the feedback loops. That is the factory.

SECTION 8

Diagnosis and Reading Order

To diagnose a failure, answer four questions, always in the same order. The order matters as much as the questions. Start at the bottom, with the harness, because a harness failure easily disguises itself as a graph failure: a tool missing from a step looks the same as a badly drawn branch. The opposite mistake is rare. Start at the top, and you end up redesigning the topology to paper over an outdated configuration file.

Order	Question	If the answer is no…
1	Did the agent have everything it needed to do the job well?	It's harness. Build the missing piece and stop here.
2	Was there mechanical proof that the work was done, judged by someone else?	It's loop. Write the stop condition and separate the judge.
3	Was it written down which step could follow which, and what travels between them?	It's graph. Declare the topology and check it.
4	Was it written down what had to be built and what counts as done?	It's spec. Pin down the task before putting it back in the tray.

One layer at a time. When two layers are involved, which happens often, fix the lower one first and measure again before touching the other. Change both at once, and the system improves while you lose track of which change caused it. To confirm you got it right, trigger the failure again and check whether the fix prevents it: these systems carry enough variability that any change can look like it works for a few hours.

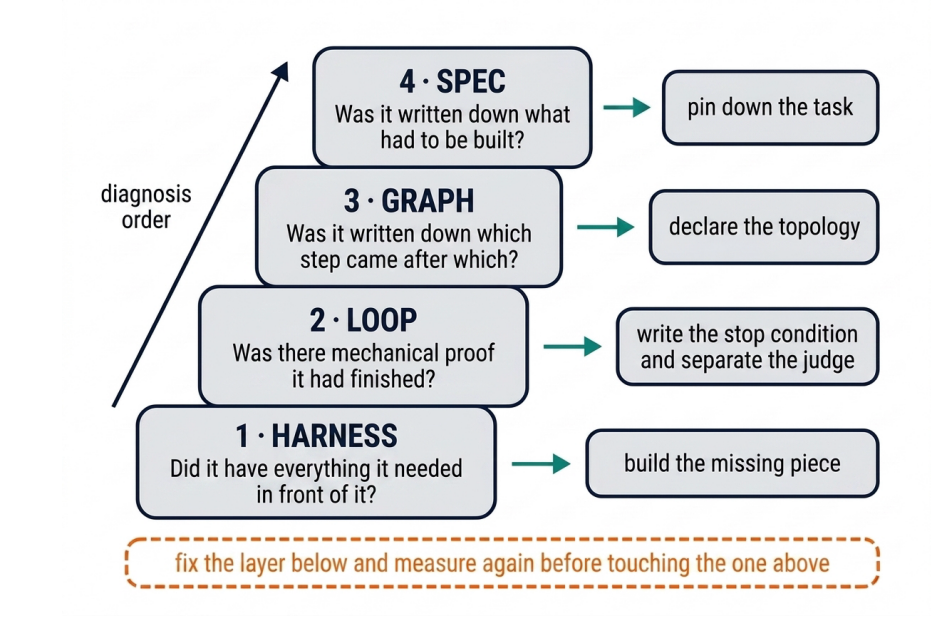


Figure 8.1: The four questions are worked through bottom to top, and each one leads to a different action.

The default diagnosis, “the model got distracted,” is the one that costs the most time, because it sends someone off to tune prompts and try different models for weeks to fix one edge or one configuration line. Before you accept it, run the four questions in order.

8.1 Twelve checks before you leave the system unattended

Answer each line with a check run on the spot, and go through them with someone who didn’t build the system.

Harness.

- The project memory file describes the project as it is, and you’ve confirmed it by opening it.
- Each step has its tools and permissions scoped to its mandate.
- Parallel steps work in isolation, each with its own directory and branch.
- The checks that authorize transitions return a result a machine can read.

Loop.

- Each step has a stop condition that comes from running something.
- The judge is neither the agent nor the model that produced the work.
- Every cycle carries a cap, a counter, and a destination on exhaustion.
- Whatever needs to survive between steps lives on disk, not in a conversation.

Graph and spec.

- Every step has a path to an end state the team recognizes.
- State fields have a declared owner and declared readers.
- The human step carries a deadline and an expiry policy.
- Each task in the tray states what's being asked, what it can touch, and what counts as done.

