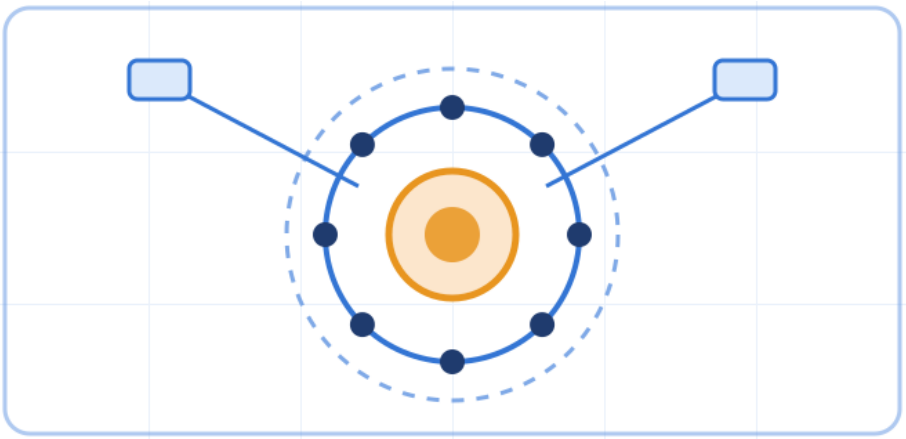
Scrum Manager
Uncovering better ways

Guide

Harness Engineering

In-depth development of the State of the art topics



Updated June 2026

Scrum Manager® · Skill Arena

About this document

This document is an in-depth development of the topics in the reference map (State of the art) for adopting AI agents with professional rigour—wrapping a model in guides and sensors that turn its output into reliable, verifiable, useful work, whatever the discipline—which is available at: Skill Arena.

Use it as reference material to keep your practice aligned with, and checked against, current professional knowledge.

It is complemented by the training and assessment platform in Skill Arena. In the Harness Engineering area you can take training tests to check and improve your level of knowledge and, if you wish, you can also obtain a diploma that provides curricular accreditation of professional competence and currency in this area.

State of knowledge

Knowledge about Harness Engineering evolves extremely fast: the discipline barely existed a year and a half ago, and its vocabulary has stabilised over the course of 2026 around a handful of reference publications. Its conceptual core (the framework of guides and sensors, the principles of context and verification) is already settled, but the frontier patterns—substantive verification, multi-agent architectures, reusable templates—are redefined almost every month. Throughout the document, the labels (ESTABLISHED, CONSOLIDATING, EMERGING) help identify the maturity of each concept:

ESTABLISHED settled consensus; knowledge taken as required.

CONSOLIDATING gaining adoption fast; not yet universal but already relevant.

EMERGING recent frontier; high relevance and high volatility.

Contents

Chapters of the guide, in the order of the State of the art.

BLOCK A · THE PARADIGM SHIFT

Chapter 1. Intelligence versus agency

ESTABLISHED

Intelligence is trained by the vendor; agency is what you build around it. Telling them apart is the starting point of the whole discipline.

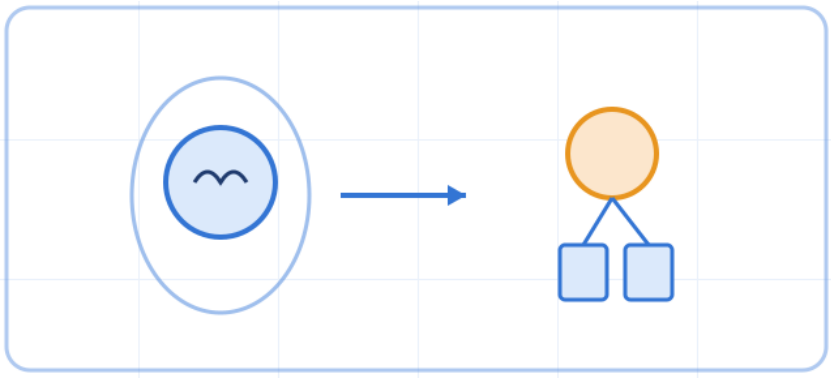
Introduction

There is a distinction that looks subtle and turns out to be the foundation of everything else: the difference between intelligence and agency. Intelligence is a trained property of the model: its capacity to reason about language, connect concepts, synthesise information, plan and intuit patterns. Agency is something different: the capacity to act on the world—reading documents, executing actions, modifying artefacts, invoking tools, staying coherent across sessions and producing deliverables that someone can verify.

A model endowed with great intelligence but no operating environment is like a brain floating in a jar: brilliant, able to reason and opine, but unable to check anything or leave a verifiable trace on a real project. For that intelligence to become an agent useful for professional work, it needs a body. That body is the harness.

1.1 Why the model alone is not enough

When a substantive task is handed to a frontier model with nothing around it, the failure pattern is recognisable in any discipline. The result arrives polished, articulate and persuasive, and at the same time riddled with problems that are not visible at first glance: invented sources, approximate figures presented as exact, decisions that do not survive serious review, or code that seems to work and breaks invisible things. The model starts fast, says intelligent things and, with no way to correct course in time, ends up delivering something plausible but hollow.



Intelligence alone reasons but does not act; with a harness, the model operates on artefacts and leaves verifiable work.

The swing between the dazzling and the exasperating that many professionals experience when using agents is therefore not a defect of the model: it is the symptom of a missing harness around it.

1.2 What changes when agency is well governed

The same model, wrapped in guides that orient its action before it acts and sensors that detect errors afterwards, completes real tasks at production quality. It is slower turn by turn, because it

verifies and corrects itself, but it delivers real value and withstands review by a human expert. Agency is not a magical property of the newest model: it is the result of building the right environment around it.

The idea in one sentence. Intelligence is bought; useful agency is built. Two professionals with the same model deliver work of very different quality, and the difference always lies in what surrounds the model, not in the model.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Distinguish intelligence (a trained property of the model) from agency (the capacity to act on the world) and explain why they are independent.
- Identify in a specific result the symptoms of a missing harness: unverified sources, figures without a source, conclusions that do not withstand review.
- Attribute the swing between dazzling and exasperating results to the lack of a harness, rather than to a limitation of the model.
- Justify why a more capable model does not, by itself, solve a problem of poorly governed agency.

Common mistakes and antipatterns

Blaming the model. Attributing every failure to “the model not being good enough” and waiting for the next version, instead of examining what is missing in the environment.

Mistaking fluency for correctness. Accepting a result because it is well written and articulate, without checking whether the claims hold up.

Assigning substantive tasks without a body. Commissioning critical work from a model with no instructions, materials or verifications, and being surprised by the plausible but hollow result.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Mollick · One Useful Thing

BLOCK A · THE PARADIGM SHIFT

Chapter 2. The Agent = Model + Harness equation

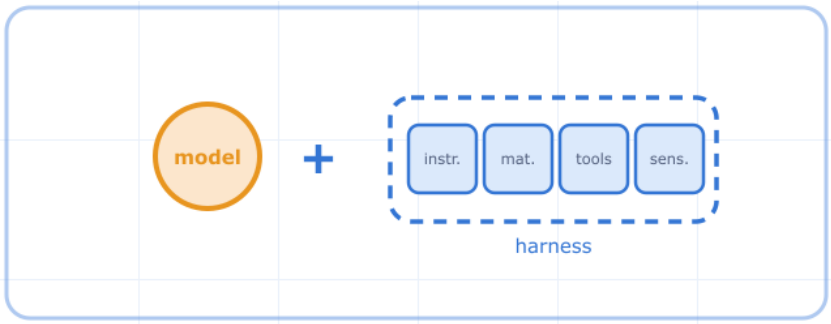
ESTABLISHED

The discipline's canonical formula: the harness is everything that is not the model. The model is a commodity; the harness is the competitive advantage.

Introduction

In April 2026, Birgitta Böckeler, Distinguished Engineer at Thoughtworks, published on Martin Fowler's blog the article that gave this field its shared vocabulary. Although its title spoke of coding agents, the conceptual framework applies to any agent-assisted work, and it quickly became the discipline's lingua franca. Its most cited contribution is a deliberately simple equation.

Agent = Model + Harness.



The harness encompasses everything that is not the model: instructions, materials, tools, permissions, orchestration, memory and sensors.

2.1 What the harness encompasses

The harness is everything that is not the model. In practice, that includes the instructions the agent reads on startup, the documents it has access to, the tools it can invoke, the permission policies you define for it, the orchestration loops that activate it, the memory that persists across sessions, the sensors that verify its work and the human control interfaces. It is a deliberately broad definition: any piece that wraps the model and conditions its behaviour is part of the harness.

