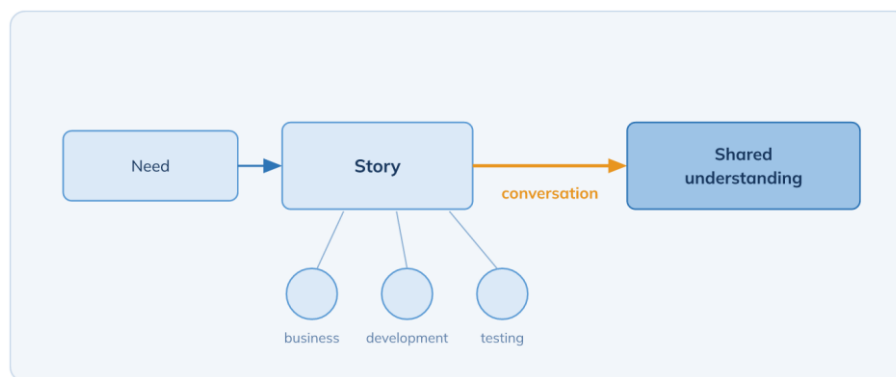


Guide

User Stories

In-depth development of the State of the art topics



Updated September 2026

About this document

This document is an in-depth development of the topics in the reference map (State of the art) for work with user stories and with the other items of the Product Backlog, from writing them when they help and talking them through to judging their quality, splitting them without losing value, keeping the context of the whole and verifying that the result matches the intent, including when it is artificial intelligence that drafts or implements them, 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 [User Stories](#) 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 user stories is settled at its core and moves very fast at its frontier: the pieces that define it are twenty to twenty-five years old and remain valid without major revisions, while a band of practice gains adoption without being universal (measuring quality against explicit criteria, example mapping, right-sizing by empirical measurement, preserving the context of conversations) and the artificial intelligence and agents layer changes within months. 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 user story: what it is and where it belongs

1. The story as a promise for a conversation: origin, the three Cs and the template
2. What the frameworks say: from Scrum's neutrality to SAFe's institutionalization
3. Shared understanding as the criterion for how much to write

Block B — The quality of a story and of the whole

4. INVEST and its rereading: a work guideline, not a checklist
5. Defects and ambiguity: what the text decides and what the conversation decides
6. The catalog of story antipatterns
7. Backlog health and the Definition of Ready

Block C — The conversation and its confirmation

8. Continuous refinement: neither handoff nor meeting
9. Who talks and in what format: Three Amigos and story-writing workshops
10. Acceptance criteria: formats, limits and confusions
11. Rules, examples and Example Mapping
12. From examples to verifiability: BDD without a tool
13. Preserving the context of the conversation

Block D — Splitting while keeping value

14. Why, how much and when to split
15. Vertical slicing and legitimate technical slices
16. The catalog of splitting patterns
17. From pattern to slice: procedure, choice and dependencies

Block E — Story mapping and backlog context

18. What gets lost in a flat backlog
19. The story map: anatomy and construction
20. Slicing by outcome and keeping the map alive
21. Neighboring maps and the map inside the tool

Block F — Not everything is a story

22. Product Backlog item versus story
23. Technical work without fake users
24. Alternative representations and how to choose the format

Block G — Artificial intelligence in story work

25. The asymmetry: artificial intelligence reviews better than it writes

- 26. Context, invention and provenance
- 27. Risks and the working pattern that does not delegate judgment

Block H — The story as the input to an agent workflow

- 28. From story to spec: intent as the source of truth
- 29. What the agent needs and the criterion that verifies itself
- 30. The requirements gate from product

BLOCK A · THE USER STORY: WHAT IT IS AND WHERE IT BELONGS

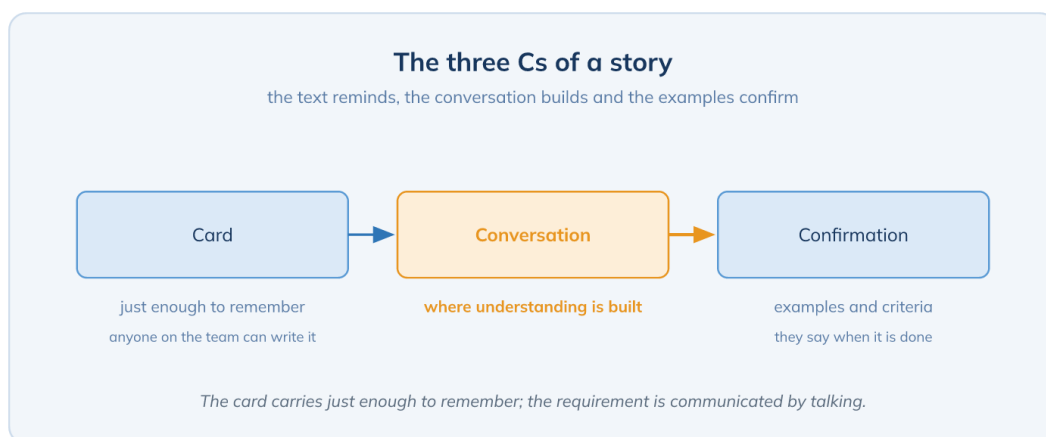
Chapter 1. The story as a promise for a conversation: origin, the three Cs and the template · ESTABLISHED

Where the user story comes from, what sets it apart from a requirement and why its value lies in the conversation it promises.

Introduction

Almost everyone who works in product has written a user story, and a fair number have done so believing they were writing a short requirement. That is where most of the problems in this guide come from. In this first chapter we go back to the origin to recover what the technique says about itself, because nearly every mistake made with it consists of forgetting what it was invented for.

The journey is short and worth taking with the sources at hand. The story is born in extreme programming in the late nineteen-nineties, is organized in 2001 with Ron Jeffries's three Cs and soon adopts the template that any tool recognizes today. We will look at the three pieces and, above all, at the part each one plays.



1.1 Where it comes from and what problem it solved

User stories appear in Chrysler's C3 project in 1997 and are first described in *Extreme Programming Explained* (Kent Beck, 1999), where they are the unit customer and team use to plan the work. Their original function was not to document but to **prompt a conversation at the right moment**. Alistair Cockburn, who visited that project, coined the formulation that captures it best: a user story is a promise for a conversation.

The contrast with the traditional requirement clarifies the intent. A requirement aims to be complete, approved and stable, and its value lies in making further questions unnecessary. A story aims for the opposite: to be small and incomplete enough that talking about it before building is worth the trouble. Mike Cohn puts it plainly: the written story is a reminder, and the real requirement emerges through conversation, examples, rules, sketches, tests and decisions.

It is also worth telling it apart from two neighbors it gets confused with. A **use case** describes a complete flow with its alternatives and exceptions, and it is a valuable tool when the flow matters (we return to it in chapter 24). A **task** is a slice of the technical work needed to complete something,

not a slice of observable value: «create the users table in the database» is not a story, even if someone wrote it with the template.

1.2 The three Cs: card, conversation and confirmation

In August 2001, Ron Jeffries arranged the technique in three parts that remain the best way to explain it. The **card** contains «just enough text to identify the requirement, and to remind everyone what the story is», which is why the traditional medium was an index card, whose size made writing too much impossible. The **conversation** is where the requirement is really communicated, through an exchange of thoughts and opinions between whoever has the need and whoever will meet it. The **confirmation** is the examples and acceptance tests that will say whether what was built meets the need.

All three, not one. A story with a card and no conversation is a specification in disguise; with conversation and no confirmation, an agreement nobody will be able to check. Professional work consists of sustaining all three.

Seven years later, Jeffries returned to the subject with a line worth keeping at hand when someone asks for «better-written stories»: a user story is not a requirements document, it is just a thing to do; and the writing is probably the least important part, far behind the thinking, the communicating and the testing.

1.3 The Connextra template and its limits

The template «As a [type of user], I want [something], so that [a benefit]» was invented in 2001 at Connextra, a British company, hence its name. It came from a concrete problem: the people writing the requests were from sales and marketing, they noted the functionality and forgot to say who needed it and why, so the team had to track down the interested party in order to have a conversation. The template forces those two facts in.

The Agile Alliance glossary describes it as training wheels and warns that spending much effort on complying with it is without much point. Cohn, who popularized it, now points out two limits: the generic «as a user», which often indicates the item does not fit the story mold at all, and slavish adherence to the formula. The third part, the why, is the most neglected and the most valuable, because a need with an explicit benefit admits several solutions and lets you judge them.

At Vantia, the tax management platform that runs through the examples in this guide, the request from the commercial area was «we need a chatbot». Framed that way there is no user, no need and no why, only a solution chosen by whoever asked for it. The same thing, after talking to the freelancers who were abandoning the platform at night, turned into something quite different: when someone gets stuck with a filing outside their accounting firm's opening hours, they want a clear explanation right there so they can finish that same night.

1.4 The unit of work and what Scrum asks for

A vocabulary of sizes has grown around the story, and it is worth using fluently and without dogma: **theme**, **epic**, **story** and **task** are containers by volume, handy for organizing a conversation, and not categories with formal properties. We return to them in chapter 14, when we discuss when to split.

What Scrum requires of a Product Backlog item is far less than people usually believe: a description, an order and a size. Nothing else. How that item is expressed belongs to the craft and is chosen case by case, and the whole of block F is devoted to it. Starting here avoids the sterile argument about whether stories are «mandatory».

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the origin of the user story and what problem it solved, citing its sources.
- Tell a story apart from a traditional requirement, a use case and a task.
- State the three Cs and say what each contributes to understanding the work.
- Use the Connextra template when it helps and drop it when it gets in the way, without losing the user or the why.
- Reframe as a need with a benefit a request that arrives already turned into a solution.

Common mistakes and antipatterns

Treating the card as the deliverable. When the effort goes into writing the card well, the conversation is assumed to have happened and the misunderstandings arrive intact at the review.

Writing «as a user, I want...». A generic actor tells you nothing and usually signals that nobody knows who benefits; if there really is no distinguishable user, the item is probably not a story.

Filling the «so that» with a tautology. «So that I can see it on screen» is not a benefit; the why must be arguable, and if there is none, it is worth finding out before building.

Turning technical tasks into stories. The Scrum Manager BoK flags this as the most frequent mistake in Spanish-speaking teams: the task disguised as a story makes value-based ordering impossible and hides who the work serves.

Further reading

- [Jeffries — Card, Conversation, Confirmation](#)
- [Jeffries — How should user stories be written?](#)
- [Agile Alliance — User story template](#)
- [Cohn — User Stories \(reference guide\)](#)
- [Scrum Manager BoK — User story \(in Spanish\)](#)

BLOCK A · THE USER STORY: WHAT IT IS AND WHERE IT BELONGS

Chapter 2. What the frameworks say: from Scrum's neutrality to SAFe's institutionalization · ESTABLISHED

What each framework requires about user stories, so as not to defend as a rule what is in fact a team's choice.

Introduction

A good part of the arguments about user stories would be settled by reading what the frameworks actually say, which is considerably less than what is attributed to them. In this chapter we review the positions with their sources, because knowing them saves time in organizations where someone claims that «Scrum requires» something.

2.1 Scrum does not prescribe them

The 2020 Scrum Guide does not contain the phrase «user story». What it asks of a Product Backlog item is a description, an order and a size, and it describes refinement as the act of breaking down and further defining those items. The 2017 version listed types («features, functions, requirements, enhancements, and fixes») and that list disappeared in 2020, which reinforces the reading.

Scrum.org stated it unambiguously in 2017, in an article whose title is already the answer: Scrum neither prescribes nor requires user stories. LeSS is just as explicit in saying that the Product Backlog does not contain user stories, it simply contains items. Scrum Alliance points in the same direction when describing the artifact.

The practical consequence. If nobody requires it, choosing the format is a professional decision that has to be justified. That justification is the content of block F.

2.2 SAFe and Disciplined Agile formalize them

At the other end, SAFe turns the story into an artifact with rules. It defines the user story with the template and with INVEST, sets that it should be completable in a few days or less, and adds a second type, the **enabler story**, in four varieties: exploration, architecture, infrastructure and compliance. It is a reasonable answer to the problem of coordinating many teams, and worth knowing because many large organizations work that way.

Disciplined Agile, within PMI, describes decomposing features into stories as a practice with its own procedure. The PMI-ACP includes decomposing backlog items as needed in its exam content outline, without prescribing a format.

Kanban is neutral about the shape of items: its guide speaks of work items and concentrates on flow, visualization and explicit policies. That neutrality is consistent with the approach of this guide, and in chapter 14 we will see what its idea of sizing items against a service-level expectation contributes.

2.3 Requirements engineering treats them as one technique

The IIBA's Agile Extension to the BABOK Guide and IREB's material on agile requirements place the story where it belongs from their point of view: one requirements expression technique among

several, with its strengths and its limits. IREB's material covers the three Cs and INVEST, and stresses that the most useful properties of the acronym are estimability, size and testability.

This view adds something the agile literature tends to take for granted: there are contexts, such as systems with regulatory certification, where the story alone is not enough and it pays to combine it with formal requirements or use cases. Chapter 24 develops this with decision criteria.

2.4 The debate about the term itself

There is an open and healthy discussion about whether it is worth continuing to call backlog items «stories». Those who propose dropping the term argue that it carries a confusion hard to undo, because it passes off a technique as a rule and pushes teams to force the template onto work that does not admit it. Those who defend it recall that the name comes from how the technique is used, by telling stories, and that the problem is not in the word.

Scrum Manager's position, which this guide adopts, is intermediate and practical: keep the term for its communicative value and its roots, teach it with its origin and its limits, and make clear from the start that a Product Backlog item does not have to be a user story.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Say what the Scrum Guide does and does not require about user stories, with its text.
- Explain how SAFe and Disciplined Agile formalize the story, and what enabler stories are.
- Place the story as one requirements technique among others, per the Agile Extension to the BABOK Guide and IREB.
- Argue in your organization that the format of a backlog item is a team decision and not a framework rule.

Common mistakes and antipatterns

Attributing to Scrum what Scrum does not say. It is the source of endless arguments; the text of the guide is enough to close them, and it is worth having it to hand.

Applying SAFe's rules outside SAFe. The few-days size or the enabler taxonomy make sense inside its coordination framework; transplanting them without that context adds bureaucracy without the benefit.

Dismissing use cases as «not agile». There are flows and regulated contexts where they are the right tool, and giving them up out of method loyalty is a poorly grounded decision.

Further reading

- [The Scrum Guide 2020](#)
- [Scrum.org — Myth 4: the Product Backlog need not consist of user stories](#)
- [SAFe — Enablers](#)
- [LeSS — Product Backlog](#)
- [IREB — RE@Agile](#)

BLOCK A · THE USER STORY: WHAT IT IS AND WHERE IT BELONGS

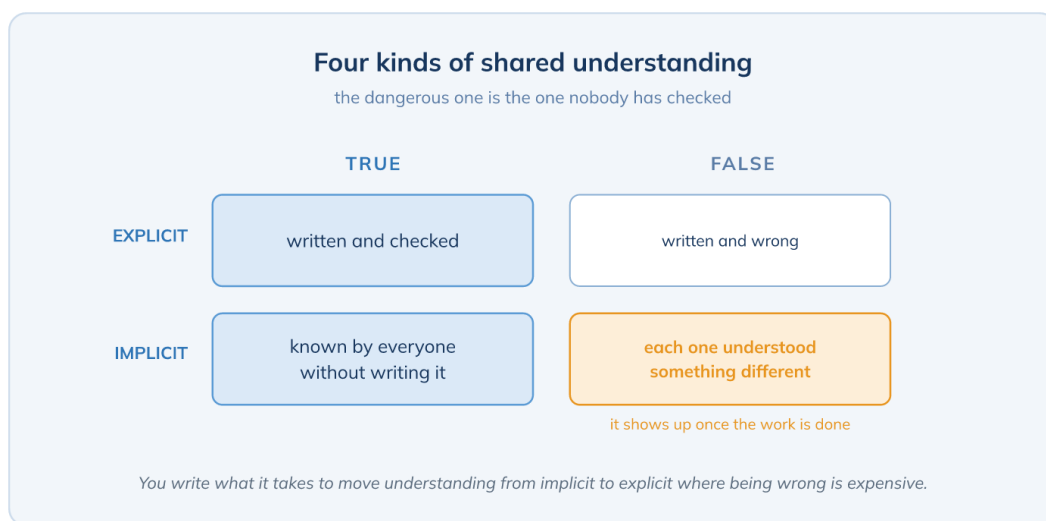
Chapter 3. Shared understanding as the criterion for how much to write

· ESTABLISHED

The criterion that decides how much is worth writing in each case, and the thread that runs through the whole skill.