8.2 Where to start

If the system doesn't exist yet, the build order matches the diagnosis order. Harness first, because without it no loop has anything to work with. Then the loop, with its stop condition and its separate verifier. The spec enters as soon as there's a tray, because it's what decides what deserves a place in it. And the graph comes last, only once real branching, approvals, or parallel work show up to justify it.

8.3 What each book in the series covers

The four volumes of the AI Engineering Series develop each layer with the practice this guide has no room for, and each reads on its own, in whatever order your system calls for.

Loop Engineering builds the loop that runs without you. It's the entry point if you're still holding the tool turn by turn.

Harness Engineering builds the environment that makes what that loop produces reliable. It's the volume for anyone who already has loops running and doesn't trust what comes out of them.

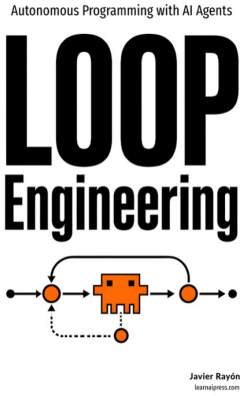
Graph Engineering dispatches work across nodes once a single task stops being enough. It's aimed at those who coordinate several agents, at the point where no one can explain the whole system anymore.

AI Spec-Driven covers the top level, the one that decides what needs to be built. It rewards anyone who already produces fast and wants to stop producing the wrong thing.

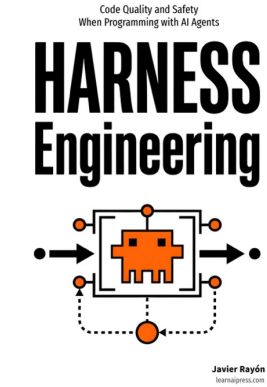
The four are available at learnaipress.com, together with the Agentic Engineering Series, which teaches the tool that the examples in this series rely on. None of them is required to apply the diagnosis in this section.

The AI Engineering Series

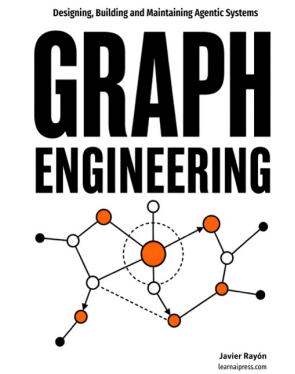
The four volumes of the series, available at learnaipress.com.



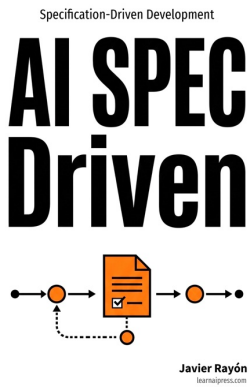
Loop Engineering



Harness Engineering



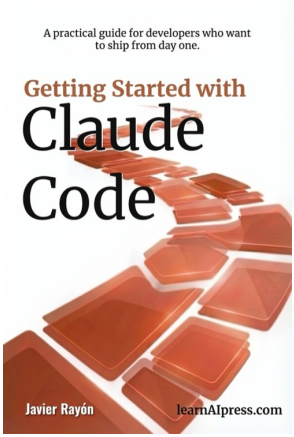
Graph Engineering



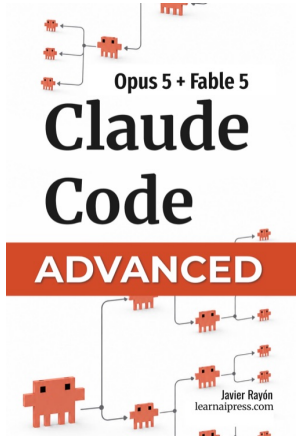
AI Spec-Driven

The Agentic Engineering Series

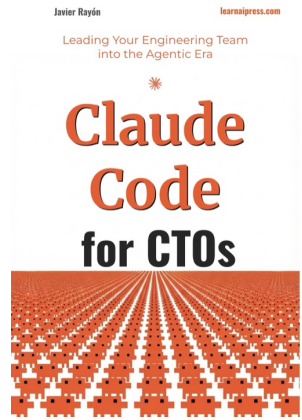
The titles in the series, available at learnaipress.com.



Getting Started with Claude Code



Claude Code Advanced



Claude Code for CTOs



Codex Advanced