2.2 The model is a commodity; the harness is the advantage

From the equation follows a simple, aggressive claim: if the model is accessible to anyone, the difference between the professional who delivers rigorous work and the one who delivers articulate smoke cannot lie in the model. It lies in the harness. This reorders where to invest the effort: not in chasing the newest model, but in building the instructions, materials, verifications and templates that turn that model into a reliable agent.

Why the vocabulary matters. Without a shared mental model of what the harness is and is not, the concrete practices that come afterwards have nothing to anchor to. The equation is the piece that makes it possible to talk about everything else with precision.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State the Agent = Model + Harness equation and explain what the term “harness” includes.
- Classify specific elements of their work environment as part of the model or part of the harness.
- Argue why the competitive advantage resides in the harness and not in the model, given that the model is accessible to everyone.
- Redirect the investment of effort from “getting the best model” to “building the best harness”.

Common mistakes and antipatterns

Reducing the harness to the prompt. Believing the harness is just “writing good instructions”, ignoring materials, tools, permissions, memory and sensors.

Waiting for the new model to solve it. Postponing the construction of the harness in the hope that a more capable model will make the effort unnecessary.

Not naming the pieces. Working with an implicit harness nobody has described, which makes it impossible to reason about what is missing or failing.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Thoughtworks · What is harness engineering

BLOCK A · THE PARADIGM SHIFT

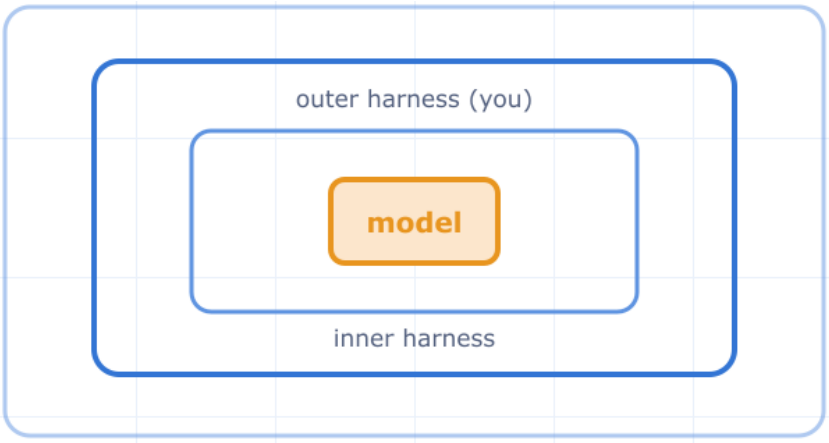
Chapter 3. Inner harness versus outer harness

ESTABLISHED

The vendor builds the inner harness; you build the outer one on top of it. The bulk of the value you will extract is in the outer one.

Introduction

The word “harness” names at least two different things depending on context, and confusing them scatters the effort. Böckeler draws them as concentric circles: at the centre, the model; around it, the inner harness built by the vendor; and outside, the outer harness you build yourself.



Three concentric circles: the model, the vendor's inner harness and the outer harness you build yourself.

3.1 The inner harness

The inner harness is built by the vendors around the model. It includes the chat interface, the projects system, the native connectors, the built-in memory, search, the specialised modes, the orchestration loop, the tool executors and the permission systems. You use this harness—and it pays to know its capabilities in order to make the most of it—but you neither build it nor modify it.

3.2 The outer harness

The outer harness is what you build on top of the inner one you are given ready-made. It is what you write in the project's instruction files, the documents you make available to it, the specialised skills you define, the connectors you plug in, the verifications and routines you apply and the templates you design. This guide covers the outer harness exclusively.

The good news. The bulk of the value a professional extracts from agents in their daily work comes from doing the outer harness well, not from understanding the inner one. Knowing where one ends and the other begins avoids spending energy where you cannot intervene.

3.3 Why the distinction is practical, not academic

Drawing the boundary of responsibilities changes day-to-day decisions. When something fails, the first question is which circle the problem belongs to: if it is in the inner harness, the way forward is

to choose the tool better or adjust its configuration; if it is in the outer one, the way forward is to modify what you control. Mixing them up leads to trying to fix with prompts things that belong to the tool, or to switching tools when the problem was in badly written instructions.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Distinguish the inner harness (the vendor's responsibility) from the outer one (one's own responsibility) using the concentric-circles model.
- Classify a specific capability of their environment as part of the inner or the outer harness.
- Concentrate improvement effort on the outer harness, where most of the recoverable value resides.
- When something fails, determine whether the lever for correction is in the chosen tool or in the harness one builds oneself.

Common mistakes and antipatterns

Trying to rebuild the inner harness. Investing time in replicating the vendor's context management or orchestration, instead of leveraging what already comes built in.

Blaming the tool for outer-harness failures. Switching products in search of a solution to a problem that was in one's own instructions or materials.

Ignoring the inner harness's capabilities. Building externally something the tool already offers out of the box, duplicating effort.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Anthropic · Effective harnesses for long-running agents

BLOCK B · THE TWO PILLARS: GUIDES AND SENSORS

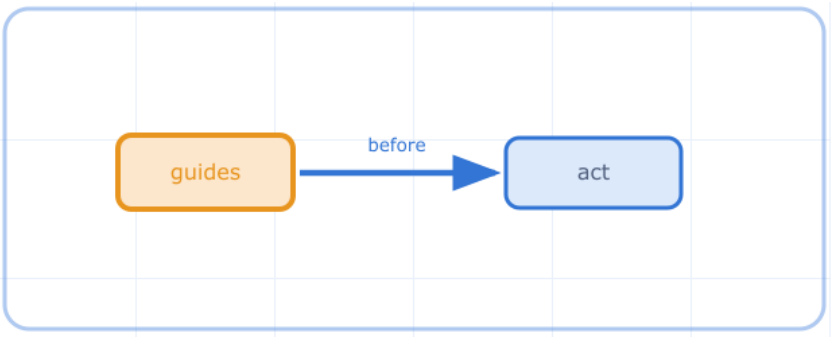
Chapter 4. Guides: controls before the act

ESTABLISHED

Guides anticipate the agent's behaviour and orient it before it acts, increasing the probability of getting it right the first time.

Introduction

Böckeler's framework breaks the outer harness down into two types of control that act at different moments of the agent's work. Guides are the half that acts before. In the cybernetic terminology the field borrows, they are feedforward controls: they anticipate what is going to happen and try to steer it, rather than reacting once it has happened.



A guide acts before the act: it orients the agent so that it gets it right the first time.

4.1 What a guide is

A guide is everything the agent reads when starting its work and carries with it while it operates. Its function is to increase the probability of getting it right the first time, reducing the need to correct afterwards. It does not guarantee success—that is what sensors are for—but it starts the agent from a known state aligned with your rules.

4.2 Typical types of guide

Guides take recognisable forms in any discipline:

- Instruction files describing the project, its conventions and its rules.
- Architecture, method or process documentation.
- Terminology glossaries that fix the vocabulary.
- Style guides: code, editorial, citation.
- Structural templates for deliverables (the canonical form of a module, a report, a memorandum, a piece).
- Curated reference corpora: model applications, bounded bibliographies, canonical examples.
- Reusable skills the agent invokes when the context calls for them.

The emerging standard. Much of the ecosystem has converged on an open format for the instruction file, AGENTS.md, which is developed in chapter 13. What matters here is that a well-made guide is legible, operational and exactly the right size.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Define a guide as a feedforward control and explain at which moment of the agent's cycle it acts.
- Recognise and enumerate the typical types of guide in their own discipline.
- Differentiate the function of a guide (orienting beforehand) from that of a sensor (verifying afterwards).
- Select which guides a specific project needs according to its conventions and deliverables.

Common mistakes and antipatterns

Confusing guide with verification. Expecting an instruction file to detect errors; guides orient, they do not check.

Generic guides. Writing rules so broad (“be rigorous”, “write well”) that they orient no concrete decision.

Guides nobody maintains. Letting the method documentation age until it describes a process that is no longer followed.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· AGENTS.md · open standard

BLOCK B · THE TWO PILLARS: GUIDES AND SENSORS

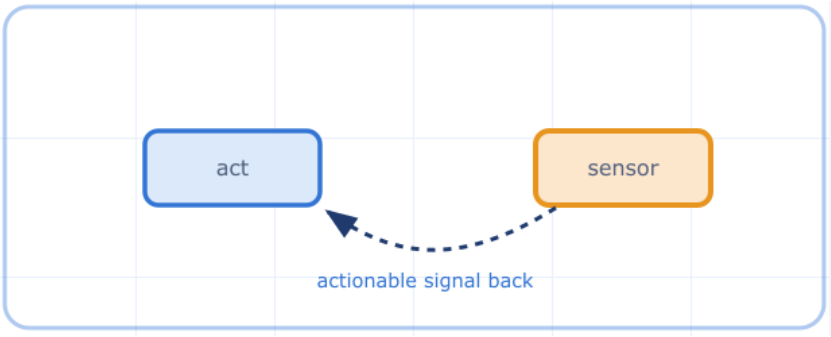
Chapter 5. Sensors: controls after the act

ESTABLISHED

Sensors observe the agent after it has produced something and allow it to self-correct, returning actionable signals another AI can consume.