Introduction

If the story is a promise for a conversation, one question remains that no template can answer: how much to write. Writing too much does not create understanding and consumes time; writing too little leaves the work at the mercy of each person's interpretation. In this chapter we present the criterion for deciding, which is also the thread running through the whole guide.



3.1 Explicit, implicit and false

Martin Glinz and Samuel Fricker proposed a model of shared understanding in software engineering that names what agile practice had been doing by intuition. Understanding can be **explicit**, when it is written down and can be checked, or **implicit**, when it lives in the heads of the people working together and rests on the context they share. And in both cases it can be **true** or **false**.

The dangerous case is false implicit understanding: everyone believes they agree, nobody has checked anything, and the discrepancy shows up once the work is done. It is exactly the phenomenon that sinks «well-written» stories nobody talked through. Conversation does not remove that risk by itself, but it is the only thing that makes it visible in time, because it forces different interpretations to collide.

The criterion. You write what it takes to move understanding from implicit to explicit wherever being wrong would be expensive. No more, and nowhere else.

3.2 Why conversation is richer than a document

Cockburn ranked communication channels by richness: two people at a whiteboard convey more, and faster, than a reviewed document; video conveys more than audio, and audio more than paper.

The reason is that a rich channel lets you ask, point, correct and check understanding on the spot, while a document only lets you read.

Hence Gojko Adzic's watchword, which sums up the craft in four words: tell stories, don't write them. It does not mean writing nothing, but that what is written serves the conversation instead of replacing it. Patton says it from another angle when he reminds us that shared documents are not shared understanding.

3.3 What changes with artificial intelligence

For twenty-five years, the practical constraint protecting this criterion was the cost of writing: nobody drafted twenty stories with detailed criteria because it took days. That cost has gone. Today, producing plausible text about any product takes minutes, and shared understanding still costs exactly what it always did.

The consequence is that the criterion matters more, not less. When volume stops being a signal of effort, it also stops being a signal of quality, and professional value shifts to three tasks that assistance does not do for anyone: giving the right context, critiquing what it returns and verifying that the result matches the intent. Blocks G and H develop those three tasks.

3.4 The route of this guide

With that thread, the route falls into place. Blocks A and B settle what a story is and how its quality and that of the whole are judged. Block C takes the story into the conversation and closes it with rules, examples and criteria. Block D teaches how to split without losing value, which is where most teams get stuck. Block E gives back the context an ordered list loses. Block F answers what to do when the work is not a story. And blocks G and H deal with artificial intelligence, first as an assistant to story work and then as the party implementing it.

All the concrete examples use the same fictional product, **Vantia**, the one from the Product Owner 2.0 core guide: a platform where thousands of freelancers keep their invoicing, expenses and taxes, sold by license to the accounting firms that serve them. Its known problem is that nearly a third of those who use it at night abandon it within the first year, because they get stuck and have nobody to ask.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain shared understanding with its four cases and recognize false implicit understanding in a real situation.
- Decide how much to write in a specific story with that criterion, rather than with a template or a fixed list.
- Argue why conversation is a richer channel than a document, and when writing is still worthwhile.
- Explain what changes in story work when writing no longer has a cost.

Common mistakes and antipatterns

Measuring refinement by written detail. A highly detailed backlog can hold less shared understanding than a brief, well-discussed one; detail is not a measure of readiness.

Settling for apparent agreement. Asking «is that clear?» produces nods; making interpretations collide with concrete examples produces understanding.

Writing to protect yourself. A story drafted as a contract so that blame can be assigned later turns the document into the goal and switches the conversation off.

Further reading

- [Glinz — Requirements Engineering, a personal reflection \(with the 2015 model\)](#)
- [Adzic and Evans — Fifty Quick Ideas to Improve Your User Stories](#)
- [Fowler — UserStory](#)
- [Skill Arena — Product Owner](#)

BLOCK B · THE QUALITY OF A STORY AND OF THE WHOLE

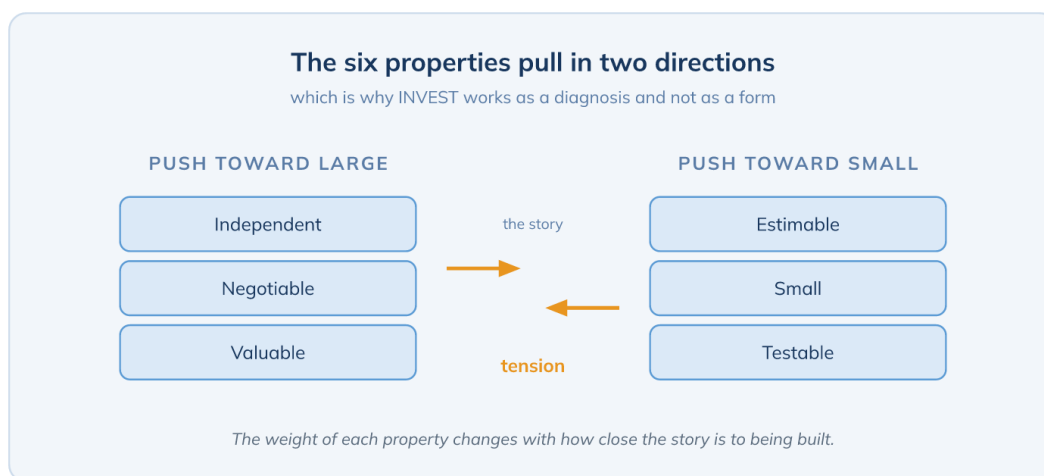
Chapter 4. INVEST and its rereading: a work guideline, not a checklist

ESTABLISHED

The six properties, their diagnostic questions, the tension between them and the revision made by its own author.

Introduction

INVEST is the most cited quality criterion in the craft and also the most mistreated. Bill Wake published it in 2003 as a mnemonic for six desirable properties of a story, and in many organizations it has ended up as a form somebody fills in before letting an item into the sprint. In this chapter we recover it as what it is: a diagnostic instrument for when a story resists.



4.1 The six properties and their useful question

Independent. Wake asked that stories not overlap in concept and be schedulable and implementable in any order, and added the caveat that this cannot always be achieved. The diagnostic question is direct: could we ship this one and not the next? If the answer is no, there is a dependency worth making visible before planning.

Negotiable. A story is not a contract for features; the detail is co-created by whoever has the need and whoever meets it. Question: is there anything left to decide together, or is it all already closed in writing?

Valuable. It must be valuable to someone identifiable. Question: who will notice the difference when this ships, and in what? If there is no answer, it is not that the story is badly written, it is that it is not ready to be built.

Estimable. Precision is not needed, only enough knowledge to order and schedule it. Question: do we know enough to say whether it is large or small? If not, the right move is to investigate rather than guess.

Small. Here lies the most striking change of scale, which we look at in 4.3. Question: does it fit comfortably in one cycle, with room to talk it through and check it?

Testable. It must be possible to say what we will observe to know it is done. Question: could we write an example that confirms it? If not, conversation is missing, and block C is devoted to that.

4.2 The internal tension, which is what makes it useful

The six properties do not pull in the same direction, and that is precisely their usefulness as an instrument. Independence, negotiability and value push toward large stories, because a slice of complete value usually needs several pieces. Estimability, size and testability push toward small ones. An item that met all six perfectly would be a rarity.

How the tension is used. The weight of each property changes with how close the story is to being built: on the distant horizon value and independence rule; once it enters the sprint, size and testability do. Diagnosing means asking which property is missing **now**.

Hence INVEST works badly as an entry gate and well as a diagnosis. When a story resists, the acronym says where to look: if it cannot be split without losing value, what is usually missing is value (we see this in chapter 17); if it is not testable, examples are missing; if it is not estimable, information is missing and probably a spike.

4.3 The author's own rereading and the change of scale

Wake has revised his acronym in public for two decades. He qualified independence in 2012, proposed reading the S as **scalable** in 2016 (a story can grow or shrink depending on the moment, and what matters is that it admits that adjustment) and in 2021 published «All You Need is INVEST? No!», where he adds the property of being **end-to-end**, that is, cutting through the system instead of staying in one layer. That last property anticipates the vertical slice of chapter 15.

The change of scale deserves attention. The 2003 text says a story represents «at most a few person-weeks worth of work». SAFe now speaks of stories finished in a few days or less, and practice in product teams runs from half a day to two days. It is not that the definition has changed: it is that the size a team can sustain with continuous integration and frequent deployment is far smaller than in 2003.

4.4 What the evidence says about its use

The REFSQ survey of 2016, with 182 practitioners, offers three facts worth knowing. First, that only one team in four uses INVEST, while almost 60 % use the Connextra template and close to 40 % follow no guideline at all. Second, and more interesting: what improves quality is **using a template**, not which one. And third, that INVEST's usefulness concentrates in teams that are starting out.

The study's authors put it as a recommendation this guide endorses: INVEST is not a checklist, but a work guideline. The Scrum Manager BoK says the same in other words when it describes it as a conversational guide and not a quality form.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Diagnose a story with the six properties and say which one is missing in each case.

- Explain the tension between the properties and adjust their weight according to how close the story is to the work.
- Incorporate Wake's rereading, with the scalable reading of size and the end-to-end property.
- Place the advisable size on today's scale rather than on that of 2003.
- Reject with arguments the use of INVEST as an entry gate to the sprint.

Common mistakes and antipatterns

Turning it into a form. An item that ticks all six boxes may still have no observable value; the acronym does not replace the conversation that decides whether it is worth doing.

Demanding absolute independence. Wake already warned it cannot always be achieved; professional work consists of making dependencies visible, not denying them.

Confusing estimable with estimated. That a story admits an estimate does not oblige you to put a number on it; chapter 14 covers right-sizing as an alternative.

Chasing tiny stories. Below a certain size the piece stops being a slice of value and becomes an implementation detail in disguise.

Further reading

- [Wake — INVEST in Good Stories, and SMART Tasks](#)
- [Wake — All You Need is INVEST? No!](#)
- [Wake — Small/Scalable Stories](#)
- [Lucassen et al. — The use and effectiveness of user stories in practice \(REFSQ 2016\)](#)
- [Scrum Manager BoK — INVEST \(in Spanish\)](#)

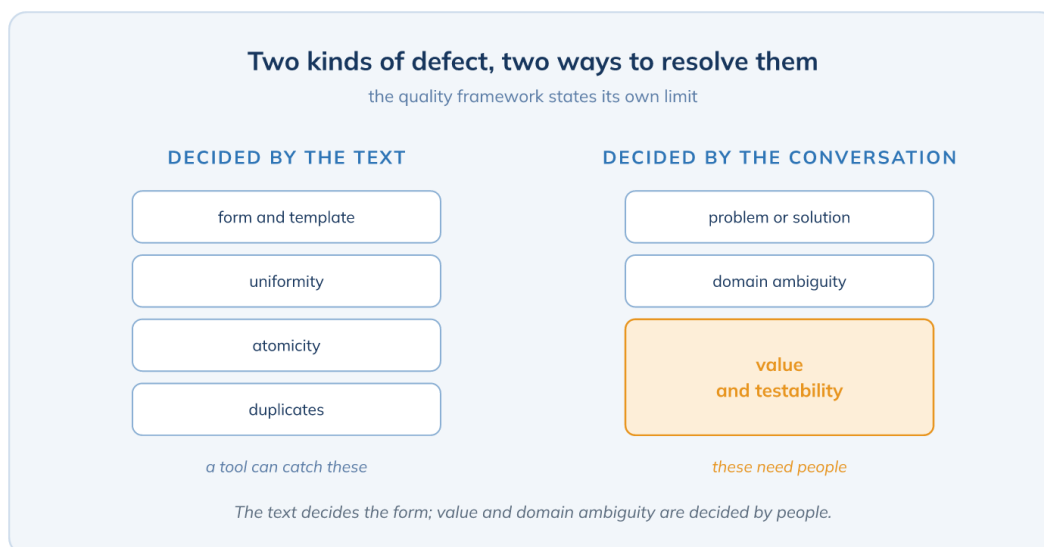
BLOCK B · THE QUALITY OF A STORY AND OF THE WHOLE

Chapter 5. Defects and ambiguity: what the text decides and what the conversation decides · CONSOLIDATING

Which defects can be caught by reading, which ones need people, and why a detected ambiguity is an opportunity.

Introduction

There is a distinction that saves a lot of refinement time: not every defect in a story is resolved the same way. Some can be seen by reading the text and a tool can catch them; others only surface when two people discover they understood different things. Knowing which group the defect in front of us belongs to decides who should resolve it and how much it will cost.



5.1 The quality framework and its stated limit

The **Quality User Story** framework (Lucassen, Dalpiaz, van der Werf and Brinkkemper, 2016) is the most rigorous attempt to measure the quality of a story. It arranges thirteen criteria in three categories. The **syntactic** ones look at form: being well-formed (with actor and action), atomic and minimal. The **semantic** ones look at content: being conceptually sound, problem-oriented rather than solution-oriented, unambiguous and conflict-free. The **pragmatic** ones look at the whole: full sentence, estimatable, unique, uniform, independent and complete.

The most valuable part of the framework is what it declares out of scope. Its authors expressly exclude **value** and **testability**, and explain why: they cannot be decided by reading the text. That boundary, stated by the very people who built the analysis tool, is the best argument against the idea that story quality can be fully automated.

The split. Form, uniformity, atomicity and duplicates: a tool or a language model can catch them. Problem versus solution, domain ambiguity, value and testability: these need people who know the product.

5.2 What was measured, and with what result

The team applied its tool to 1,023 real stories from 18 organizations. 56 % had at least one defect, and the most frequent were precisely the mechanical ones: lack of uniformity, stories that were not minimal (with detail that does not belong on the card) and not atomic (two needs in one). It is a useful figure because it shows where automation genuinely helps, and also its ceiling.

Brought into daily work, it suggests a sensible order: let the mechanical review clean up the form before refinement, so the conversation is spent on what only conversation resolves. In block G we will see that a language model does that job well, and that this is its highest-yield, lowest-risk use.

5.3 Ambiguity has two faces

Requirements research distinguishes several kinds of ambiguity: **lexical** (a word with more than one meaning), **syntactic** (a sentence that admits two parses), **semantic** (a sentence with two readings in its context), **pragmatic** (it depends on who reads it) plus **vagueness** and **generality**. For story work, the practical thing is to have the risky constructions of your own language pinned down.

The short list in Spanish covers evaluative adjectives without a measure (adequate, fast, easy, intuitive, efficient), imprecise quantifiers (several, some, most), the «and/or» conjunction, escape clauses (as appropriate, as far as possible, if necessary) and catch-all verbs (manage, support, handle), which hide behavior without describing it. To that we add hidden assumptions and language that is either too technical or too generic.

The second face is the interesting one. Ferrari, Spoletini and Gnesi's research on requirements elicitation interviews shows that detected ambiguity is a **signal of tacit knowledge**: where an expression admits two readings there is usually a domain rule nobody has stated because it seems obvious. Hence the operating rule: you do not have to eliminate all ambiguity in writing, but to locate the kind with high probability of misunderstanding and high cost, ask the question and decide whether to document it or simply agree it.

5.4 The Vantia example

«As a freelancer, I want the assistant to quickly resolve my tax questions, so that I do not waste time.» The story is well-formed, atomic and has its why, so it would pass any mechanical review. And yet it hides three ambiguities that decide the work: **quickly** (a minute? a second?), **resolve** (answer? answer with the regulatory reference? hand off to the advisor when it does not know?) and **tax questions** (any? only those related to the user's own data?).

None of the three is resolved by reading. The first is closed with an agreed number; the second is a business rule that at Vantia turned out to be critical, because a made-up answer is worse than admitting the limit; and the third bounds the scope. All three came out of a twenty-minute conversation, and the last two became acceptance criteria with their examples.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Classify a story defect as mechanical or conversational and decide who resolves it.
- Apply the thirteen criteria of the quality framework and explain why value and testability are left out.

- Locate risky constructions in a text and formulate the question that closes each one.
- Decide which ambiguities are worth eliminating in writing and which can simply be agreed in conversation.
- Use a detected ambiguity to surface a domain rule nobody had stated.

Common mistakes and antipatterns

Polishing the form and calling it refinement. Making fifty stories look consistent does not change what the team understands about them; it is necessary work, but it is not refinement.

Chasing zero ambiguity. Documenting every nuance produces specifications nobody reads and removes the conversation that would have made them unnecessary.

Delegating value detection to a tool. The framework's own authors exclude value because it cannot be decided from the text; no tool knows it.

Accepting catch-all verbs. «Managing expenses» can mean five different behaviors, each with a different cost; it is worth breaking down before estimating.

Further reading

- [Lucassen et al. — Quality User Story framework \(REJ 2016\)](#)
- [Lucassen et al. — Forging High-Quality User Stories \(RE 2015\)](#)
- [Femmer et al. — Requirements Smells](#)
- [Ferrari et al. — Ambiguity and tacit knowledge](#)
- [Berry et al. — Linguistic sources of ambiguity](#)

BLOCK B · THE QUALITY OF A STORY AND OF THE WHOLE

Chapter 6. The catalog of story antipatterns · ESTABLISHED

The ten named failure modes, how each one is recognized and what action corrects it.

Introduction

Problems with stories repeat so often that they have names. Knowing the catalog saves half the argument, because each pattern calls for a different action and the wrong action makes things worse: splitting a story that is actually a duplicate produces two duplicates. In this chapter we go through the ten most frequent ones with their remedy.

6.1 Antipatterns of the individual story

Story-task. Describes technical work without saying who it serves («create the index on the invoices table»). Recognized because nobody outside the team notices anything when it ships. Remedy: find the real beneficiary or reclassify it as a technical item, per chapter 23.

Fake or generic user. «As a user, I want...» or, worse, «as a system, I want...». Recognized because the actor makes it impossible to argue about priority. Remedy: identify the specific user or accept that the item is not a story.

Solution disguised as a need. «I want a chatbot», «I want a dropdown with the provinces». Recognized because the «so that» restates the solution in other words. Remedy: dig down to the problem and reframe, which is what Vantia did with the chatbot request.

Story with no why or value. It has an actor and an action, and the benefit is a tautology. Remedy: ask what changes for whoever asked for it, and if there is no answer, do not start building.

Epic disguised as a story. It fits in a sentence but contains weeks of work («I want to manage my taxes»). Recognized by the catch-all verb and by the impossibility of giving a single example that confirms it. Remedy: split, using block D.

6.2 Antipatterns of the whole

Excessive fragmentation. Stories so small that none delivers value on its own and all of them must ship for anyone to notice. Remedy: recombine before splitting again, with a value criterion.

Duplicated or contradictory. Two items asking for the same thing in different words, or contradicting each other on a rule. Remedy: merge or resolve the conflict in conversation, and reorder.

Obsolete. Stories that answered a goal that is no longer current. Remedy: discard them without ceremony; chapter 7 covers expiry and backlog bankruptcy.

Prescriptive story. Dictates the implementation or the interface design («with a three-tab side menu»). Remedy: separate what is a need from what is a proposal, and give the how back to the team.

Story treated as a contract. Closed in writing before being talked through and invoked afterwards to point out breaches. Remedy: it is a relationship problem rather than a drafting one, and it is fixed in refinement (chapter 8).

How the catalog is used. In refinement, naming the pattern out loud shortens the discussion and depersonalizes the critique: what is being diagnosed is a backlog item, not a colleague's way of writing.

6.3 From detection to action

The ten patterns are resolved with six actions: **split**, **merge**, **rewrite**, **reclassify** (stop treating it as a story), **postpone** and **discard**. The mapping is not unique, but it is fairly stable: the disguised epic and the large story call for splitting; fragmentation and duplicates, for merging; the fake user and the disguised solution, for rewriting; the story-task, for reclassifying; and obsolete ones, for discarding.

The Scrum Manager BoK flags the first on the list as the most frequent mistake in Spanish-speaking teams, writing technical tasks instead of value, and that matches the experience of most teams starting out. It is worth reviewing before anything else.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Recognize the ten antipatterns in a batch of real stories and name them.
- Choose the corrective action that matches each pattern, instead of applying splitting as a universal remedy.
- Tell a legitimate large story from an epic disguised as a story.
- Use the catalog in refinement as a shared diagnosis rather than as personal criticism.

Common mistakes and antipatterns

Splitting everything that does not fit. If the problem is duplication or lack of value, splitting multiplies the problem instead of solving it.

Confusing prescriptive with detailed. A story can carry plenty of useful detail about rules and examples without dictating the implementation; what gets in the way is the technical decision, not the detail.

Keeping obsolete items «just in case». A backlog with expired items forces the team to reread them at every refinement and erodes trust in the order.

Further reading

- [Pichler — Why User Stories Fail](#)
- [Pichler — 10 Tips for Writing Good User Stories](#)
- [Humanizing Work — Does everything in our backlog need to be a user story?](#)
- [Scrum Manager BoK — User story \(in Spanish\)](#)

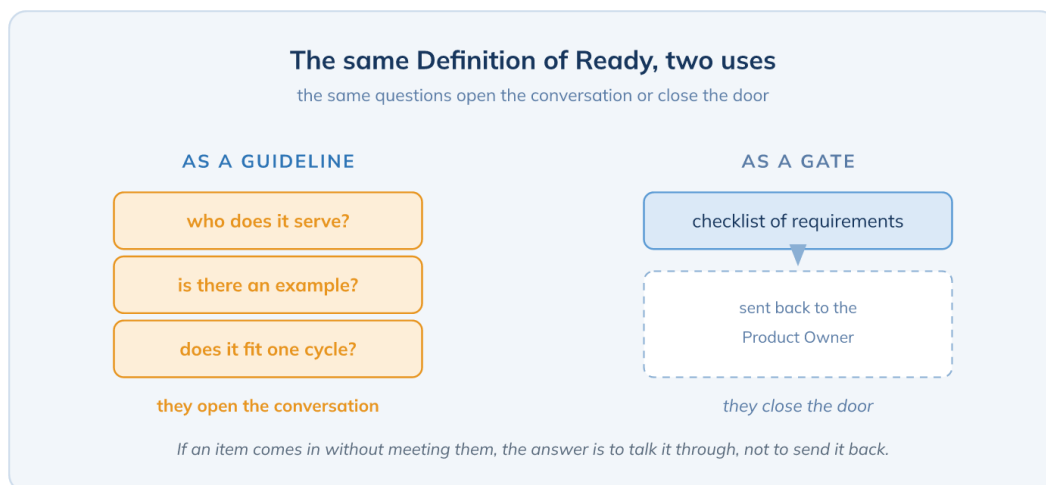
BLOCK B · THE QUALITY OF A STORY AND OF THE WHOLE

Chapter 7. Backlog health and the Definition of Ready · ESTABLISHED

The criteria that cannot be seen in an isolated story, the hygiene of the whole and how to keep the Definition of Ready from becoming a gate.

Introduction

A backlog can be full of impeccable stories and still be useless. The quality of the whole has criteria of its own, and its decay is gradual: items that age, duplicates nobody notices, detail piled onto things that will never be done, and dependencies that stay invisible until they block a sprint. In this chapter we look at how that health is measured and maintained.



7.1 The criteria of the whole

The quality framework of chapter 5 contributes five criteria that only make sense when looking at the whole backlog: **uniqueness** (no item duplicates another), **uniformity** (a reasonably similar format, so it can be read in one pass), **conflict-freedom** (two items do not ask for incompatible rules), **independence** (dependencies are identified) and **completeness** (no piece is missing without which the rest does not hold).

To those we add Roman Pichler's **DEEP** acronym, which describes a healthy backlog: **detailed appropriately** (plenty of detail at the top, little at the bottom), **estimated** (with enough sizing to order it), **emergent** (it changes with what is learned) and **prioritized**. The first property is the most often broken, and breaking it is expensive: effort goes into detailing what may never be built.

7.2 Dependencies, age and expiry

A backlog's dependencies come in three kinds: between items (one story needs another), with other teams and with third parties outside the organization. The hierarchy for handling them is always the same: **make them visible**, **remove the avoidable ones** and **manage the unavoidable ones**, in that order. Chapter 17 returns to them, because splitting badly multiplies them.

The age of items is an underused indicator. An item that has sat in the backlog for months without moving up is telling you something: either it does not matter, or nobody knows how to tackle it, or it

answers an abandoned goal. Putting a review date on items and discarding what has expired without ceremony keeps the whole readable. When the decay is general, there is a radical and healthy way out, declaring **backlog bankruptcy**: archive everything and rebuild from the current goal. It costs less than it sounds, because what matters resurfaces on its own.

Non-functional requirements. Performance, accessibility or security almost never fit in a story. Their natural home is a cross-cutting constraint or a Definition of Done criterion, verified in every increment. Chapter 10 develops this.

7.3 The Definition of Ready and its drift

The **Definition of Ready (DoR)** is a team agreement about what an item must meet before work can start on it. It is not part of Scrum: the Scrum Guide does not mention it and merely says that items that can fit within a Sprint are deemed ready for selection. The Scrum Manager BoK documents it as a practice, with its variant for AI-assisted work.

Its drift is well known and Mike Cohn has described it bluntly: turned into a gate, the DoR gives the team a formal reason to hand work back and the Product Owner a form to fill in, and it replaces conversation with paperwork. Then the two typical consequences appear: a bottleneck in refinement and a supplier-customer relationship inside the same team.

The healthy way to write it is as a **conversation guideline**: a few questions that help decide whether it is worth starting (is it clear who it serves? is there an example that confirms it? is the dependency identified? does it fit in one cycle?), agreed by the team, revisable, and with no power of veto. If an item comes in without meeting them, the answer is to talk it through, not to send it back.

7.4 With artificial intelligence

Reviewing the whole is one of the uses where assistance performs best, because a good part of the criteria of the whole are mechanical: detecting duplicates and overlaps, flagging inconsistent formats, finding contradictory rules between items and listing those that have not moved for the longest time. It is worth asking for that review with the criteria stated explicitly, one by one, and keeping the resulting list of candidates to decide in conversation.

What must not be delegated is the consequence: discarding, merging or reordering remains a product decision with an identifiable responsible party. Block G develops that split.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Assess the health of a backlog with the criteria of the whole and with DEEP.
- Identify a backlog's dependencies and handle them in the right order.
- Detect expired items and decide when declaring backlog bankruptcy is the right move.
- Recognize a Definition of Ready turned into a gate and reframe it as a conversation guideline.
- Place non-functional requirements outside the individual story, as a constraint or a Definition of Done criterion.

Common mistakes and antipatterns

Detailing the whole backlog. Detail on distant items is wasted when the goal changes, and it also gives a false sense of control.

Using the DoR to hand work back. The team and the Product Owner share responsibility for understanding; a formal gate breaks that and creates two camps.

Counting the backlog in number of items. A backlog of three hundred items says nothing about its health; what matters is detail in proportion, absence of conflicts and age.

Putting non-functional requirements in as stories. «As a user, I want pages to load fast» has no distinguishable beneficiary and does not fit in a sprint: it is a system constraint.

Further reading

- [Pichler — Make the Product Backlog DEEP](#)
- [Cohn — Definition of Ready: why it is dangerous](#)
- [Scrum.org — Why isn't the Definition of Ready in the Scrum Guide?](#)
- [ProductPlan — When to declare backlog bankruptcy](#)
- [Scrum Manager BoK — Ready \(in Spanish\)](#)

BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

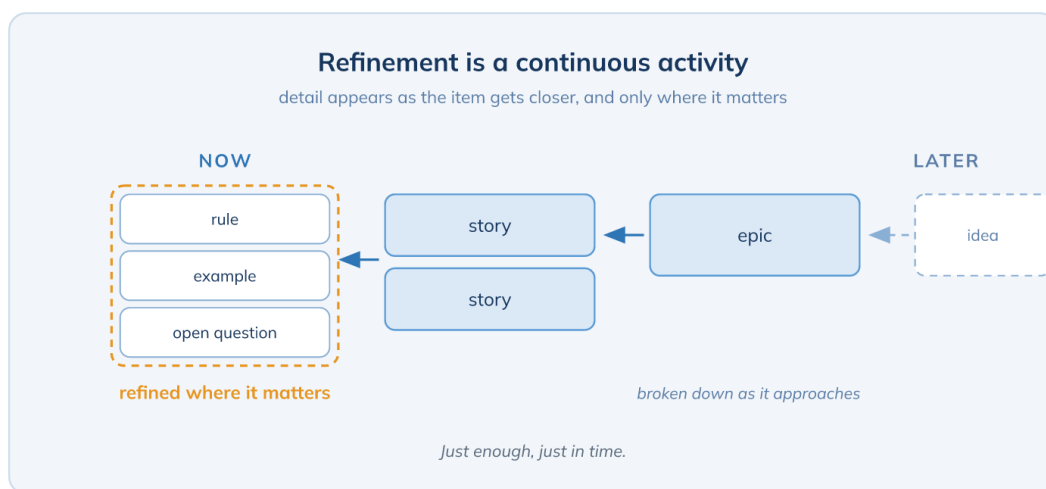
Chapter 8. Continuous refinement: neither handoff nor meeting

ESTABLISHED

Refinement as a continuous activity, its two classic degenerations and the new one brought by artificial intelligence.

Introduction

Refinement is where almost everything this guide teaches gets decided, and also where it gets lost. Its two degenerations are opposites and equally damaging: someone who refines alone and hands over the result, or a recurring meeting where the team fills in fields. In this chapter we look at what Scrum says, which organizing patterns work and how each degeneration is recognized.



8.1 What Scrum says and what it does not

The Scrum Guide defines Product Backlog refinement as the act of breaking down and further defining items into smaller, more precise ones, and adds that it is an ongoing activity. It does not describe it as an event, does not set a duration, does not say who attends and does not make it mandatory. Scrum.org has spent years dismantling the opposite myth, because mainstream practice turned it into a weekly meeting.

The Scrum Manager BoK sets the Spanish reference and adds two useful facts: the share of capacity usually spent on it (around 5 %, though it is worth measuring in your own team) and the rejection of the term *grooming*, withdrawn for its connotations.

An activity, not an event. That refinement is continuous does not mean there are no sessions. It means the sessions serve the conversation needed now, with whoever is needed, and not a calendar.

8.2 The handoff and its disguises

The **handoff** is the quietest degeneration, because it looks like efficiency. It comes in three disguises. First, the Product Owner who refines alone and delivers closed items; the team receives work, takes no part in defining it, and the questions appear during the sprint. Second, the «story author»,

someone so good at drafting that others stop trying, with the effect that their effort discourages participation. Third, acceptance criteria delivered as a contract, which turn the conversation into negotiation.

Artificial intelligence has added a fourth version, faster and harder to detect: product generates stories with a model, development interprets them with another model, and nobody has talked to anybody. The result has a better finish than ever and the same shared understanding as a forwarded email. We return to this in chapter 27.

The antidote is not organizational, it is about responsibility: understanding the work is a shared responsibility. Whoever has the need answers for it being understood; whoever will meet it answers for asking what they do not understand.