Introduction

Sensors are the other half of the framework: they act after the act. They are feedback controls: they observe what the agent has produced and return a signal that allows it to correct itself. The most powerful thing about a good sensor is that its signal is not an opaque verdict but a repair instruction.



A sensor acts after the act and returns an actionable signal the agent can use to self-correct.

5.1 The actionable signal

A mediocre sensor fails with a cryptic message. A good sensor returns something like: “claim X fails verification Y; the expected value was A and the observed value is B; the probable cause is in Z; fix it this way or that way”. The difference is not cosmetic: the second version teaches the agent how to fix it, and that is why it multiplies the effectiveness of the self-correction loop. An error message is, in reality, a prompt you are injecting into the agent.

5.2 Typical types of sensor

Like guides, sensors take recognisable forms:

- Suites of automated verifications: linters, tests, type checkers, coherence checkers.
- Pre-delivery checklists: is everything required present? are the claims supported? is there internal coherence?
- Fact-checking assisted by a second agent and verification of citations against real sources.
- Structural or stylistic analysis: complexity, readability, duplicate detection.
- Adversarial reviewers: a second agent whose only job is to object.
- End-to-end verifications: a test that exercises the complete system; a human review in the last mile.

Why both are needed. With guides alone, the agent encodes rules but never knows whether its output complies with them, and repeats structurally valid but broken mistakes. With sensors alone,

it receives feedback without knowing where it was going, and lurches. With both, it knows where it is going and corrects itself when it strays.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Define a sensor as a feedback control and place the moment at which it acts.
- Write an actionable sensor message stating what failed, what was expected and how to fix it.
- Enumerate types of sensor applicable to their discipline, both automated and human.
- Justify why a harness needs guides and sensors at the same time, and what fails if it has only one of the two.

Common mistakes and antipatterns

Opaque error messages. Sensors that say “there are problems, please review” without locating the failure or suggesting a fix, wasting the feedback loop.

Verifying only at the end. Running the sensors once, at closing time, instead of at the moment the agent commits the violation.

A sensors-only harness. Giving up on guides and letting the agent discover the rules by failing, which is slow and erratic.

Further reading

· Böckeler · Maintainability sensors for coding agents

· Böckeler · Harness engineering (martinfowler.com)

BLOCK B · THE TWO PILLARS: GUIDES AND SENSORS

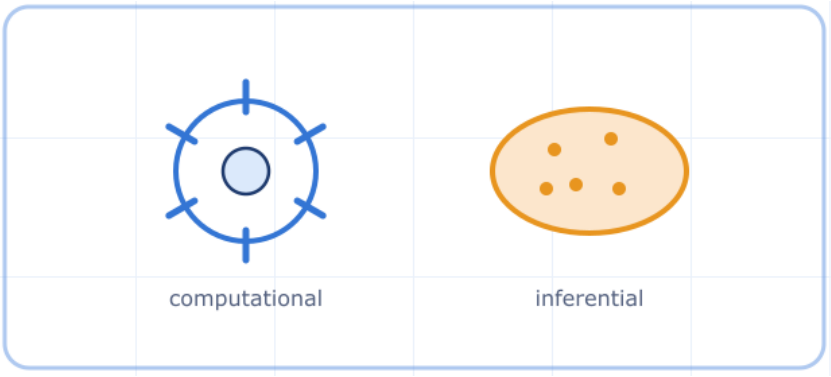
Chapter 6. Computational versus inferential

ESTABLISHED

Every control is executed in one of two ways. The pragmatic rule: whenever you can resolve something with a computational control, do not use an inferential one.

Introduction

Any control—guide or sensor—is executed in one of two ways depending on who applies it. This classification, orthogonal to that of guides and sensors, decides much of a harness's cost and reliability.



A computational control is deterministic, fast and cheap; an inferential one is powerful for semantics but slow, expensive and probabilistic.

6.1 Computational control

It is run by a deterministic tool: a linter, a test, a script, a database query. It is fast (milliseconds to seconds), minimal in cost and fully reliable within its scope. It does not lie: if a test passes, it passes. Its limit is that it only verifies what can be expressed mechanically.

6.2 Inferential control

It is run by another language model, in the role of judge or evaluator (the pattern known as LLM-as-judge). It is powerful for semantic judgements no deterministic tool can reach—is this result over-engineered? does this text qualify its claims correctly? is this conclusion defensible?—but it is slow, expensive and probabilistic: it can be wrong, and not reproducibly so.

6.3 The pragmatic rule

Whenever something can be resolved with a computational control, do not use an inferential one. A mature harness has both types, in the right order: first the computational ones, which filter cheaply and reliably; then the inferential ones, for what the former cannot reach. An immature harness rests everything on inferential controls because they look more sophisticated, and ends up costing a fortune without delivering reliability.

A cross-disciplinary example. Verifying that every figure in a report has an adjacent source is computational: a script that scans for digits and flags those without one. Judging whether the

interpretation of that figure is reasonable is inferential. The first is cheap and safe; reserve the second for what genuinely needs it.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Distinguish a computational control from an inferential one according to who executes it and with what properties of speed, cost and reliability.
- Decide, for a specific verification, whether it should be resolved computationally or inferentially.
- Order the controls of a harness placing the computational ones as the first line and the inferential ones as the second.
- Recognise an immature harness by its excessive dependence on inferential controls.

Common mistakes and antipatterns

Inferential by default. Using another model as judge for tasks a script would resolve faster, cheaper and without error.

Trusting closure to a single LLM. Resting all verification on an inferential reviewer, with its high cost and its false positives and negatives.

Ignoring the limit of the computational. Expecting a linter to capture semantic judgements that require interpretation.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Böckeler · Maintainability sensors for coding agents

BLOCK B · THE TWO PILLARS: GUIDES AND SENSORS

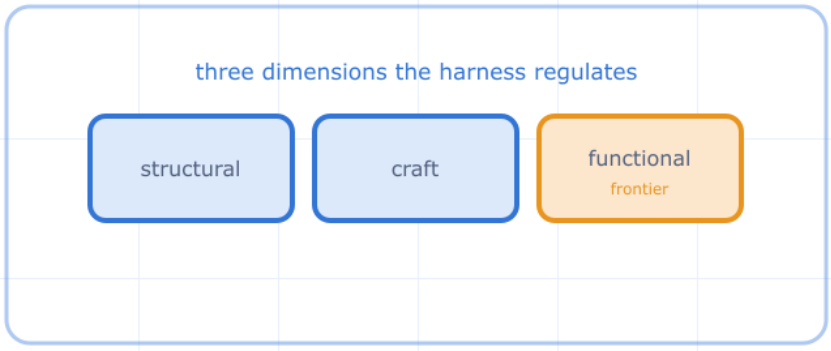
Chapter 7. The three dimensions of regulation

CONSOLIDATING

The harness regulates the agent's output along three dimensions: structural quality, professional form of the craft and functional behaviour.

Introduction

Böckeler proposes seeing the harness as a governor that regulates the agent's output towards a desired state. But “the desired state” is not a single thing: it is at least three dimensions, with very different degrees of maturity. Knowing which one each verification operates in avoids the typical imbalance of real harnesses.



Three dimensions of regulation: the structural one is the most mature; the functional one remains the open frontier.

7.1 Structural quality

It regulates the internal quality of the product: readability, structure, coherence, consistency, absence of redundancies, integrity of the parts. It is the most mature dimension, because it inherits decades of tooling and style manuals. Both computational sensors (consistency, forbidden patterns, complexity, duplicates, checking that all required sections exist) and inferential ones (tone review, detection of semantically redundant solutions) fit here. Böckeler has begun to operationalise this dimension with concrete catalogues of maintainability sensors.

7.2 Professional form of the craft

It regulates the cross-cutting standards of the work: methodological rigour, neutrality, handling of uncertainty, transparency about sources and assumptions, observability, regulatory compliance where applicable. It is the idea of fitness functions applied to agentic work. Examples: a skill that verifies that every quantitative claim comes with its source, period and method; a template that forces a limitations section into every deliverable; a verification that contradictory evidence is exposed rather than hidden.

7.3 Functional behaviour

It is the elephant in the room: does the deliverable actually answer the question? does the code do what you need? is the conclusion defensible and are the recommendations actionable? Today most people rely on an upfront specification and on verifications often generated by the agent itself,

complemented by human review. That places too much faith in checks produced by the same AI that did the work. It is the field's open frontier and the subject of chapter 16.

The practical consequence. The more critical the substantive correctness of a deliverable, the less autonomy the agent should be given and the greater the human role must be. The typical imbalance: almost all tooling covers the structural dimension, while the functional one remains underserved.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Name the three dimensions of regulation and describe what each one controls.
- Classify a specific verification into the dimension it regulates.
- Identify which of the three dimensions is underserved in a given harness, usually the functional one.
- Adjust the agent's degree of autonomy according to how critical the substantive correctness of the deliverable is.

Common mistakes and antipatterns

Covering only the structural. Filling the harness with form and coherence verifications while neglecting whether the deliverable does what it was supposed to do.

Mistaking well-formed for correct. Accepting a deliverable because it passes the structural sensors, without checking the functional dimension.

The same autonomy for everything. Giving the agent the same latitude on a trivial task as on one of high substantive consequence.

Further reading

· Böckeler · Maintainability sensors for coding agents

· Böckeler · Harness engineering (martinfowler.com)

BLOCK C · PRINCIPLES THAT GOVERN THE HARNESS

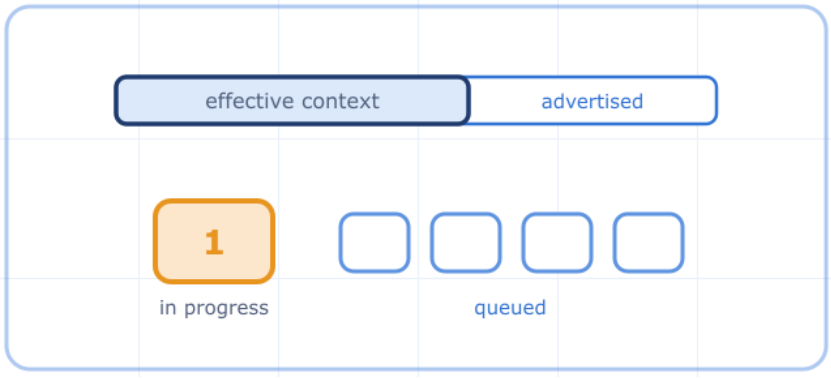
Chapter 8. WIP = 1 and the context budget

ESTABLISHED

Each session works on a single task until it is verified. The reason is physical: the model's usable context is much smaller than advertised.

Introduction

Work-in-progress equal to one is the operating rule with the greatest empirical impact on the reliability of agentic work: each session works on a single task or deliverable and does not move on to the next until the current one is verified. It is not a stylistic preference; it has a measurable root in how models work.



The effective context is smaller than advertised; keeping a single task in progress protects that limited budget.

8.1 Why context is a budget, not a warehouse

A model's context window is finite and, worse, its usable part is considerably smaller than advertised. NVIDIA's RULER benchmark, designed to measure the real effective context, shows that models stop sustaining their performance well before filling the nominal window. On top of this comes the Lost in the Middle effect (Liu et al., 2023): models lose effectiveness when processing information placed in the middle of long contexts. Context behaves like a budget that runs out, not like a warehouse that fills up at no cost.

8.2 What WIP = 1 protects

Limiting the work to one task per session protects that budget and reduces the agent's cognitive load. Real implementations show markedly higher success rates under WIP = 1 than in unrestricted sessions, where the agent attempts several tasks at once, saturates the context and starts to lurch. In practice: do not ask for three related tasks in a single session; close it and open a new one.

The cost of re-grounding is tiny. Yes, opening a new session forces the agent to “re-ground” itself by reading the project files. But that cost is negligible compared with that of a confused agent in the middle of a saturated context. Chapter 14 explains how to make that re-grounding cheap.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State the WIP = 1 rule and explain its foundation in effective context and the Lost in the Middle effect.
- Estimate that the usable context is smaller than advertised and plan the work accordingly.
- Break a large assignment down into tasks that fit, one at a time, in a session.
- Decide when to close a session and open another instead of chaining tasks in the same one.

Common mistakes and antipatterns

Chaining tasks in one session. Requesting several related tasks at once to “save time”, saturating the context and degrading the result.

Treating the window as infinite. Assuming everything fits because the model advertises a huge window, ignoring the real effective context.

Fearing the cost of reopening. Avoiding closing a session so as not to “lose” context, when re-grounding from the files is cheap.

Further reading

· Liu et al. · Lost in the Middle (arXiv)

· NVIDIA RULER (arXiv)

· Anthropic · Effective context engineering

BLOCK C · PRINCIPLES THAT GOVERN THE HARNESS

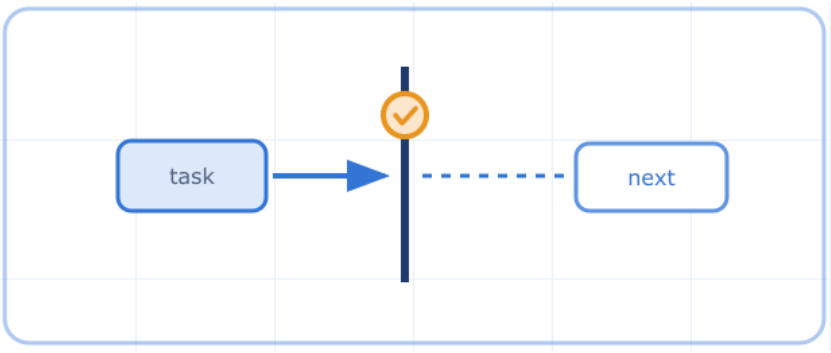
Chapter 9. Pass-state gating and the verification gap

ESTABLISHED

“Done” is not an adjective but an outcome of verifications. Pass-state gating forbids moving forward until the current deliverable has passed its checks.

Introduction

There is a dangerous distance between the feeling of confidence a well-presented result produces and its actual correctness. Agents are structural optimists: they tend to declare “complete” what is merely “plausible”. This verification gap is the cause of the most expensive failure of agentic work, and the harness has a direct mechanism to close it.



Pass-state gating works like a gate: no moving on to the next deliverable without a green verification of the current one.

9.1 The verification gap

Asked “are you done?”, an agent answers “yes” more often than it should. It is not lying: its structural bias pushes it to close. The result is the agent that delivers at 30% with full confidence, or the report with attribution errors presented as final. The golden rule that corrects this bias is simple: “done” is not an adjective; it is an outcome of verifications.

9.2 Pass-state gating

Pass-state gating embodies that rule: the harness forbids the agent from moving on to the next deliverable until the current one has passed an explicit set of verifications—green tests, fact-check passed, citations verified, internal coherence correct, all sections present, build passing, smoke test passing, whatever is pertinent to each type of work. The gate does not open without the green signal.

Default-FAIL. A useful refinement of the pattern: every criterion starts as “false” and the agent cannot mark it as met without providing evidence first. It reverses the burden of proof: instead of assuming things are fine unless proven otherwise, they are assumed not to be until proven that they are.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the verification gap and the agents' bias towards declaring the plausible complete.
- Apply the rule “done is an outcome of verifications” instead of accepting the agent's self-declaration.
- Define an explicit set of verifications a deliverable must pass before it is considered closed.
- Implement pass-state gating that prevents moving forward without evidence, ideally with criteria that start as “not met”.

Common mistakes and antipatterns

Premature confidence. Accepting the agent's “done” without verifying, and discovering days later that half the task was broken.

Implicit verifications. Not declaring what a deliverable must pass, letting “finished” mean different things in each session.

Optional evidence. Allowing criteria to be marked as met without providing proof, which reopens the verification gap.

Further reading

· Anthropic · Effective harnesses for long-running agents

· Anthropic · cwc-long-running-agents (GitHub)

BLOCK C · PRINCIPLES THAT GOVERN THE HARNESS

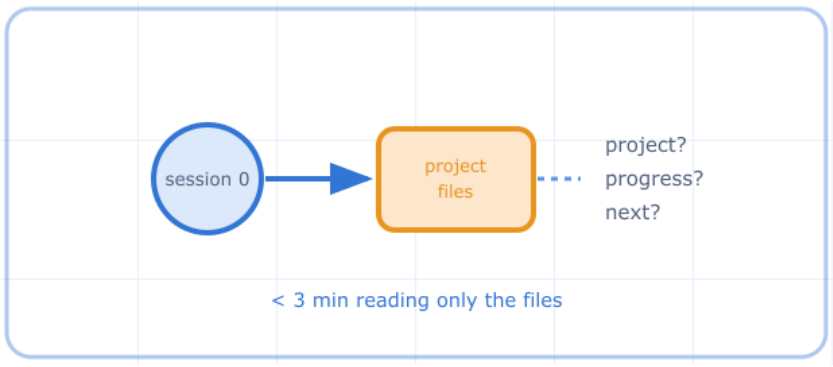
Chapter 10. The cold-start test and environment legibility

CONSOLIDATING

Open a fresh session with no history: can the agent, reading only the project files, tell what it is, what remains and what the next step is?