8.3 Organizing patterns that work

Short horizon. Refine what is close to being built, one or two cycles ahead, and not the whole backlog. The detail-in-proportion of chapter 7 only holds with this discipline.

Small groups. A conversation among three or four people moves; one among ten becomes a presentation. Structures that split the work into pairs or trios and then integrate (along the lines of «alone first, then in pairs, then all together») work better than the round table.

Diverge and converge. First open up interpretations, questions and edge cases; then close what needs closing. Mixing the two phases produces the feeling that the group is arguing without progressing.

Silent work first. Five minutes of individual reading and note-taking before speaking prevent the first opinion from anchoring the whole group.

Time boxes per story. Putting a limit on the conversation for each item (the Example Mapping of chapter 11 suggests about twenty-five minutes) forces a decision: it is ready, it must be split, or it must be investigated.

8.4 The named antipatterns

Refinement as a meeting. Fixed invitation, the whole team, a long agenda and no closure. Signal: nobody can say what was decided.

Data entry. The session is spent filling in tool fields. Signal: the outcome is measured in items updated.

Criteria as a contract. Criteria are closed before the conversation and invoked afterwards. Signal: the word «we agreed» shows up in reviews with a complaining tone.

Horizontal slices. Refinement produces pieces by technical layer. Signal: none of the pieces can ship on its own. Chapter 15 covers this in detail.

Technical dependencies governing the order. The backlog is ordered by what is convenient to build rather than by value. Signal: the order does not look like the product strategy.

At Vantia, the first version of the tax assistant refinement fell into the third: the Product Owner brought the criteria already drafted and the session was spent arguing about their wording. Reorganized around an example map, the same twenty minutes produced two new rules and a

question nobody had asked, namely what to do when the assistant finds no support in the regulations.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain what the Scrum Guide says about refinement and what practice attributes to it without grounds.
- Organize refinement as a continuous activity, with a short horizon and small groups.
- Recognize the four disguises of the handoff, including the one introduced by AI assistance.
- Name the five refinement antipatterns from their signals.
- Facilitate a refinement conversation with prior individual work and a time limit per item.

Common mistakes and antipatterns

Refining the whole backlog. Detail on distant items expires before it is used; refining far ahead is the most common way of working hard without progressing.

Inviting the whole team to every conversation. A productive conversation needs the necessary perspectives, not full attendance; chapter 9 develops this.

Measuring refinement by items updated. What has to be measured is whether the team can start without blocking questions, and that cannot be seen in the tool.

Refining without deciding. A conversation that does not end in «it is ready», «it must be split» or «it must be investigated» will have to be held again from scratch.

Further reading

- [The Scrum Guide 2020](#)
- [Scrum Manager BoK — Product Backlog maintenance \(in Spanish\)](#)
- [Scrum.org — What is a user story?](#)
- [Ferrari et al. — Ambiguity and tacit knowledge](#)

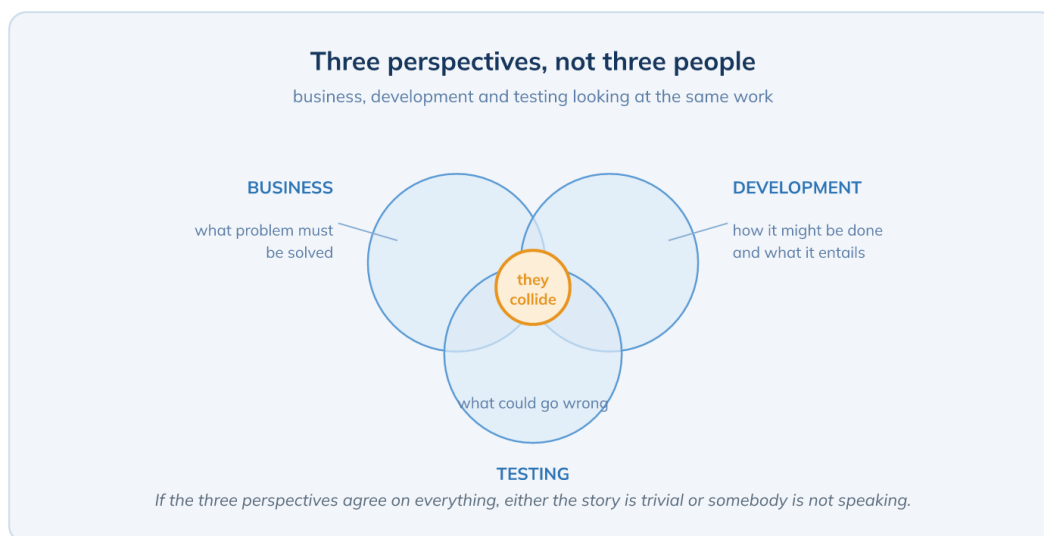
BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

Chapter 9. Who talks and in what format: Three Amigos and story-writing workshops · ESTABLISHED

The three necessary perspectives, the story-writing workshop and the criterion that sends high uncertainty to a spike.

Introduction

A refinement conversation works when the perspectives that are going to collide are present. If one is missing, the collision happens anyway, only later and with code in the way. In this chapter we look at the practice that names that requirement, the workshop format that organizes it, and the scale that decides when what is needed is not conversation but investigation.



9.1 Three Amigos: three perspectives

George Dinwiddie proposed the name in 2009, in an article whose opening question was what to do if you do not automate acceptance tests. The answer was to bring three points of view together before building: **business** (what problem must be solved), **development** (how it might be solved and what it entails) and **testing** (what could go wrong and how we will know). When the work calls for it, design, operations, security or compliance join in.

The nuance most often lost is in the name: they are three **perspectives**, not three **people**. In a small team one person can bring two; in delicate work the testing perspective may need two specialists. The practice also travels under other names, Triad (Ken Pugh) and the Power of Three (Lisa Crispin and Janet Gregory).

What it is really for. Not to distribute tasks, but to make different interpretations of the same sentence collide before it costs money. If the three perspectives agree on everything, either the story is trivial or somebody is not speaking.

Its mistakes are documented and there are four: limiting it literally to three people, expanding it to the whole team (whereupon it becomes a meeting again), turning it into a recurring formal event, and arriving with the scenarios already closed, which is the conversational version of the handoff.

9.2 The story-writing workshop

When a set of items has to be tackled rather than a single story, the right format is the writing workshop. An agenda that works has seven steps: review the priority items and their goal; identify users and expected outcomes; surface assumptions and hidden questions; rewrite what does not hold; split what does not fit; sketch acceptance criteria with examples; and close with next steps and owners.

The workshop is not a collective drafting session, which produces slow, committee-written prose. It is a decision session: what goes in, what gets split, what gets investigated and what gets discarded. The writing can be left to one person afterwards, with the agreement already made.

For a single story, the twenty-five-minute format of chapter 11, Example Mapping, is more effective than any long workshop, and it is worth mastering both.

9.3 Questions before answers: deliberate discovery

Dan North called **deliberate discovery** the discipline of actively seeking what is not known, instead of waiting for it to show up. His argument is that ignorance is the greatest cost in development, and that there is a particularly expensive variety, **second-order ignorance**: not knowing that you do not know.

Liz Keogh turned that into a practical complexity scale, running from «we all know how to do this» to «nobody has ever done this», with intermediate steps («somebody in our organization has done it», «somebody in the world has done it», «somebody has tried it»). The scale is used to decide what to do with each story: on the low steps, talk and build; on the high ones, stop talking and buy information with a **spike** or an experiment.

It is a criterion that saves whole sessions. When the group has spent twenty minutes speculating, what is missing is not more conversation: it is a fact nobody has.

9.4 Deferring the decision while options remain open

The last piece of this chapter is about timing. While options remain open and the cost of waiting is low, deciding later produces better decisions, because they arrive with more information. That holds for the detail of a story, for its slicing and for its criteria: closing them as soon as they are stated is the surest way to fix assumptions that have not been checked.

The limit is blocking: deferring a decision that prevents progress is not prudence, it is indecision. The operating rule is to identify the last responsible moment for deciding and to reach it with the options still alive.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Convene a refinement conversation with the necessary perspectives and justify who is missing or who is surplus.

- Facilitate a story-writing workshop with its agenda and close it with decisions and owners.
- Place a story on the complexity scale and decide whether to talk, investigate with a spike or experiment.
- Recognize the four frequent mistakes with Three Amigos.
- Identify the last responsible moment for closing a decision and hold the deferral until then.

Common mistakes and antipatterns

Turning it into a ritual with three fixed people. The composition depends on the work; freezing it turns the practice into a committee and leaves necessary perspectives out.

Arriving with the scenarios closed. If whoever convenes brings the answer, the session serves to validate rather than to discover, and the three perspectives stop contributing.

Speculating instead of investigating. When uncertainty is second-order, no conversation resolves it: information has to be bought.

Closing the criteria in the first session. Criteria agreed too early fix unchecked assumptions and then turn out to be expensive to change.

Further reading

- [Agile Alliance — Three Amigos](#)
- [Keogh — Estimating Complexity](#)
- [Extreme Programming — Create a Spike Solution](#)
- [Cohn — User Stories \(reference guide\)](#)

BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

Chapter 10. Acceptance criteria: formats, limits and confusions ·

ESTABLISHED

What they are, what formats they take, how they differ from the Definition of Done and how probabilistic behavior is confirmed.

Introduction

Acceptance criteria are the third C, the confirmation, put in writing. They are also the part of the story most often confused with other things: with the Definition of Done, with non-functional requirements and, in organizations with a poor relationship between business and development, with a contract. In this chapter we put them in their place.

10.1 What they are and what formats they take

An acceptance criterion is an **observable condition** that will tell us the need is met. Observable is the key word: if it cannot be checked by looking at the product's behavior, it is not a criterion.

There are four formats in common use, and it pays to choose per case. The **list of conditions** is the simplest and serves most stories. The **«Given / When / Then» scenario** organizes cases with a precondition, an action and a result, and is the one behavior-driven development popularized. **Rules with examples** work better when behavior depends on several conditions, and they are the natural format of Example Mapping. The **table** is unbeatable when there are combinations of data and results.

Choosing the format. If the scenario forces you to write six steps and an artificial precondition, the wrong format is the scenario. Form serves the conversation, not the other way round.

10.2 Acceptance criteria and the Definition of Done

This is the most widespread confusion and it has a clear answer. **Acceptance criteria** are specific to one item and say what *this* must do. The **Definition of Done (DoD)** is common to the whole increment and says what finishing *anything* means in this team (tests, review, documentation, accessibility, security, deployment). Scrum requires the second and does not mention the first.

Mixing them causes two kinds of damage at once: criteria that repeat in every story what the DoD already requires, and a DoD that changes with every story and stops being a stable agreement. The practical rule: if something must hold for everything you deliver, it goes in the DoD; if it belongs to this need, it is a criterion.

Non-functional requirements are distributed by the same criterion. General performance, accessibility or encryption belong to the DoD or to a cross-cutting constraint. A threshold specific to this functionality, on the other hand, is an acceptance criterion of this story.

10.3 How many criteria, and what an excess signals

There is no number, but there is a signal. When a story accumulates many criteria (say, more than seven or eight), it is almost always saying one of three things: that it actually contains several stories,

that things belonging to the DoD are being written as criteria, or that somebody is trying to close in writing what should be talked through.

It also pays to tell a criterion from its example. «Every answer includes the consultable regulatory reference» is a criterion; «when asked about deducting a business meal, the answer cites the applicable article» is an example that confirms it. Criteria are kept few and canonical; examples can be several. In chapter 29 we will see that this distinction becomes critical when the work is executed by an agent.

10.4 When behavior is probabilistic

Some functionality does not admit a binary criterion because its behavior is statistical, and the answers of an assistant based on a language model are today's example. «The assistant answers correctly» is not checkable as a criterion of a story, because there is no single right answer.

The way to confirm it is different: you define a **rubric** for what makes a good answer, you build an **evaluation set** with cases that pass, cases that fail and edge cases, and the criterion becomes a **threshold** over that set. At Vantia, the tax assistant's rubric has four points (correct per the regulations, with its consultable reference, understandable to someone who is not an accountant, and informative without replacing the advisor's counsel), and the set includes a deliberate edge case: a doubtful query where the right answer is not a verdict but explaining the criterion in the regulation and handing off to the advisor.

The full toolkit for evaluating AI-based functionality belongs to the Product Owner core training and to the AI applied to agile work skill; here it is enough to know that the acceptance criterion changes form, not function.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Write observable acceptance criteria and choose the format that fits each case.
- Separate with judgment what goes into acceptance criteria and what goes into the Definition of Done.
- Place non-functional requirements in the right home according to their scope.
- Read an excess of criteria as a signal of pending splitting or avoided conversation.
- Express the confirmation of probabilistic functionality with a rubric, an evaluation set and a threshold.

Common mistakes and antipatterns

Repeating the DoD in every story. It adds noise to every item and switches off the shared agreement; when something holds for everything, its home is the Definition of Done.

Writing criteria that dictate the interface. «A blue button at the top right» is not an observable condition of the need, it is a design decision put in the wrong place.

Criteria without a single example. An abstract criterion survives opposite interpretations; the concrete example is what makes interpretations collide.

Covering only the happy path. Edge and negative cases are where the business rules nobody has stated are hiding.

Asking for a binary criterion for probabilistic behavior. It produces endless arguments in the review; what is needed is a threshold over an agreed set.

Further reading

- [Agile Alliance — Given-When-Then](#)
- [Scrum Manager BoK — Acceptance criteria \(in Spanish\)](#)
- [Scrum Manager BoK — Definition of Done \(in Spanish\)](#)
- [Cohn — Non-functional requirements as user stories](#)
- [Skill Arena — Product Owner](#)

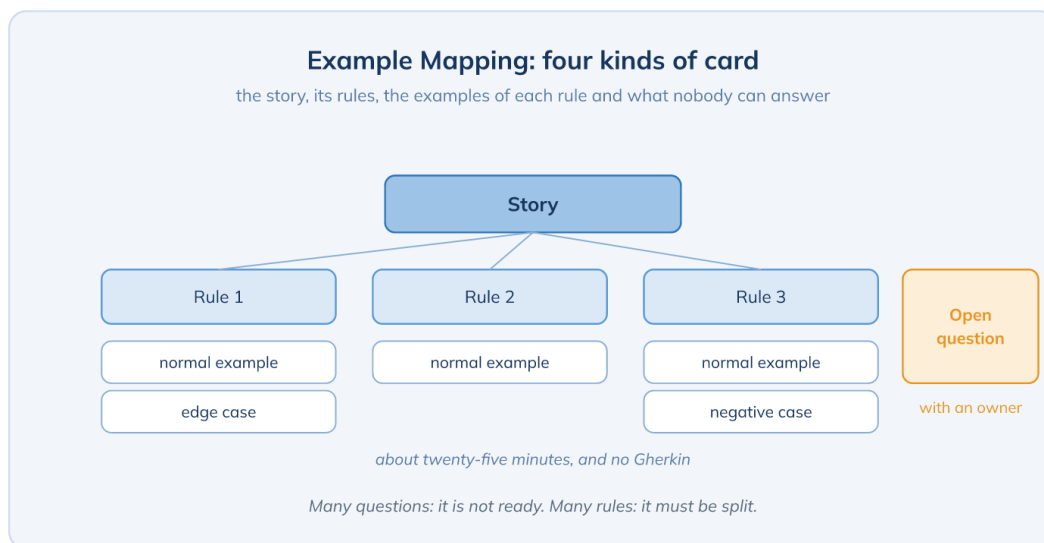
BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

Chapter 11. Rules, examples and Example Mapping · CONSOLIDATING

How the rules of a story are found, which examples are needed and how the resulting map is read.

Introduction

This chapter contains, probably, the technique with the best effort-to-result ratio in the whole guide. Example Mapping organizes the conversation about a story in twenty-five minutes and produces four things at once: its rules, the examples that pin them down, the questions nobody can answer and a diagnosis of whether the story is ready.