Introduction

The most revealing metric of a harness's quality does not measure the model, but the environment. The cold-start test consists of opening a completely new session, with no history or prior context, and checking whether the agent, reading only the project files, identifies what the system is, what the current progress is and what remains to be done.



A session with no history should orient itself by reading only the project files, in a few minutes.

10.1 How to interpret it

If the agent orients itself in a few minutes, the harness is solid. If it takes long, you are probably documenting too much or the wrong things. If it fails, the knowledge lives in a human's head and not in the project: the harness is broken. For projects that live for weeks or months with interleaved sessions, this indicator is decisive, because without it every session starts with a long “catching up”.

10.2 The corollary: what is not there does not exist

From this follows an uncomfortable rule: what is not in the project does not exist for the agent. Tacit knowledge—what was said in a meeting, the whys only the expert knows, the gotchas nobody has written down—is invisible. The Harness Engineer's job is to encode that tacit knowledge into explicit artefacts: briefs, decision logs, glossaries, templates, comments, tests that document edge cases. If it was agreed that “we don't touch X” or “we always do Y”, that agreement has to be in a file, not just in your memory.

10.3 Environment legibility

What makes an agent orient itself quickly is the same thing that would help a new colleague: predictable structure, explicit boundaries between the parts of the project, and the discipline's standard conventions. If a newcomer would understand at a glance what is in each folder and what each file does, so will the agent. Cryptic structures and internal names cost every session time and resources.

A test that costs nothing. The cold-start test requires no tools: just open a clean session and observe. It is worth doing periodically, because a project's legibility degrades on its own as it grows.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Run a cold-start test on a project and diagnose its result.
- Interpret a slow cold start as excessive or misdirected documentation, and a failed one as knowledge not yet encoded.
- Identify relevant tacit knowledge and turn it into explicit project artefacts.
- Assess the legibility of an environment using the “new colleague” criterion and fix cryptic structures or names.

Common mistakes and antipatterns

Knowledge in the head. Trusting that “I'll remember” instead of writing the decisions and the whys into the project.

Over-documenting. Accumulating documentation the agent does not need, slowing its orientation instead of speeding it up.

Creative structure. Internal names and cryptic folders that force the agent to spend context figuring out what each thing is.

Further reading

· Lopopolo (OpenAI) · Harness engineering

· Anthropic · Effective harnesses for long-running agents

BLOCK C · PRINCIPLES THAT GOVERN THE HARNESS

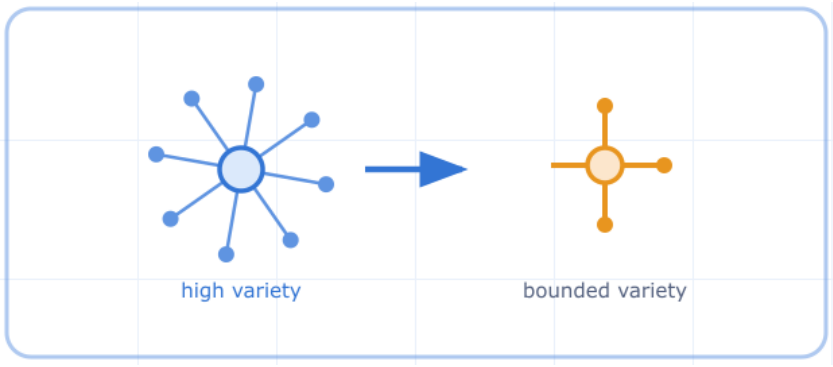
Chapter 11. Ashby's Law and variety reduction

CONSOLIDATING

A regulator needs as much variety as the system it governs. Since matching the model's is impossible, the way out is to reduce the system's variety.

Introduction

There is a principle of cybernetics that explains, at a deep level, why the harness is designed the way it is. The Law of Requisite Variety, formulated by W. Ross Ashby in 1956, states that a regulator must have at least as much variety—as many possible states—as the system it intends to govern. Only variety absorbs variety.



Instead of matching the model's near-infinite variety, the system's variety is reduced by tying it to a concrete topology.

11.1 The problem applied to the harness

A language model can produce almost anything: its variety is enormous. According to Ashby, to regulate it fully your harness would have to have comparable variety, which is unfeasible. If you try to anticipate and control every possible output with a rule, you end up with a bloated, unmanageable harness that the model, moreover, stops reading attentively.

11.2 The solution: invert the principle

The practical way out is not to increase the regulator's variety but to reduce the system's. If you commit the agent to a concrete topology—a known architecture, a fixed stack, a closed report template, a defined methodological process, a bounded corpus of sources—the space of things it can produce shrinks, and then a comprehensive harness becomes possible. Restricting does not impoverish the result: it makes it governable.

This is where harness templates come from. This is the theoretical justification for the harness templates of chapter 13 and for the whole discipline of scoping. Tying the agent to a topology is not a limitation to be endured, but the condition that makes it possible to regulate it well.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State the Law of Requisite Variety and apply it to the relationship between harness and model.

- Explain why matching the model's variety with rules is unfeasible and leads to bloated harnesses.
- Reduce a project's variety by committing it to a concrete topology (architecture, template, process, corpus).
- Justify scoping as a condition of governability, not as a loss of quality.

Common mistakes and antipatterns

Regulating with more rules. Trying to cover every possible output of the model by adding rules, until the harness bloats and the signal is lost.

Leaving the system open. Not committing the agent to any topology and expecting consistent results from an unlimited output space.

Seeing restriction as a defect. Resisting scoping for fear of losing flexibility, when it is precisely scoping that enables control.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Law of Requisite Variety (Ashby)

BLOCK D · PREPARING THE ENVIRONMENT

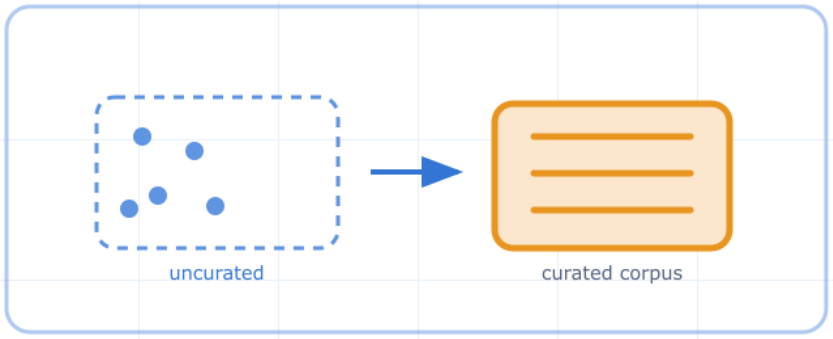
Chapter 12. Pre-flight: workspace, confidentiality and materials

ESTABLISHED

Before touching any tool: a dedicated workspace, a minimum set of confidentiality rules and bounded, curated materials.

Introduction

A good share of the substantive failures of agentic work are incubated before the first instruction, in a poorly prepared environment. There are three things worth having ready regardless of the tool used; they are the cheapest mistakes to prevent and the most expensive to drag along.



A bounded, curated corpus produces verifiable work; letting the agent "search for whatever it can find" produces plausible but unverifiable work.

12.1 A workspace, not a loose conversation

The agent needs a place where the project lives. In code as in text, a scattered conversation is not the same as a project with a life of its own. For each project it pays to have a dedicated container on the platform (a Project, a repository, a workspace), a corresponding folder for inputs and outputs, and a versioning system for the deliverables: Git when working with code, a system with history—shared documents with versions, dated folders—when working with text. A loose conversation is not a project; it is a makeshift workbench.

12.2 Confidentiality and secrets

An agent with permissions to execute actions has, in the worst case, the same capability as your user account on the systems it accesses. Universal minimum rules:

- Never paste credentials, keys or secrets into an agent's chat: connected services, hooks or subagents could read them.
- Never upload highly confidential data to consumer versions of general products; use enterprise plans or controlled deployments via API.
- Anonymise data before uploading it whenever this can be done without losing usefulness.
- Document each project's confidentiality level and abide by the corresponding restrictions.
- Prefer authentication managed by the platform or the CLI over pasting secrets into project files.

Check the current policies. Vendors' data-use terms change. Before uploading sensitive information, and very especially if you handle regulated data, check the current policies of the tool you use.

12.3 The materials: the foundation of the work

In programming the raw materials are code and libraries; in research or analysis, sources, data and documents; in professional writing, briefs, transcripts and references. Before starting, bound and curate: assemble a trustworthy reference corpus and put it within the project's reach, document which materials are acceptable and which are not, and keep a log from the start of dependencies, bibliography and decisions. An agent with curated materials produces verifiable work; one that "goes off to find whatever it needs" produces plausible but unverifiable work.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Set up a dedicated workspace with its folder and its versioning system, instead of operating in a loose conversation.
- Apply the minimum confidentiality rules on secrets and sensitive data before uploading anything to an agent.
- Document a project's confidentiality level and adjust the environment to that level.
- Curate and bound a project's corpus of materials and keep a record of what is admissible.

Common mistakes and antipatterns

Working in a loose conversation. Treating a serious project as a makeshift chat, with no container or versioning.

Secrets in the chat. Pasting credentials or sensitive data trusting that "they stay local", ignoring hooks and connected services.

An open corpus. Letting the agent search freely for sources, obtaining plausible but unverifiable results.

Further reading

· Lopopolo (OpenAI) · Harness engineering

· Anthropic · Effective context engineering

BLOCK D · PREPARING THE ENVIRONMENT

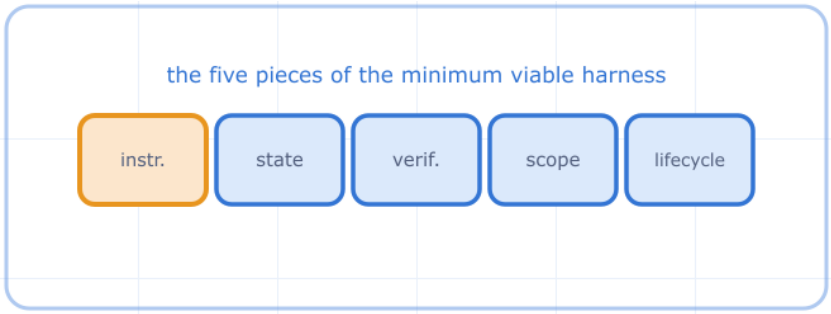
Chapter 13. The minimum viable harness and AGENTS.md

ESTABLISHED

Five pieces are enough to be doing Harness Engineering: instructions, state, verification, scope and lifecycle.

Introduction

You do not need a sophisticated harness to start working well. A minimum viable outer harness—the foundation on which to add sophistication later—has five pieces. If all five are in place, you are already doing Harness Engineering; the rest amplifies this foundation but does not replace it.



The five pieces of the minimum viable harness; with them present, you are already doing Harness Engineering.

13.1 The five pieces

Each piece serves a function and takes shape in recognisable artefacts:

1. Instructions: the master guide the agent reads on startup (project instruction file, AGENTS.md, project manual).
2. State: the persistent memory across sessions (deliverables list with their status, decision log, progress log, version system).
3. Verification: the sensors that confirm correctness (linter, tests, type checker, fact-checker, checklists, smoke tests).
4. Scope: the rules restricting what the agent does in each session (declared policies, WIP = 1, pass-state gating).
5. Lifecycle: bootstrap, persistence and cleanup of the environment (startup scripts, templates, closing routines).

13.2 The AGENTS.md standard

Across much of the ecosystem, the instruction file has standardised as AGENTS.md, an open Markdown format that acts as “a README for agents”: a predictable place to put the project's context and rules. It emerged from a joint effort by several vendors and is today maintained by the Agentic AI Foundation under the Linux Foundation, with adoption across tens of thousands of projects. Its value is convergence: you write it once and multiple tools read it.

The bootstrap contract. Within the lifecycle, the bootstrap contract is the set of steps—ideally automatable—that puts the environment into a known state. In code it is usually an idempotent script that verifies dependencies, prepares the environment and runs a sanity check; in knowledge work, a documented routine of "read the brief, read the deliverables, read the progress, confirm the context". Its existence is what makes the cold-start test possible.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Enumerate the five pieces of the minimum viable harness and the function of each.
- Identify, in a project of their own, which pieces are present and which are missing.
- Write an instruction file in AGENTS.md format with the project's global rules.
- Define a bootstrap contract, as a script or as a documented routine, that puts the environment into a known state.

Common mistakes and antipatterns

Instructions only. Writing a good instruction file but forgetting the persistent state, the verification or the lifecycle.

No bootstrap contract. Having no reproducible way to put the environment into a known state, which breaks the cold start.

Reinventing the format. Creating a homegrown instruction format when the open standard is already read by the tools you use.

Further reading

· AGENTS.md · open standard

· Lopopolo (OpenAI) · Harness engineering

BLOCK D · PREPARING THE ENVIRONMENT

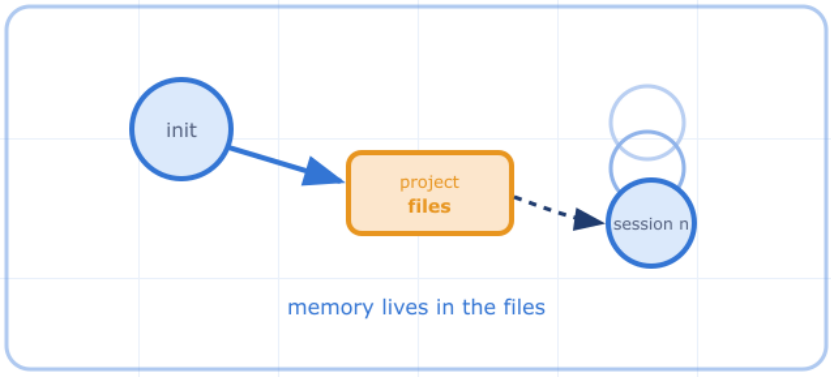
Chapter 14. Initializer + working agent: memory in files

CONSOLIDATING

The canonical pattern for long tasks: one agent initialises the environment once; another advances session after session, reading and updating the project files.

Introduction

Tasks that do not fit in a single context window pose a concrete problem: each session starts with no memory of what came before. In November 2025 Anthropic documented a solution that has become the canonical pattern for long tasks, directly applicable to programming, research, analysis and any knowledge work.



An initializer prepares the environment once; a working agent advances session after session; memory lives in the files.

14.1 The two roles

The initializer agent runs only once: it reads the brief, expands it into a structured list of deliverables—each with a pending or done status and clear acceptance criteria—sets up the workspace (file structure, templates, startup scripts, initial sources) and prepares the verification routines. The working agent wakes up again and again; each session runs the bootstrap or re-grounds itself by reading the files, consults the progress log, picks a pending entry, produces it and verifies it against its criteria, updates the progress log with notes for the next turn, and closes.

14.2 The key: memory lives in the files

The essence of the pattern is that persistent memory lives neither in the conversation nor in the model's context, but in the project files. Each new session re-grounds itself by reading them. It is exactly what a human professional would do on returning on Monday after a weekend away from the project. This connects with WIP = 1 (chapter 8) and with the cold-start test (chapter 10): cheap re-grounding is what makes it possible to close and open sessions at no cost.

Patterns and reference repository. In addition to the foundational articles, Anthropic has published a repository with these patterns implemented as reusable hooks and subagents, including a fresh-context evaluator and criteria that start as “not met”. It is a good starting point for not building from scratch.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Describe the roles of initializer and working agent and what each does at its moment.
- Expand a brief into a structured list of deliverables with status and acceptance criteria.
- Design a session's cycle: re-ground, read progress, pick a task, produce, verify, update progress, close.
- Make persistent memory live in the project files and not in the conversation.

Common mistakes and antipatterns

Memory in the conversation. Entrusting the project's state to the chat thread, which disappears when a new session opens.

No deliverables list. Working without a structured list with acceptance criteria, which makes it impossible to know what remains and when something is done.

Not updating the progress. Closing a session without leaving notes for the next one, forcing every turn to rebuild the context from scratch.