11.1 Business rules, explicit and implicit

A **business rule** is a condition the behavior must always respect. Some are written down somewhere (the regulations, a contract, a company policy); others live in the head of whoever knows the domain and go unsaid because they seem obvious. The second kind is what sinks releases.

Alongside the rules you have to surface their **exceptions** and their **conditions**: when the rule does not apply, and what the system does then. The question that works best for finding them is a single one and worth memorizing: **is there any context in which this would be different?**

11.2 Examples that illustrate rather than exhaust

An **example** is a concrete case, with concrete data, that shows the rule in action. Its function is to make disagreement visible: two people can nod at an abstract rule and disagree the moment a data point is put on the table.

Three kinds of example are needed, and in this order: the **normal case**, the **edge case** (the value at the boundary, the maximum, zero, empty) and the **negative case** (what must not happen, or the expected error). What is not needed is exhausting the combinations: examples illustrate so that the team understands the rule, and exhaustiveness belongs to test design.

Illustrating versus exhausting. Three well-chosen examples teach the rule; twenty examples hide it and turn the conversation into a spreadsheet review.

11.3 Example Mapping, step by step

Matt Wynne published the technique in December 2015 and its protocol is deliberately simple. Four card colors are used: **yellow** for the story, **blue** for its rules, **green** for the examples of each rule and **red** for the questions nobody can answer. The yellow one goes on top, and the group adds rules below it, examples under each rule and questions to one side, in silence or out loud, as needed.

The stated duration is about twenty-five minutes for a well-understood, well-sized story, and that figure is part of the technique: if it does not close, that fact is itself information. And there is an express prohibition: no Gherkin is written during the session. Precise formulation comes later, and mixing it with discovery stalls the conversation in arguments about syntax.

It is worth practicing often. Wynne recommends doing it every couple of days with upcoming stories rather than saving it for the difficult ones: it is in the ones that looked easy that the missing rule usually appears.

11.4 Reading the map: four diagnoses

The resulting map reads as a measuring instrument, and this is what makes it valuable beyond the conversation:

Many red cards (questions): the story is not ready. Do not estimate it or pull it into the sprint; go find answers, and maybe a spike.

Many blue cards (rules): the story is too large. Each rule or group of rules is a candidate story of its own, and block D says how to slice it.

One rule with a great many examples: it is probably not one rule but several badly stated ones. It is worth breaking down.

Few rules, clear examples and no questions: the story is ready, and it already has its acceptance criteria written.

At Vantia, the tax assistant's map produced three rules (answer with support in the regulations and in the user's own data; always include the consultable reference; acknowledge the limit and hand off to the advisor when there is no support), five examples and one red question nobody could close in the session: what happens when the freelancer insists after a handoff. That question became the second item in the backlog.

11.5 Variants worth knowing

Feature Mapping (John Ferguson Smart) adds the user journey and the tasks, and works well when the story has a flow with several steps. **OOPSI** (Jenny Martin) walks through outcomes, outputs, processes, scenarios and inputs, and serves to frame a set before going down to the stories. Both are cousins of Example Mapping and share its central idea: discover with concrete examples before formulating.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Find the business rules of a story, with their exceptions, using the question that locates different contexts.
- Choose normal, edge and negative examples, and tell illustrating from exhausting.
- Facilitate an Example Mapping session with its four card types and its time limit.
- Read the resulting map and decide whether the story is ready, must be split or must be investigated.
- Turn the map's rules and examples into acceptance criteria without rewriting them.

Common mistakes and antipatterns

Writing Gherkin in the discovery session. The discussion shifts to syntax and the missing rules never appear; formulation comes afterwards.

Saving the technique for difficult stories. The ones that look easy are the ones hiding the unstated rule, and they are the cheapest to map.

Ignoring the red cards. A question with no owner and no deadline reappears mid-sprint as a blocker.

Chasing exhaustiveness. A map with thirty examples stops being a conversation instrument and becomes a specification nobody reviews.

Further reading

- [Wynne — Introducing Example Mapping](#)
- [Smart — Feature Mapping](#)
- [Adzic — Specification by Example \(resources\)](#)
- [Cucumber — Gherkin reference](#)

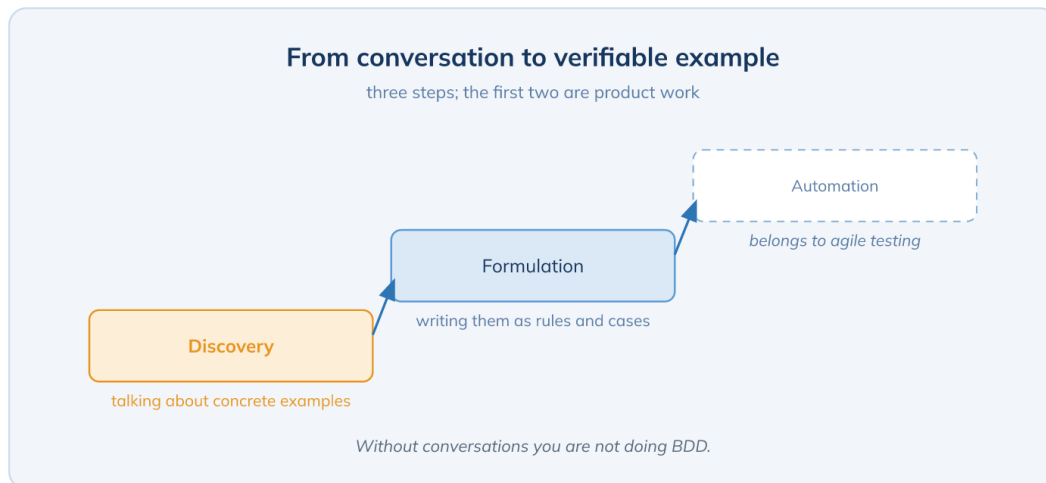
BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

Chapter 12. From examples to verifiability: BDD without a tool · ESTABLISHED

The ladder of discovery, formulation and automation, and why the first two steps are product work.

Introduction

Behavior-driven development suffers the mirror image of the misunderstanding that affects stories: it gets confused with a testing tool. In this chapter we recover its product side, which is its first two steps, and leave the third where it belongs.



12.1 Three steps and who climbs each one

The BDD community arranges the practice in three activities. **Discovery**: talking with concrete examples to understand the behavior sought (that is chapter 11). **Formulation**: writing those examples so that anyone understands them and they serve as criteria. **Automation**: turning them into tests that run.

The first two steps are product work and require no tool: a whiteboard and a conversation are enough. The third is engineering work and belongs to agile testing, with its own toolkit. This guide teaches the first two and refers the third.

The order matters. Starting with automation produces hundreds of scenarios nobody discussed, whose maintenance costs more than they contribute. It is BDD's classic failure mode.

12.2 What its authors say

They are worth quoting because they are unequivocal. Liz Keogh: if you are not having conversations, you are not doing BDD at all. Aslak Hellestøy, creator of Cucumber: BDD is not test automation. Dan North, who coined the term, presented it as a way of talking about expected behavior using examples, and his article «What's in a Story?» remains the best short introduction.

The Scrum Manager BoK carries that reading and places North at its origin. The practical conclusion for story work is that the «Given / When / Then» vocabulary is useful as a way of organizing an example, and its value does not depend on later automation.

12.3 Specification by example and living documentation

Gojko Adzic systematized in *Specification by Example* the idea of using agreed examples as the specification of behavior. Its most valuable side effect appears when those examples are automated: the documentation stops aging, because a description that runs and fails announces that it is no longer true. That is what is called **living documentation**.

For product work, the interesting point is the previous one: agreed examples are already the specification, automated or not. And that connects with chapter 13, because they are also the only thing worth keeping from a refinement conversation.

12.4 Writing readable scenarios

When an example is formulated as a scenario, there are two styles and one works better. The **imperative style** describes interface steps («when I click New invoice, when I type 100 in the amount field, when I press Save...»): it breaks with every interface change and does not say what behavior is sought. The **declarative style** describes the intent («when I issue an invoice for 100 euros to a customer in another EU country»): it survives changes and it reads.

Three practical rules: few lines per scenario, one behavior in each, and concrete data instead of descriptions («a customer with a Portuguese tax number» is better than «an intra-EU customer»). Scenarios written this way serve as acceptance criteria without rewriting, and they are the starting point for the machine verification of chapter 29.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the three steps of behavior-driven development and say which ones are product work.
- Formulate an agreed example as a readable scenario, in declarative style and with concrete data.
- Tell a scenario that serves as an acceptance criterion from one that merely describes an interface.
- Argue why conversation is the condition of the method, and not the tool.
- Explain what living documentation is and what keeps it true.

Common mistakes and antipatterns

Starting with the tool. Installing the automation framework before having any conversation produces ownerless scenarios and a maintenance load that ends up abandoned.

Writing imperative scenarios. Interface steps turn every redesign into a batch of broken scenarios without the behavior having changed.

Translating every criterion into a scenario. Not every criterion needs that shape; forcing it adds artificial preconditions and lengthens the read.

Calling automation BDD. It is the misunderstanding its own authors have spent fifteen years correcting, and it drags the team into skipping discovery.

Further reading

- [North — What's in a Story?](#)
- [Keogh — A Third Conversational Pattern in BDD](#)
- [Rose — User stories and BDD](#)
- [Scrum Manager BoK — Behavior-Driven Development \(in Spanish\)](#)
- [Adzic — Specification by Example \(resources\)](#)

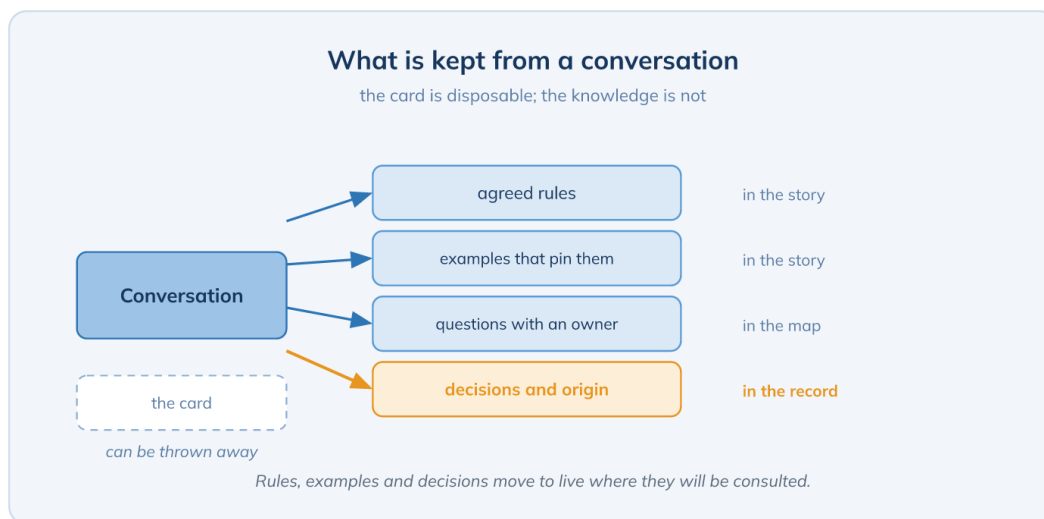
BLOCK C · THE CONVERSATION AND ITS CONFIRMATION

Chapter 13. Preserving the context of the conversation · CONSOLIDATING

What is worth keeping from a refinement session, where to put it and how the tension with discarding delivered stories is resolved.

Introduction

If the value of a story lies in the conversation, a practical problem remains: the conversation is forgotten. Six months later nobody remembers why the assistant hands off to the advisor in doubtful cases, and the rule gets questioned again or, worse, changed without knowing what it cost to agree. In this chapter we look at what to keep and where.

**13.1 Four things are worth keeping**

The agreed rules. They are domain knowledge and outlive the story that discovered them.

The examples that pin them down. Without them, the rule admits interpretations again.

The open questions, with an owner. A question with no owner is a lost question.

The decisions and their provenance. What was decided, on what evidence, which alternatives were discarded and who answers for it.

What is not worth keeping is almost everything else: the narrative of the session, the intermediate opinions and the discarded drafts.

13.2 Where to put it

In the story itself go the rules and the examples that confirm them, because that is where they will be read.

In the example map stays the record of the conversation, if the photo of the map with its pending red cards is kept.

In the living documentation stay the automated examples, which also warn when they stop being true (chapter 12).

In a **product decision record** go the decisions that outlive a single story. This last piece is a **Scrum Manager proposal**, and it is worth saying so: we bring it by analogy with architecture decision records, which software engineering has used since 2011 and which consist of half a page per decision, with context, options, decision and consequences. The Product Owner 2.0 core guide includes a complete example of that record applied to product.

Half a page per decision. The format matters less than the discipline: record only the decisions that were hard to make, and always in the same place. A record that grows without criteria does not get consulted.

13.3 The tension with discarding delivered stories

Agile tradition recommends throwing the cards away when the story ships, and it is right: keeping an archive of completed stories produces a graveyard nobody consults and gives a false sense of traceability.

The tension is resolved by telling two things apart. **The card is disposable**; it was a reminder for a conversation that already happened. **The knowledge is not**: the rules, the examples and the decisions move to live where they will be consulted, which is not the backlog. With that split, throwing away the story loses nothing.

13.4 The context that only exists in a chat

AI assistance has created a new place where context gets lost: the private conversation with the model. Whoever drafts a story with help usually builds valuable context along the way (assumptions made, alternatives discarded, questions the model raised), and that context does not reach the team, because what reaches the team is the finished story.

The practice that corrects it is simple and worth adopting from the start: bring **the assumptions and the questions** to refinement, not just the result. Put another way, if the conversation with the model produced five decisions, the team should see all five, even if the final text does not mention them. Chapter 26 treats this as part of provenance.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Decide what is kept from a refinement conversation and what is discarded.
- Place each kept piece where it will be consulted.
- Write a half-page product decision record, with its provenance.
- Explain why the card is disposable and the knowledge is not.
- Make visible to the team the context built in a private conversation with a language model.

Common mistakes and antipatterns

Keeping the whole story «for traceability». It produces an archive nobody reads and displaces the genuinely reusable knowledge, which is the rules and the decisions.

Recording every decision. A record with hundreds of entries stops being consulted; only decisions that were hard to make and that someone will question deserve an entry.

Leaving questions without an owner. A red card with no owner and no date reappears in the sprint as a blocker.

Presenting the result without the assumptions. The team inherits decisions it does not know were made, and finds out when one of them turns out to be wrong.

Further reading

- [Adzic — Specification by Example and living documentation](#)
- [Wynne — the example map as a record of the conversation](#)
- [Skill Arena — Product Owner](#)
- [Scrum Manager BoK — AI as a teammate \(in Spanish\)](#)

BLOCK D · SPLITTING WHILE KEEPING VALUE

Chapter 14. Why, how much and when to split · ESTABLISHED

The underlying reason for small batches, size measured rather than inherited, and the moment when splitting is worthwhile.

Introduction

Splitting is the most requested skill in story work and the least taught. Before looking at how it is done, it is worth being clear about why it is done, because a team that splits without understanding the reason ends up producing small pieces that are good for nothing. In this chapter we cover the why, the how much and the when; the next three chapters cover the how.

14.1 Why small batches

The reason is not planning convenience but learning. A small batch shortens the time between deciding and knowing whether the decision was good, and that cycle is the only thing that turns delivery into knowledge. Donald Reinertsen systematized it for product development: large batches accumulate risk, delay the discovery of errors and hide what does not help.

The DORA program supports it with data year after year, and places working in small batches among the capabilities associated with organizational performance. For story work, the practical consequence is three concrete benefits: earlier feedback, less risk per release and the possibility of **deprioritizing or discarding** a part when it turns out not to matter. That third benefit is the most forgotten and the one that best tells a good slice from a bad one.

The discard test. A slice that does not allow you to give up any of its pieces has split nothing: it has distributed the same work across more cards.