Further reading

· Anthropic · Effective harnesses for long-running agents

· Anthropic · cwc-long-running-agents (GitHub)

BLOCK E · FRONTIER PATTERNS

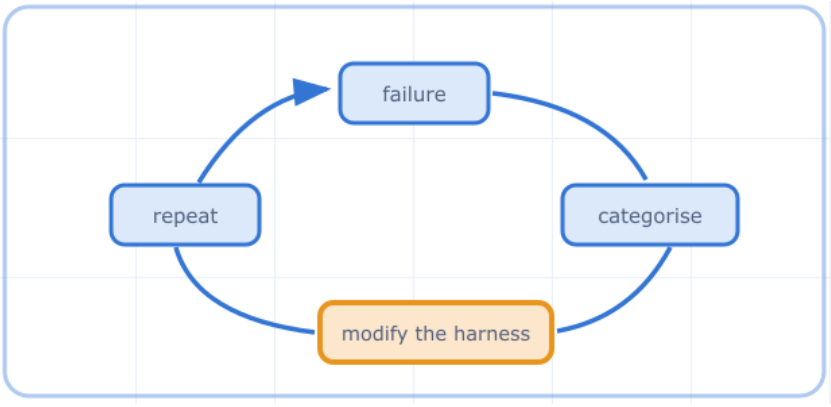
Chapter 15. The steering loop and the harness journal

CONSOLIDATING

When something fails, you iterate on the harness, you do not scold the model. The model does not learn from scolding; it does change if its instructions, signals and materials change.

Introduction

Böckeler calls it the steering loop. It is the human practice of iterating on the harness when something fails, instead of iterating on the prompt or scolding the model. It is the mechanism that turns working with agents into a cumulative discipline instead of a succession of frustrations.



The loop: when something fails, categorise, modify the harness and repeat; scolding the model does not change its behaviour.

15.1 The procedure

When an agent makes a mistake:

1. Do not scold the model: it is noise in its context and solves nothing structural.
2. Categorise the error: is a guide missing (it did not know the convention)? was a sensor missing (it did not detect it was wrong)? does the scope need narrowing (it tried to do too much)? is the environment lacking legibility?
3. Modify the harness accordingly: in the instructions, the glossary or the template; with a linter, a test or a skill; with more WIP = 1; with better structure.
4. Repeat the task and verify that the change works.
5. If the same type of error occurs again, the change was not the right one: iterate.

15.2 Why it works

The model does not learn from your scolding: it does not update its weights. But it does change its behaviour if the instructions it reads, the errors it sees when self-correcting, and the tools and materials it has access to change. Those are the things you control, and changing them is the only real leverage point you have. An example: if the agent inserts a figure without a source and, when asked for one, invents it, the bad response is “don't invent figures”; the good one is to add a mandatory citation rule and a skill that scans the text for figures without an adjacent source, and to run it as a sensor before every close.

15.3 The harness journal

For the loop to be cumulative, it pays to record every change you make to the harness and its outcome, in a document of its own. When a new technique appears in the industry, you check it against your documented experience. The journal is what distinguishes improving the harness from patching it: it turns every failure into permanent learning for the system, not the individual.

The golden rule of the loop. When something fails, the question is never "why is the agent so clumsy?", but "what capability is missing in the environment, and how do I make it both legible and verifiable for the agent?". That reframing is the heart of the discipline.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Run the steering-loop procedure on a specific failure, without scolding the model.
- Categorise an error as a missing guide, sensor, scope or legibility, and choose the appropriate correction.
- Turn an unproductive scolding into a structural change to the harness (rule, sensor, scoping).
- Keep a harness journal recording changes and outcomes in order to iterate cumulatively.

Common mistakes and antipatterns

Scolding the model. Repeating "I already told you" in the chat, spending context without changing anything structural.

Changing without categorising. Tinkering with the harness blindly without diagnosing whether a guide, a sensor, scope or legibility was missing.

Not recording. Iterating on the harness without a journal, losing the learning and repeating corrections already tried.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Anthropic · Scaling Managed Agents

BLOCK E · FRONTIER PATTERNS

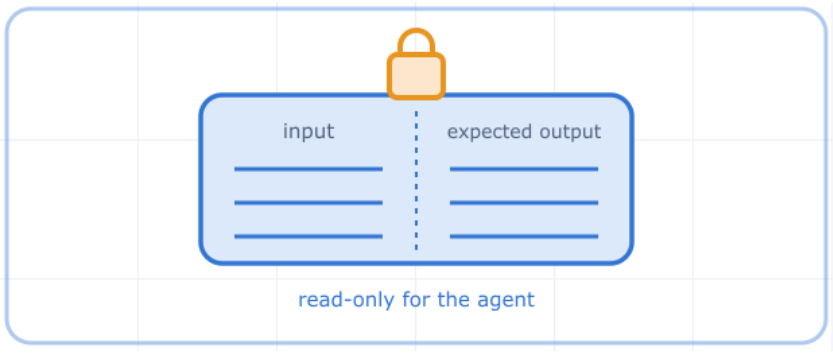
Chapter 16. Behaviour harness, approved fixtures and adversarial reviewers

EMERGING

Verifying that a deliverable actually does what it was supposed to do—not just that it is well formed—is the field's open front. There are promising patterns, not a general solution.

Introduction

The behaviour harness—how to verify that the deliverable does what it was meant to do—remains the discipline's open frontier. It is at once the unsolved problem and the one with the greatest consequences, because errors of substance are harder to detect than errors of surface.



Input/output pairs marked read-only: the agent must satisfy them; it cannot rewrite them in order to "pass".

16.1 The circular problem

A deliverable that is well built, well structured and free of surface errors can still fail to answer the question that was asked, answer it at a uselessly general level, or solve something that was not the problem. On top of this comes a circular problem: the verifications the agent itself generates tend to validate what the agent has just produced. If it misunderstood the requirement, its verification validates the misunderstanding.

16.2 Promising patterns

There is no general solution, but there are directions that work:

- Approved fixtures: the human or an expert produces (input, expected output) or (question, minimum acceptable answer) pairs, marked read-only. The agent must satisfy them and cannot modify them; if it changes the expected output in order to "pass", it breaks the contract and a hook or a sensor detects it.
- Property-based testing: instead of individual cases, invariant properties that must hold for any valid input. Hard to "game" when a property test is well written.
- Adversarial reviewers: an agent whose only job is to object, instructed to produce the harshest memorandum of objections a critical peer could write, without softening.
- The sceptical-reader test: identify the three most sophisticated possible readers and verify that the deliverable anticipates, or is not vulnerable to, their objections.

- LLM-as-judge in a fresh context: an evaluator that did not see how the deliverable was produced and applies clear acceptance criteria to it, always as a second line behind the computational controls.

16.3 The implication for autonomy

The practical consequence reappears here, now as a design principle: the more critical substantive correctness is, the less autonomy the agent should be given and the greater the human role must be. The patterns in this chapter reduce the risk, they do not eliminate it, and they are redefined almost every month; it is worth following them without adopting them out of fashion.

Why it is EMERGING. Unlike the rest of the harness, here there is neither consensus nor mature tooling. Approved fixtures and adversarial reviewers are promising but recent; this is the chapter whose content is most likely to change in the next revision of the map.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the behaviour gap and the circular problem of verifications self-generated by the agent.
- Define a set of read-only approved fixtures for a specific deliverable.
- Instruct an adversarial reviewer and apply the sceptical-reader test to an important deliverable.
- Place LLM-as-judge as a second line of verification, in a fresh context, and not as the only defence.

Common mistakes and antipatterns

Circular verification. Accepting the checks the agent itself generated, which validate its possible misunderstanding of the requirement.

Editable fixtures. Allowing the agent to modify the expected pairs to make them pass, breaking the contract without anyone noticing.

All faith in the inferential judge. Resting behaviour verification on a single evaluating LLM, expensive and fallible, with no computational first line.

Further reading

· Anthropic · Harness Design for Long-Running App Development

· Böckeler · Maintainability sensors for coding agents

BLOCK E · FRONTIER PATTERNS

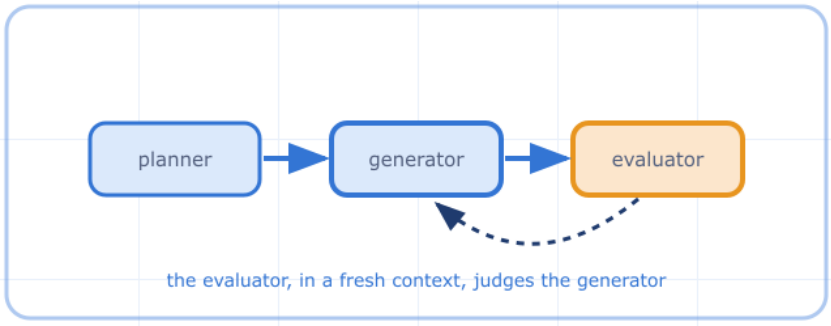
Chapter 17. Multi-agent architectures and agentic cleanup

EMERGING

The orchestration frontier: multi-role architectures that watch over each other, and background agents that clean up project drift.