14.2 How much: from the inherited band to your own measurement

The figures in circulation are a band of heuristics from different eras: the original INVEST text spoke of a few person-weeks; Cohn and the Humanizing Work school work with the idea of fitting six to ten stories into a sprint; SAE speaks of a few days or less; and practice in product teams with frequent deployment runs from half a day to two days.

Rather than adopting one of those figures, it is worth replacing them with your own team's measurement. The Kanban community has explicitly separated two ideas that often get confused: **right-sizing**, which means checking whether an item fits within the timeframe the team has committed to sustain, and **same-sizing**, which means chopping everything up until it measures the same. The first is useful; the second is a ritual.

Your own measurement comes from looking at what has already shipped: how long the last thirty items took from start to finish. With that distribution, the question about a new story stops being how many points it has and becomes whether it looks like the ones finished in under three days.

14.3 The lower limit

There is a size below which splitting makes things worse. When a piece can no longer state what changes for anyone, it has stopped being a slice of value and has become an implementation detail in story format. The signal is twofold: nobody outside the team notices the release, and the cost of managing the card (refining, ordering, reviewing) starts to look like the cost of doing it.

When that happens, the answer is to **recombine**: put the pieces back into an item that does express an observable change and split it again with a different criterion. We look at that in chapter 17.

14.4 Epic, theme and feature: containers, not categories

The vocabulary of sizes is useful for conversation and dangerous when taken for a taxonomy. An **epic** is a large item that will have to be split; a **theme** groups items related by subject; a **feature** is a product capability that usually contains several stories. Tools turn them into types with rules and mandatory hierarchies, and that is where the problem of chapter 18 begins.

Cohn sums it up well: they are labels of convenience, and the question that matters is not what type an item is but whether it fits in one cycle and whether it expresses value. The Scrum Manager BoK documents the theme, epic, story and task hierarchy as a convention, and it is worth using in that spirit.

14.5 When to split

The moment is **one or two cycles before** the story reaches the top of the backlog. Splitting earlier is wasted work when the goal changes; splitting during planning is late, because the conversation that discovers the slice does not fit in that slot.

The opposite antipattern is splitting the whole backlog in advance, which produces hundreds of small pieces ordered by build convenience and with no visible relationship to value. It is the most effective way of losing the product narrative, and block E is devoted to recovering it.

It is also worth telling apart two operations that get confused: **splitting into stories** (dividing value into slices of value, which is what this block is about) and **decomposing into tasks** within the sprint (distributing the technical work of a story, which is the development team's business). The second does not produce backlog items.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain with evidence why small batches improve learning and reduce risk.
- Obtain your own team's size measurement from what has already shipped, rather than adopting an inherited figure.
- Tell right-sizing from same-sizing.
- Recognize the lower limit of splitting and recombine when it has been crossed.
- Decide the moment to split and tell splitting into stories from decomposing into tasks.

Common mistakes and antipatterns

Splitting so it fits in the sprint. Size is a consequence of the slice, not its goal; slicing by capacity produces pieces with no value of their own.

Making all sizes equal. It turns splitting into paperwork and wastes the information that real size variation gives you.

Splitting the whole backlog in advance. Detail expires, the narrative is lost and effort goes into pieces that will never be built.

Calling the tasks of a story stories. It inflates the backlog with items that have no beneficiary and makes the order stop expressing value.

Further reading

- [DORA — Working in small batches](#)
- [ProKanban — Right-sizing is not the same as same-sizing](#)
- [Kanban Guides — Open Guide to Kanban](#)
- [Cohn — Epics, features and themes](#)
- [Scrum Manager BoK — Epic \(in Spanish\)](#)

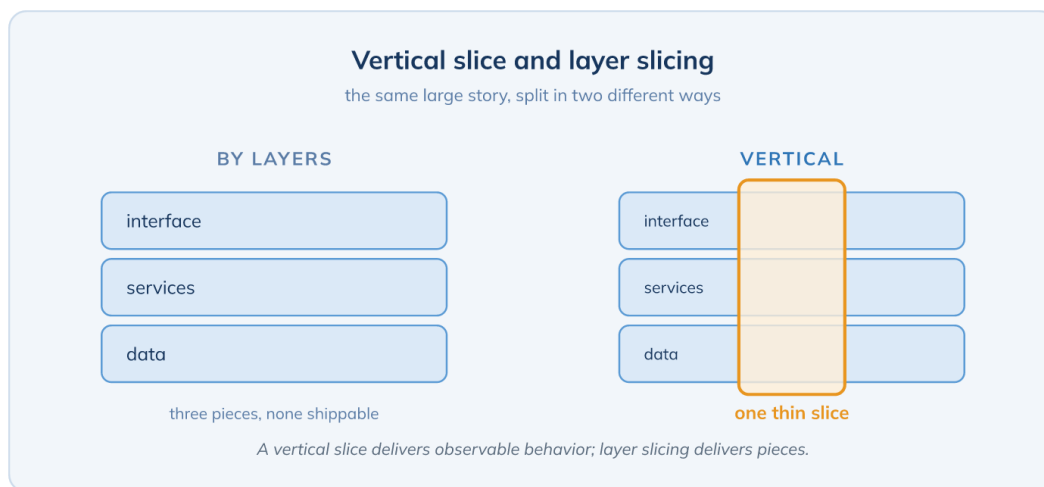
BLOCK D · SPLITTING WHILE KEEPING VALUE

Chapter 15. Vertical slicing and legitimate technical slices · ESTABLISHED

The slice that cuts through the system versus layer slicing, and the three technical slices that are justified.

Introduction

The most widespread splitting mistake has a name and a recognizable shape: slicing by technical layers. It produces three pieces (interface, services, data) of which none can be shipped or valued separately, and it forces all three to be finished before anyone notices anything. In this chapter we look at the alternative and at the cases where a technical slice is justified.



15.1 The vertical slice

A **vertical slice** delivers an observable change in behavior and cuts through as many layers as it needs: if it needs interface, service and data, all three go inside. The slice can be extremely thin (one case, one data type, one path) and it is still vertical as long as somebody can use it and notice it.

The advantage is not aesthetic. A vertical slice can be deployed, measured, discussed with whoever asked for it and discarded; three horizontal layers can only be shown in an internal demo. Chapter 4 anticipated this property with the letter Wake added to INVEST, being end-to-end.

How to recognize it. One question is enough: could somebody outside the team try this and say whether it is useful? If the answer is no, the slice is horizontal, even if the card has a story template on it.

15.2 The walking skeleton and the tracer bullet

There is a legitimate, well-documented exception for the first slice of a product or a large feature. The **walking skeleton**, from Alistair Cockburn, is a minimal implementation that performs a function end to end, cutting through every component with the bare minimum. Its value lies in integrating the architecture before there is code to rewrite.

The **tracer bullet**, from Hunt and Thomas, is the same idea from the code's point of view: build the complete path with reduced functionality to check that the pieces fit and to have something to aim with afterwards.

Gojko Adzic added a practical corrective worth knowing, because the pure skeleton usually ends up in a demo that serves nobody: put it on crutches. That is, complete the slice with whatever it takes (data loaded by hand, a manual process behind it, a provisional screen) so that somebody can **actually use it** rather than merely watch it work.

15.3 The three legitimate technical slices

The spike. When uncertainty is about knowledge rather than scope, the right move is to buy information with work bounded in time and with an explicit question. It is a last resort, not a companion to every story: one spike per item is a signal that refinement is not doing its job.

The enabler story. Work that prepares the ground for several later stories (a migration, an integration, an infrastructure change). SAFe institutionalizes it in four types, and outside SAFe it is worth stating plainly, with its beneficiary and its economic justification.

The walking skeleton, already covered, for the first slice of something large.

What these three have in common, and what tells them apart from layer slicing, is that they are stated honestly: nobody pretends they are user value. Chapter 23 develops how to write them without fake users.

15.4 The open debate: where to start

When a vertical slice has to be built, one decision remains with two schools and no clear winner.

Interface first, with simulated data or manual processes behind it: it allows validating with users very early and discovering that the need was something else, at the cost of work that will have to be replaced. **Services first**, with the functionality hidden behind a feature flag: it secures the foundations and avoids rework, at the cost of taking longer to learn from the user.

The choice depends on which risk dominates. If the greater risk is not knowing whether the solution helps, it pays to start with what can be shown. If the greater risk is technical, it pays to start with what can be tested. At Vantia the team chose the former in its most extreme form: before building the assistant, it set up a **Wizard of Oz** with advisors answering at night behind an assistant interface, and in two weeks obtained the real catalog of queries, which reordered the whole backlog.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Tell a vertical slice from layer slicing with the user question that tests it.
- Build the first slice of a large feature as a walking skeleton, with the crutches it needs to be useful.
- Justify a spike with its question and its time box, and not turn it into routine.
- State an enabler story with its beneficiary and its economic reason.
- Choose between starting with the interface or with the services according to which risk dominates.

Common mistakes and antipatterns

Slicing by layers and calling it splitting. It is the first of the five splitting mistakes Cohn documents: it produces pieces nobody can value and forces all of them to ship.

Shipping the skeleton without crutches. A slice that can only be demoed internally produces no real user learning.

One spike per story. It turns investigation into a toll and hides the fact that refinement is not resolving the uncertainty.

Technical stories with invented users. «As a server, I want...» communicates nothing and makes ordering impossible; chapter 23 gives the honest alternatives.

Further reading

- [Adzic — Forget the walking skeleton, put it on crutches](#)
- [Hunt and Thomas — Tracer bullets and prototypes](#)
- [Humanizing Work — Dealing with technical stories](#)
- [Cohn — Five story-splitting mistakes](#)
- [SAFe — Enablers](#)

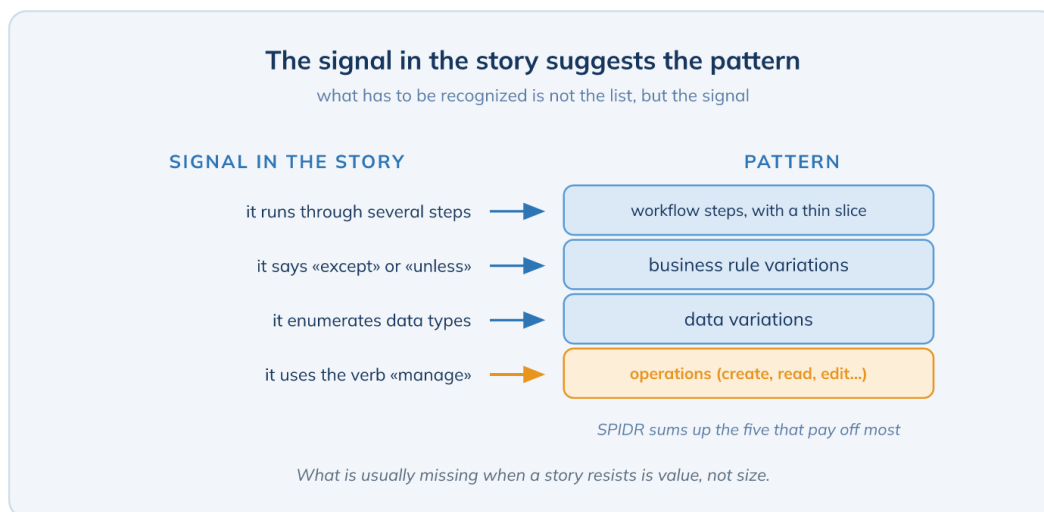
BLOCK D · SPLITTING WHILE KEEPING VALUE

Chapter 16. The catalog of splitting patterns · ESTABLISHED

Thirteen patterns with their provenance and the signal in the story that suggests each one.

Introduction

The splitting patterns are the craft's toolkit, and their value lies less in the list than in knowing which signal in the story suggests each one. In this chapter we go through them grouped by that signal, with their provenance, and at the end we cross-reference them with the strategies of the Scrum Manager basic manual this guide supersedes, so that anyone coming from there finds their equivalence.



16.1 When the story describes a flow

Workflow steps. A story that runs through several steps («request, review, approve, notify») can be split by steps, with one condition almost everyone breaks: the first slice must cut through the complete flow in the simplest possible way, and the following ones fatten the steps up. Splitting by steps and delivering the first one complete produces useless pieces, which is why Humanizing Work titled their article on the subject by saying that most people get it wrong.

Paths and interfaces. If there are several alternative paths (by card or by transfer; from web or from mobile), each path is a slice. It is the most productive pattern when the story hides an «or».

16.2 When the story hides variety

Business rule variations. A rule with exceptions («except for intra-EU customers») can be split into the general rule first and the exceptions later. The signal is the word «except», «unless» or «depending on».

Data variations. If behavior changes with the type of data (an invoice type, a tax regime, a language), each variety is a slice. The signal is enumerations.

Operations. The four operations on an entity (create, read, update, delete) rarely carry the same value: almost always one of them solves the problem and the others are convenience. The signal is the catch-all verb «manage».

User segment. Serving one segment first (Vantia's night-time freelancers, rather than all users) allows earlier learning with less scope. The signal is a user described in a generic plural.

16.3 When complexity is unevenly distributed

Simple first, complex later. Implement the central case and defer the rare ones. The signal is a percentage: 80 % of cases are alike.

Major effort isolated. When one specific part concentrates almost all the work, it pays to separate it so the rest can ship and the decision about it can wait. The signal is an estimate dominated by a single element.

Deferred system qualities. First it works, then it works fast (or accessible, or at scale). Deferring the quality is legitimate if made explicit and with a date; the risk is that the deferral becomes permanent.

Capacity and volume. Start with a limit (ten invoices, a hundred users) and raise it later. The signal is a volume requirement that multiplies complexity.

16.4 When what is missing is information or value

Manual before automated. Solve it first with a person behind the scenes and automate later, once you know what to automate. It is the pattern of Vantia's Wizard of Oz.

Learn before earn. Adzic formulates it as separating learning from earning: if the story mixes the two, it pays to deliver first the part that teaches something, even if it produces no revenue, and then the part that does.

Break out a spike. When uncertainty prevents sizing, the question is extracted as bounded work. It is the last resort of chapter 15.

SPIDR, to remember five. Mike Cohn grouped the five highest-yield patterns into a handy acronym: **S**pike**, **P**ath**, **I**nterface**, **D**ata** and **R**ules**. It works as a first pass before resorting to the full catalog.

16.5 Equivalence with the Scrum Manager basic manual

Anyone who has worked with the Scrum Manager basic manual «User stories» will recognize its eleven strategies here under other names. The mapping is direct: splitting by workflow steps, by business rules, by data types or parameters, by operations and by roles corresponds to the patterns in 16.1 and 16.2; splitting by happy and unhappy flow is a case of rule and path variations; splitting by input options or platforms is the interface pattern; splitting by test cases or scenarios is equivalent to data variations guided by the examples of chapter 11; optimizing now or later is the deferral of qualities; browser compatibility is a particular case of interfaces; and strategy zero, building a spike, is the break-out-a-spike pattern.

The unified catalog presented here is an **editorial synthesis** over patterns established one by one, and it is worth stating it with the Agile Alliance's caveat: there is surprisingly little consensus on how to split stories. That is why the next chapter devotes its space to the selection criterion, which is what really distinguishes someone who knows how to split.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Recognize in a story the signal that suggests each splitting pattern.
- Apply the workflow-steps pattern with a first slice that cuts through the whole flow.
- Use SPIDR as a first pass and turn to the full catalog when it is not enough.
- Explain the provenance of the patterns and their degree of consensus.
- Translate the strategies of the Scrum Manager basic manual into the canonical patterns.

Common mistakes and antipatterns

Splitting by steps and delivering the first one complete. It produces a piece that does not cut through the flow and that nobody can judge; the first slice must reach the end, however minimally.

Splitting «manage» into four equal operations. They almost never carry the same value: it pays to deliver the one that solves the problem and decide about the rest afterwards.

Deferring qualities without a date. A deferral without commitment becomes permanent debt and comes back as an incident.

Memorizing the catalog. What has to be recognized is the signal in the story; the list without signals produces arbitrary slices.

Further reading

- [Humanizing Work — Guide to Splitting User Stories](#)
- [Humanizing Work — Why most people split workflows wrong](#)
- [Wake — Twenty Ways to Split Stories](#)
- [Cohn — SPIDR](#)
- [Agile Alliance — Story splitting](#)

BLOCK D · SPLITTING WHILE KEEPING VALUE

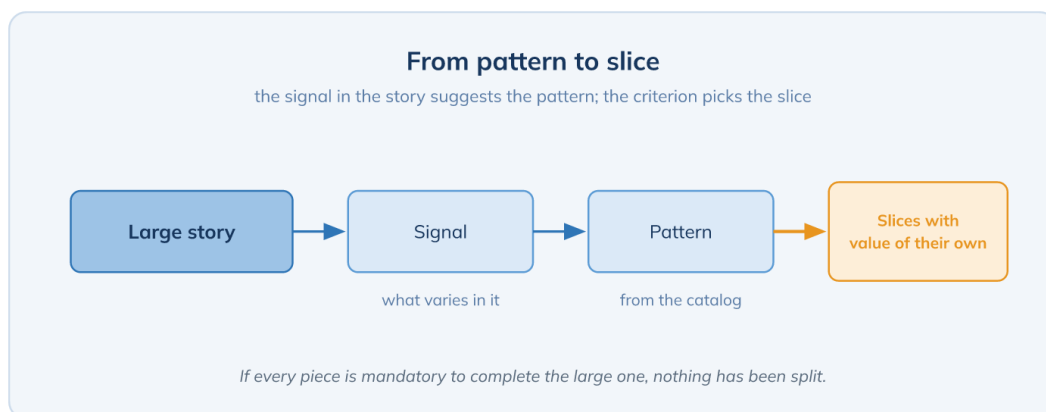
Chapter 17. From pattern to slice: procedure, choice and dependencies

CONSOLIDATING

The full procedure, the two rules that judge a slice, the meta-pattern for stories that resist and the dependencies a slice introduces.

Introduction

Knowing the patterns is not enough: you have to know which one to apply, how to judge the slices that come out and what to do when a story resists them all. In this chapter we close the block with the procedure, the selection criterion and the effect a slice has on dependencies.



17.1 Before splitting: preparing the story

The precondition of the procedure is counterintuitive and comes from Humanizing Work: before splitting, the story must meet INVEST **except for the S**, the one about size. That is, it must be independent, negotiable, valuable, estimable and testable, and fail only for being large.

When another property fails, splitting fixes nothing. And the one that almost always fails is the same: **value**. The story is large because it does not really express a slice of value but a bundle of technical work. In that case the right operation is the reverse, **recombining** until you find the observable change the bundle produces, and splitting that.

What is usually missing is value. Humanizing Work say it with twenty years of workshops behind them: when a story resists splitting, the property that is missing is the V, not the S.

17.2 Applying the patterns and generating candidates

The procedure has three moves. First, **identify the signal** the story offers (a flow, an «except», an enumeration, a catch-all verb, a percentage). Second, **apply two or three patterns** compatible with that signal, without stopping at the first: practice shows the second or third slice is usually better than the first. Third, **compare the candidates** against the criteria in 17.3.

It is group work at a whiteboard, not tool work. Five minutes with three people produce two or three candidate slices; half an hour with the tool produces one, and it is usually the horizontal one.

17.3 Judging a slice: the criteria and the false-slice test

A candidate slice is judged against six criteria, and it need not meet them all: it **keeps value** (each piece expresses an observable change), it is **verifiable** (an example can be written that confirms it), it **allows deprioritizing** (some piece could go undone), it **yields similarly sized pieces**, it **teaches something** (some piece reduces uncertainty) and it is **independent enough**.

The two rules that discriminate most are the third and the fourth, and they are worth applying every time. And there is a test that rules out false slices in a second: **if every piece is mandatory to complete the large story, nothing has been sliced**. The work has been distributed across more cards, which is the most frequent result of splits done under planning pressure.

17.4 The meta-pattern for stories that resist

When no pattern fits, Humanizing Work propose a three-step procedure that works surprisingly often: **find the core complexity** (which specific part makes the story hard), **identify the variations** around that complexity, and **reduce them to one** for the first slice. What was «manage the quarterly filing for every tax regime» becomes «file the quarter for a freelancer under the general regime», and the other variations become later slices.

There is also Gojko Adzic's **hamburger method**, particularly useful when the team can only think in layers: you list the technical layers (the bun and the fillings), list for each one the quality options from lowest to highest, and pick a minimum level of each layer to form the first complete slice. It is a way of using layered thinking to arrive at a vertical slice.

The Scrum Manager BoK documents this method; it is worth knowing that its page **inverts the terms vertical and horizontal** relative to the craft convention, and that the correction is pending.

17.5 Go big before you go small

There is a frequent case in which the «unsplittable» story turns out to be a task: if, when trying to split it, every piece is technical, the item was probably never a story. The way out is to go up a level, find the feature it belongs to, and split that. It sounds costly and usually saves half the work.

17.6 The dependencies a slice introduces

Every slice creates dependencies, and the workflow-steps one creates them especially: if slice two needs slice one, they can no longer be reordered. That does not invalidate the slice, but it does mean handling them with the hierarchy of chapter 7: **make them visible** (note them on the item, not in somebody's head), **remove the avoidable ones** (sometimes a provisional data set or a toggle is enough) and **manage the unavoidable ones** (order accordingly and tell whoever needs to know).

The reason to take them seriously is about flow: every unmanaged dependency lowers the probability of finishing on time, because it introduces waits that do not depend on the team. The flow management community insists on reducing them before managing them, and that is the right order.

17.7 Practice: Elephant Carpaccio

Splitting is learned by splitting. The canonical exercise is **Elephant Carpaccio**, from Alistair Cockburn, with Henrik Kniberg's facilitation guide: split a small feature into twenty vertical slices of a few

minutes each. Humanizing Work hold that fluency is reached with about three hours of deliberate practice, which makes this exercise one of the most profitable investments in a team's training.

At Vantia the exercise was done with the tax-deadline warning, the fifth item in the backlog. The first slice was emailing freelancers under a single regime about a single deadline, with no dashboard and no preferences, and that was enough to discover that almost half the warnings went to people who had already filed, which reordered the remaining slices.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Check the INVEST-except-size precondition before splitting, and recombine when what is missing is value.
- Generate two or three candidate slices from the signal in the story.
- Judge the candidates against the six criteria and rule out false slices with the mandatory-piece test.
- Apply the core-complexity-and-variations meta-pattern to a story that resists.
- Use the hamburger method to bring a team that thinks in layers to a vertical slice.
- Identify the dependencies a slice introduces and handle them in the right order.

Common mistakes and antipatterns

Splitting a story with no value. It produces pieces that have none either; you have to recombine first until you find the observable change.

Stopping at the first slice. The second or third is usually better, and comparing them takes five minutes.

Accepting a slice where every piece is mandatory. It is not a split but a distribution; nothing can be deprioritized and the large-batch risk remains intact.

Leaving dependencies in somebody's head. They reappear as waits mid-sprint and nobody had planned for them.

Further reading

- [Humanizing Work — Unsplittable stories: myth or reality?](#)
- [Humanizing Work — Story splitting Q&A](#)
- [Adzic — Splitting user stories: the hamburger method](#)
- [Cockburn — Elephant Carpaccio](#)
- [Kniberg — Elephant Carpaccio facilitation guide](#)

BLOCK E · STORY MAPPING AND BACKLOG CONTEXT

Chapter 18. What gets lost in a flat backlog · CONSOLIDATING

What information does not fit in an ordered list, which structures have been proposed and why the debate is still open.

Introduction

A well-split backlog has an uncomfortable side effect: the better it is split, the more pieces there are, and the harder it becomes to see which product they belong to. The ordered list is an excellent instrument for deciding what comes first and a terrible one for understanding the whole. In this chapter we look at exactly what gets lost, before recovering it in chapter 19.

18.1 Four things the list does not hold

The narrative. A list does not say what a person does with the product from start to finish. With two hundred items ordered by priority, nobody can reconstruct the journey of a Vantia freelancer from issuing their first invoice to filing the quarter.

The context. Each item appears alone, without the activity it belongs to, so its importance has to be remembered or argued again.

The relationships. That two stories serve the same activity, or that one is a variant of another, cannot be seen in a list; it is seen when somebody remembers it.

The composition of a release. The most useful question in product planning (which set of stories, shipped together, produces a meaningful change) cannot be answered by looking at an ordered column.

The diagnosis is old and still current. Jeff Patton stated it in 2008 by saying that the new user story backlog is a map. Almost twenty years later, most teams still work with the column.

18.2 The alternative structures proposed

Tool hierarchies. Epic, feature, story, subtask. It is the most widespread solution and its problem is that it organizes folders: it says what type an item is and not what a person does with the product.

The backlog board (Roman Pichler): a grid with themes on one axis and horizon on the other, which gives some context without claiming to tell the narrative.

The opportunity solution tree (Teresa Torres): goal, opportunities, solutions and experiments. It is not a backlog structure but a discovery one, and it is the best way of justifying why an item exists.

Impact mapping (Gojko Adzic): goal, actors, impacts and deliverables, connecting the work with the outcome sought.

The opportunity backlog (Marty Cagan): separating opportunities still to be explored from committed work, so they do not compete in the same list.

18.3 The case for the flat list

The opposite argument deserves to be taken seriously, because it is solid. Those who defend a flat list with labels hold that hierarchy introduces maintenance work, invites top-down planning and makes items inherit importance from their container instead of earning it through their value. They add that ordering a list forces decisions, while a tree allows deferring them indefinitely.

This guide's position is that the two are compatible and serve different purposes: **the order is decided in a list** (one dimension, no ties) and **the context is seen in a map** (two dimensions, narrative and detail). What does not work is asking one structure to do both.

18.4 What to do when the tool imposes a hierarchy

In most organizations the tool already imposes a hierarchy and it cannot be changed. Three practical measures avoid the worst. First, **do not confuse the container with the value**: an epic is not a priority because it contains many stories. Second, **keep the map outside the tool** if it does not fit inside, on condition of updating it (chapter 20). Third, **use labels for the narrative** (the activity each item belongs to) instead of creating new levels.

What you should be able to do

Having mastered this chapter, a professional is able to:

- State what information gets lost in a flat backlog and show it in a real example.
- Place the alternative structures and say which question each one answers.
- Sustain the argument for the flat list and explain the split between list and map.
- Live with a tool-imposed hierarchy without letting the container decide the value.

Common mistakes and antipatterns

Prioritizing by container. A large epic drags along items with no value of their own and displaces others that do have it.

Replacing the list with the tree. The tree allows you not to decide; the single-dimension order forces you to, and that obligation is its usefulness.

Creating new levels in the tool. Every level adds maintenance and gives the illusion of structure without contributing narrative.

Further reading

- [Patton — The new user story backlog is a map](#)
- [Bastow — Why your backlog hierarchy is broken](#)
- [Hinshelwood — Should your backlog be flat?](#)
- [Pichler — The Product Backlog Board](#)
- [Cagan — The opportunity backlog](#)

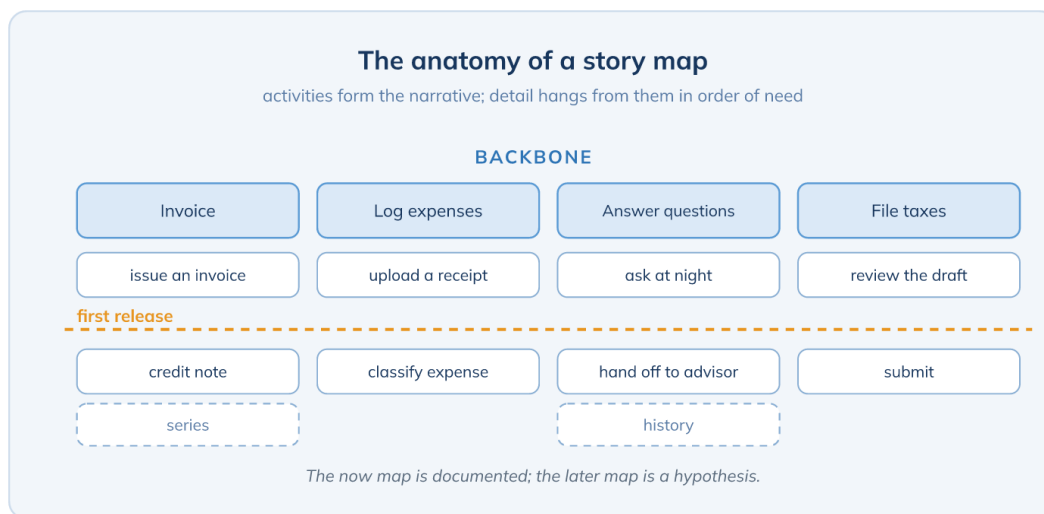
BLOCK E · STORY MAPPING AND BACKLOG CONTEXT

Chapter 19. The story map: anatomy and construction · ESTABLISHED

The pieces of the map in Patton's vocabulary, the five steps to build it and the mistakes its author documents.

Introduction

The story map recovers in two dimensions what the list loses: on the horizontal axis, the narrative of what a person does; on the vertical, the detail in order of need. It is this block's technique and one of the most profitable in the craft, because it is learned in one session and used for years.



19.1 The pieces and their vocabulary

Jeff Patton fixed the vocabulary and it is worth using precisely. **Users** are those who do things with the product. **Activities** are the large blocks of what they do (invoicing, logging expenses, filing taxes). **User tasks** are concrete steps, and Patton insists they be written as **short verb phrases** («issue an invoice», «upload a receipt»), not as nouns and not as complete stories.

Activities and main tasks, placed in their narrative order, form the map's **backbone**: the top line that reads left to right and tells the complete story. Below each task hang the **ribs**, the detail and the variants, ordered top to bottom by need.

There is one more distinction, **goal levels**: not every task sits at the same granularity, and the map allows moving up or down depending on what you want to see. Mixing levels is the most common cause of unreadable maps.

The map is not a flowchart. The horizontal axis is narrative rather than strictly temporal: it tells the story of use in the order in which it is best understood, not the exact sequence of clicks.

19.2 The now map and the later map

Patton distinguishes two uses with very different consequences. The **now map** documents how the work is done today, with or without a product, and it is built with data: interviews, analytics, observation. It serves to find where it hurts.

The **later map** describes how it will be, and it is a **hypothesis** until it is released and measured. Treating it as a plan is the mistake that turns the map into a project document: it is drawn in full, approved and executed, and by the time the first release arrives nobody remembers it was a conjecture.

At Vantia, the now map of night-time work showed something the backlog was hiding: the freelancer who gets stuck at eleven at night does not abandon the platform at that moment but two or three attempts later, and in half the cases the sticking point is classifying an expense rather than issuing an invoice. That fact reordered the later map.

19.3 The five steps to build it

Frame. Agree which product, which users and which problem are being discussed, and what is out of scope. Without this step the map grows without limit.

Map the big picture. Walk the whole activity «a mile wide and an inch deep»: every main task, none in detail. This is where what was missing gets discovered.

Explore. Go down into detail only where there is uncertainty or value: variants, edge cases, different users, what can go wrong.

Slice out viable releases. Draw horizontal lines separating shippable sets, each with its expected outcome (chapter 20).

Slice out a development strategy. Within the first release, decide where to start, usually with a skeleton that cuts through the narrative (chapter 15).

Facilitation follows the rules of chapter 9: a small group with the necessary perspectives, individual work before talking, and a time limit per phase. Two hours are enough for a first useful map.

19.4 Progressive decomposition and existing products

The map is refined where it matters, just like the backlog: plenty of detail in the first release, almost none in the following ones. A map with the same level of detail from end to end is a map that has spent effort on what may never be done.