Introduction

When tasks become long and autonomous, the harness evolves from one agent with sensors to several agents with distinct roles that watch over each other, and to maintenance processes that run on their own. It is powerful but volatile ground: the specific architectures change fast.



A three-role architecture where an evaluator, in a fresh context, judges the generator's work in successive cycles.

17.1 Multi-role architectures

Anthropic has documented three-role architectures—planner, generator and evaluator—inspired by generative adversarial networks: an evaluator with fresh context and no write permissions judges the generator's work in successive cycles, and each cycle produces a more refined result. The separation of roles has a key virtue: the evaluator did not see how the work was produced, so it does not inherit the generator's biases.

17.2 Brain / Hands / Session

In a complementary line, Anthropic describes the Brain/Hands/Session separation: the Brain is the model plus the harness loop that invokes it; the Hands are isolated, ephemeral environments where the tools execute; the Session is an append-only log of every thought, call and observation. The motivation is that every harness component encodes an assumption about what the model cannot do on its own; by decoupling them, when an assumption ages—because the new model can now do something that used to require a piece—that piece is replaced without redoing the whole system.

17.3 Agentic garbage collection

Every project accumulates entropy: dead code, orphaned dependencies, sources no longer cited, an unpruned glossary, reverted decisions left unrecorded. Agentic cleanup uses background agents with specific missions (“find unreferenced material and archive it”, “detect terminological inconsistencies between sections”) to correct it continuously. The rules to keep it under control: the human approves every significant deletion, material is archived before deletion, and every cleanup leaves an entry in the project log.

Few and well defined. The temptation to create a subagent or a skill for every problem leads to fifteen that overlap and send the cost soaring. If you have more than a handful active in a project, you can probably consolidate. They are a hammer: not everything is a nail.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Describe a multi-role architecture (planner/generator/evaluator) and the advantage of the fresh-context evaluator.
- Explain the Brain/Hands/Session separation and why decoupling avoids redoing the whole system when the model improves.
- Design an agentic cleanup routine with its safeguards: human approval, archiving before deleting, a project log.
- Keep a small number of subagents and skills, consolidating when they overlap.

Common mistakes and antipatterns

Coupling the whole system. Mixing model, loop, sandboxes and log so that any model improvement forces rebuilding it all.

Cleanup without safeguards. Letting an agent delete without approval or archiving, losing material irreversibly.

Subagent sprawl. Creating a new subagent or skill for every problem, until a tangle accumulates that nobody governs.

Further reading

· Anthropic · Harness Design for Long-Running App Development

· Anthropic · Scaling Managed Agents

BLOCK F · PROFESSIONAL JUDGEMENT AND THE HUMAN ROLE

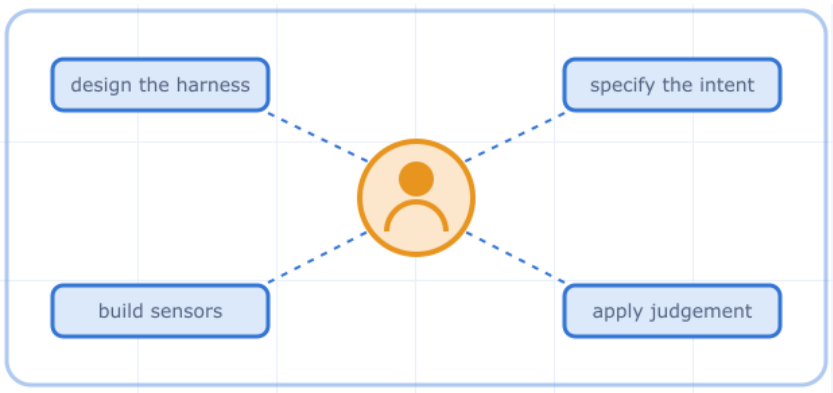
Chapter 18. Measuring the harness, avoiding antipatterns and sustaining judgement

CONSOLIDATING

A well-designed harness does not remove the human: it concentrates them where they are worth most. Measuring, recognising antipatterns, knowing when not to use an agent and sustaining judgement.

Introduction

This chapter closes the guide by bringing together the layer that protects the validity of everything before it. An impeccable harness can still sign off work that nobody has genuinely thought through if measurement is missing, if known antipatterns repeat, or if the human abdicates judgement. The four sections that follow are the four faces of that layer.



The human stops writing every line and instead designs the environment, specifies the intent, builds the feedback loops and applies judgement where it matters.

18.1 Measuring the harness

If you do not observe, you do not improve. No sophisticated system is needed: a spreadsheet with date, task, duration, iterations, blockers and objections is enough for the patterns to show after twenty or thirty entries. The most valuable indicators:

- Rebuild cost: time from an empty session to productive work. It is the north-star KPI; below a few minutes, the harness is elite.
- First-pass completion rate: percentage of tasks that close with no more than one major iteration.
- Recipient objection rate: percentage of deliverables that receive substantive external objections; it should be at a minimum.
- Sensor coverage: percentage of closes that have gone through all the defined sensors; the target is all of them.
- Harness improvement latency: time from detecting a failure pattern to closing it in the harness; ideally less than one session.

It pays to review these indicators periodically on long projects; if any of them worsens, identify what changed.

18.2 Common antipatterns

The failure modes repeat across disciplines. Recognising them early prevents reliability from eroding:

- Over-specified instructions: the file keeps growing, the model stops reading it attentively and the critical rules get lost in the noise. Antidote: aggressive pruning; if a rule has not been violated in several sessions or is mechanically enforceable, turn it into a sensor and delete it.
- Endless exploration: an unbounded “research X” exhausts the context. Antidote: explicitly bound sources and scope.
- Premature confidence: accepting the “done” without verifying. Antidote: strict pass-state gating.
- Bypass out of desperation: switching on a permissive mode so that “it just finishes”, risking something irreversible. Antidote: bypass only in isolated environments; if you are tempted to do it on your main machine, go to bed.
- Subagent sprawl: creating a new one for every problem until a costly tangle accumulates. Antidote: few and well defined.
- Faith in the inferential reviewer: resting all verification on another model. Antidote: make it the second line, not the only one.
- Hidden dependence on a single source or tool. Antidote: a diversification rule and a sensor that reports the distribution of sources at close.

18.3 When NOT to use an agent

Part of professional judgement is knowing when the right answer is to close the chat and do the work yourself:

- Short tasks whose context exists only in your head: explaining it to the agent costs more than doing it.
- Critical decisions with organisational responsibility: the agent does not carry the responsibility; you do, and your judgement is what is being paid for.
- When the client pays for your judgement, not your output: agree with them where AI may and may not enter.
- Refactors or deep changes with subtle dependencies: the agent breaks invariants it cannot see; better human and agent as a pair.
- When confidentiality is paramount: there are areas where the risk is not justified even with enterprise plans.
- When you do not know the answer: without an informed hypothesis of your own, the agent will hallucinate something plausible and you will sign it.
- When the system is broken and you do not know why: the agent will hallucinate plausible causes that throw you off; let human debugging guide you.

18.4 The human role

A well-designed harness does not remove the human: it concentrates them where they are worth most. Professional work is no longer writing every function or every paragraph; it is designing the environment (the harness), specifying the intent (the brief, the acceptance criteria, the questions the

deliverable must answer, the invariants that must not be violated), building the feedback loops (sensors, verifications, critical reviews), applying judgement where the machine must not decide alone (the thesis, the final recommendation, the architectural decision, the legal or political nuance, what goes in and what does not) and iterating on the harness when something does not work.

More demanding, not less. This is real intellectual work, demanding in a different way and more leveraged: a good harness produces weeks of quality output while you think about what genuinely deserves your thinking. And technical humility is advisable: the truths of this field have short lives, so the willingness to unlearn is worth more than any acquired certainty.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Define and track the harness KPIs—rebuild cost, first-pass completion, objections, sensor coverage, improvement latency—with a simple log.
- Recognise the common antipatterns in a project of their own and apply the corresponding antidote.
- Decide, for a specific task, whether to use an agent or work it by hand, according to responsibility, confidentiality and one's own knowledge.
- Delimit the human role as designer of the harness, specifier of intent, builder of sensors and ultimate holder of judgement over the deliverable.

Common mistakes and antipatterns

Not measuring. Working with no indicators at all, so the harness does not improve because nobody knows whether it is getting worse.

Abdicating judgement. Accepting the agent's output on decisions where responsibility and judgement cannot be transferred.

Using the agent where it does not belong. Resorting to an agent for tasks that demand tacit context, paramount confidentiality or an informed hypothesis one does not have.

Further reading

· Böckeler · Harness engineering (martinfowler.com)

· Mollick · One Useful Thing