In an existing product there is a shortcut that saves weeks: **map only what is new** and the stretch of the narrative it affects, instead of reconstructing the product's whole map. An exhaustive map of a mature product is an expensive exercise whose result ages before it is finished.

19.5 The five mistakes Patton documents

Losing the narrative. The map becomes a list of features in groups, with no journey that can be read.

Drowning in detail. The group goes deep on the first activity and never reaches the end of the big picture.

Obsessing over variants. Every possible alternative is collected before anyone knows which ones matter.

Mapping the whole product. A map of everything, permanently out of date, instead of a map of the work at hand.

Slicing releases with no outcome. Horizontal slices that group whatever fits the team, without saying what changes for anyone. That is the subject of the next chapter.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Build a story map with the right vocabulary, with its backbone and its ribs.
- Tell the now map from the later map, and treat the second as a hypothesis.
- Facilitate the five steps, with the big picture before the detail.
- Refine the map by proximity and map only what is new in an existing product.
- Recognize the five frequent mistakes in a map that has already been built.

Common mistakes and antipatterns

Writing complete stories on the backbone. The narrative line is read at a glance with short verb phrases; with whole stories it stops being readable.

Mapping the whole product «to have it». The cost is high, the result ages and nobody consults it; it pays to map the work at hand.

Mixing goal levels. Tasks of very different granularity on the same row make the map unreadable and distort the narrative.

Treating the later map as an approved plan. It is a hypothesis; turning it into a commitment reintroduces the project into product work.

Further reading

- [Patton — Story Map Concepts \(quick reference\)](#)
- [Patton — The new user story backlog is a map](#)
- [Patton — Five common story mapping mistakes](#)
- [Scrum Manager BoK — Story map \(in Spanish\)](#)

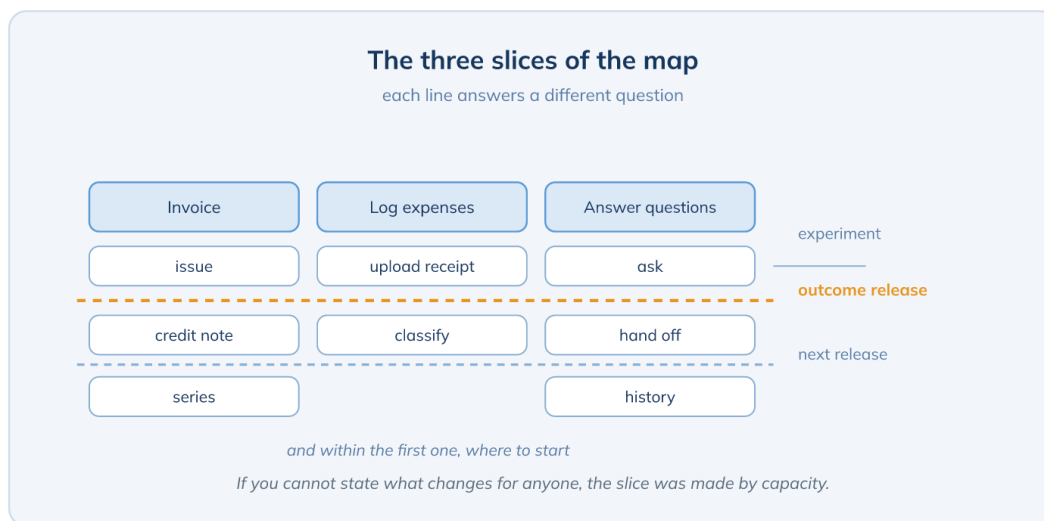
BLOCK E · STORY MAPPING AND BACKLOG CONTEXT

Chapter 20. Slicing by outcome and keeping the map alive · CONSOLIDATING

The three slices of the map, how a capacity-driven slice is recognized and what keeps the map useful after the workshop.

Introduction

A map is for deciding what ships together, and that is where most teams go back to their old habits: they group whatever fits the next sprint and call that a release. In this chapter we look at the three slices Patton distinguishes, with their criteria, and at what it takes for the map to still be true three months later.



20.1 The three slices

The outcome release. A set of stories that, shipped together, produce a change that can be stated and measured. The condition is twofold: an expected outcome and a metric that checks it. At Vantia, the assistant's first release was «freelancers who get stuck at night get a useful answer there and then», with night-time abandonment as the measure.

The experiment or minimum viable product. A smaller slice whose purpose is not to deliver sustained value but to check a hypothesis. Patton defines the minimum viable product in a way worth quoting because it shifts the criterion: the smallest product you could build that reaches your target outcomes. Not the smallest that can be built.

The development strategy. Within the first release, the build order, which normally starts with a skeleton cutting through the whole narrative (chapter 15) and then fattens up where the most is learned.

Three slices, three questions. The release answers «what changes for someone?»; the experiment, «what do we want to find out?»; the strategy, «where do we start?». Confusing them produces releases that change nothing and experiments that take a quarter.

20.2 How a capacity-driven slice is recognized

A slice made by team capacity has an unmistakable signal: **you cannot state what changes for anyone**. If, when asked «what will be different when this ships?», the answer talks about what was built («the expenses module will be there») rather than about what somebody will be able to do, the slice was made by capacity.

The second signal is that the slice contains pieces from unrelated activities, chosen because they fitted the slot. The third is that there is no associated metric, or the metric is about progress («percentage of stories completed») rather than about outcome.

Fixing it does not require redrawing the map: it is enough to ask, for each slice, which user will notice what, and move the pieces until the answer exists.

20.3 From the map to the conversation and the criteria

The map does not replace anything from block C: it feeds it. Each task in the first release is a candidate story, and each story goes into the workshop of chapter 9 or the Example Mapping of chapter 11, where it acquires its rules, its examples and its criteria.

The complete route, then, is this: discovery produces opportunities; the map arranges them into a narrative and slices out outcome releases; the conversation turns each task into a story with its examples; and splitting adjusts the size where needed. Every piece of this guide has a place in that chain.

20.4 Keeping the map alive

A map that is not updated stops being an instrument and becomes decoration. Three practices keep it useful. First, **update it with what has been learned** after each release: what was discovered, what was dropped, what appeared. Second, **use it in the conversations where decisions are made** (the stakeholder review, the refinement of the next stretch); a map that is never opened is never maintained. Third, **keep its date and status visible**, telling documented apart from hypothetical.

When the map is abandoned after the initial workshop, what happens is that the team goes back to the flat list with the cost of the workshop already paid. It is the most frequent failure mode and the subject of chapter 21.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Slice a map into releases with their expected outcome and their metric.
- Tell apart the outcome release, the experiment and the development strategy.
- Apply the outcome-based definition of the minimum viable product rather than a size-based one.
- Recognize a capacity-driven slice by its three signals and correct it.
- Keep the map alive with the three practices, and tell documented from hypothetical within it.

Common mistakes and antipatterns

Calling whatever fits the sprint a release. With no statable outcome, the slice makes it impossible to judge whether it was worth it, and the review turns into a work tally.

Defining the minimum viable product by its size. The criterion is the outcome sought; «the smallest that can be built» usually reaches none.

Measuring the release with progress metrics. Percentage complete says nothing about the change produced for whoever uses the product.

Abandoning the map after the workshop. The cost of the workshop is already paid; not updating it wastes the investment and sends the team back to the flat list.

Further reading

- [Patton — It's All in How You Slice It](#)
- [Kniberg — Making sense of MVP](#)
- [Patton — Dual Track Development is not Dual Track](#)
- [Torres — Outcomes versus outputs](#)

BLOCK E · STORY MAPPING AND BACKLOG CONTEXT

Chapter 21. Neighboring maps and the map inside the tool · CONSOLIDATING

Four techniques that get confused, how they chain together, and the three failure modes of taking the map into a tool.

Introduction

Around the story map there are three techniques with similar names and different purposes, and choosing badly costs a whole session. In this chapter we tell them apart, show how they chain together and close the block with what happens when the map enters a management tool.

21.1 Four techniques, four questions

Story map. Question: what does a person do with the product, from start to finish, and which part of that do we ship first? Its material is user tasks.

Customer journey map. Question: how does a person live that journey, with their emotions, pain points and channels? Its material is stages and experiences, and its natural use is discovery.

Impact map. Question: which behaviors of which actors bring us closer to the goal, and what could we build to prompt them? Its material is goal, actors, impacts and deliverables.

Opportunity solution tree. Question: what opportunities lie under this goal and what solutions might address them? Its material is opportunities found in research.

The typical confusion. Using the story map to do discovery (and ending up without data) or the journey map to plan releases (and ending up without tasks). Each answers its own question well.

21.2 How they chain together

The natural chain runs from discovery to delivery. Customer research produces opportunities, which the **tree** arranges under the goal. The **impact map** connects the goal with the behaviors to be prompted. The **journey map** shows how the activity is lived today and where it hurts. And with that, the **story map** organizes the narrative and slices out the first release.

The first two techniques belong to discovery and have their own skill in Scrum Manager, as does customer research; they are cited here so you know when to change tools.

There is one more neighbor worth knowing: Paulo Caroli's **Product Backlog Building**, which produces an initial backlog out of an inception. It is the contact point with the Agile Inception skill, and its output is exactly what this block teaches you to structure.

21.3 The map inside the tool: three failure modes

The hierarchy replaces the two axes. The tool offers epic, feature and story, and the team believes that is a map. It is not: a hierarchy has one axis (containment) and the map has two (narrative and detail). Precisely what you wanted to recover is lost.

The giant synchronized map. With an extension that syncs the map to the backlog comes the temptation to map the whole product. The result is a wall of hundreds of cards nobody reads and whose upkeep eats hours.

The abandoned physical map. The workshop is held with sticky notes on the wall, photos are taken and it is never touched again. Within weeks the map contradicts the backlog and nobody knows which one rules.

All three are avoided with the same decision: choose **one single place** where the map is true, keep it the size of the work at hand and update it in the conversations where decisions are made.

21.4 With artificial intelligence

Mapping tools now offer to generate the map from a product description, and they do it fluently: they produce plausible activities and tasks in seconds. The problem is block G's in its purest form: a **now** map generated without data documents nothing, because it comes from no observation; and a **later** map generated without evidence is the machine's hypothesis, not the team's.

The use that does pay off is the mirror image of chapter 25's: asking it to **critique** a map the team has made. It finds gaps in the narrative, tasks missing between two steps, variants not considered and slices with no storable outcome, and all of that is excellent material for the next session.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Choose between story map, journey map, impact map and opportunity solution tree according to the question to be answered.
- Chain the techniques from discovery to delivery and know when to switch from one to another.
- Recognize the three failure modes of the map in a tool and decide where the truth lives.
- Use assistance to critique your own map rather than to generate one without data.

Common mistakes and antipatterns

Confusing the hierarchy with the map. The narrative is lost, which is exactly what the map adds over the list.

Mapping the whole product in the tool. The resulting wall is not read and its upkeep competes with the real work.

Two sources of truth. A map on the wall and a backlog in the tool diverge within weeks and force a decision about which rules at the worst possible moment.

Generating the now map with a model. The now map is documented with data; generated, it only reflects what the model has seen in other products.

Further reading

- [Adzic — Impact Mapping](#)
- [Nielsen Norman Group — Journey Mapping 101](#)
- [Torres — Opportunity Solution Trees](#)
- [Caroli — Product Backlog Building](#)
- [Skill Arena — Agile Inception](#)

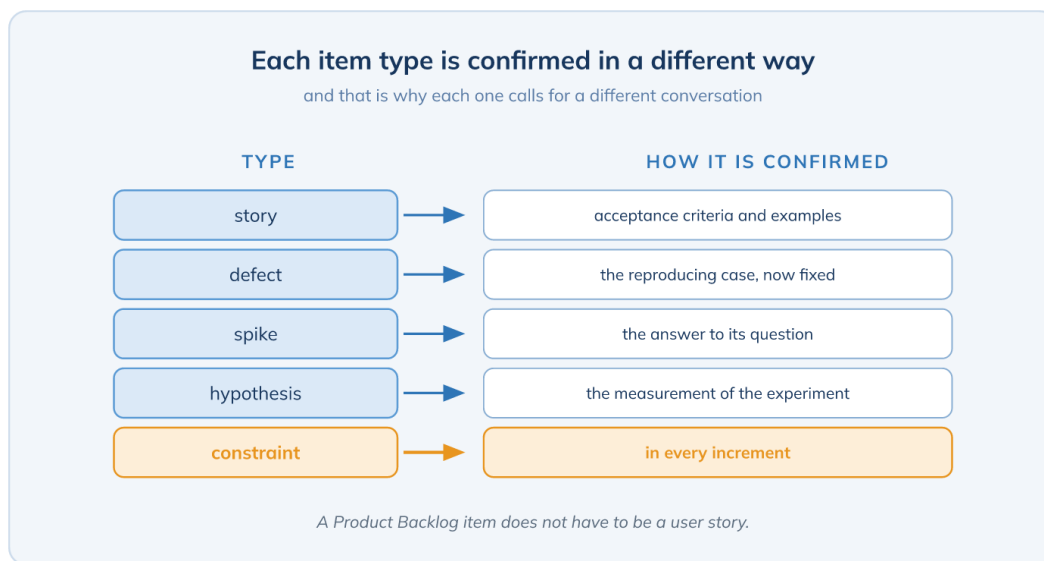
BLOCK F · NOT EVERYTHING IS A STORY

Chapter 22. Product Backlog item versus story · ESTABLISHED

What kinds of item coexist in a real backlog and how each one is confirmed.

Introduction

A real backlog does not contain only stories, and it never did. It contains defects, investigations, technical work, constraints, regulatory obligations and hypotheses to validate. Forcing all of that into the story mold produces the fictions of chapter 6. In this chapter we sort out the catalog of types and, above all, how each one is confirmed.



22.1 The only thing Scrum asks for

It is worth repeating here because it is the basis of the block: a Product Backlog item needs a **description, an order and a size**. The 2017 Scrum Guide listed types (features, functions, requirements, enhancements and fixes) and the 2020 version removed that list, leaving the form in the team's hands.

Scrum Manager's Product Owner 2.0 core guide states it in the same direction: a backlog item does not have to be a user story; the story is a tool, not an obligation. That sentence sums up the whole block.

22.2 The catalog of types and their confirmation

User story. Confirmed with acceptance criteria and their examples (chapters 10 and 11).

Defect. Confirmed by reproducing the failure and checking that it stops happening. It needs the case that reproduces it, not a story template.

Spike. Confirmed with the answer to its question and with the decision it enables. It needs an explicit question and a time box, and it produces no functionality.

Technical or enabler work. Confirmed with the agreed technical criterion and with the confirmation that it enables what it claimed to enable (chapter 23).

Constraint or non-functional requirement. Confirmed **in every increment**, not once: it belongs to the Definition of Done or to a cross-cutting constraint (chapter 10).

Regulatory compliance. Confirmed with the evidence the regulation requires, which usually includes traceability and approval by an identifiable responsible party.

Hypothesis with its outcome. Confirmed by measuring the experiment, and the result may be discarding it. The full toolkit belongs to Product Owner core training.

Why the mode of confirmation matters. It decides which conversation is needed. A spike calls for agreeing the question; a constraint, for agreeing the threshold; a hypothesis, for agreeing what measure would refute it. Treating them all as stories means holding the wrong conversation with all of them.

22.3 All in the same list, ordered by value

That the types are confirmed differently does not mean they live in separate lists. They compete for the same team time, and therefore they are ordered together. A defect that makes users abandon comes before a new feature, and an investigation that unblocks three stories may come before all three.

Vantia's backlog excerpt illustrates it: alongside the assistant's stories sit the handoff to the advisor (which came out of a refinement conversation), the reinforcement of two weak subject areas (which came from analyzing pilot errors) and a dashboard for advisors (which a stakeholder asked for). Each one is confirmed in its own way and all of them sit in the same column, with a note about their provenance.

22.4 What the BoK vocabulary requires

The Scrum Manager BoK keeps pages for the pieces of this catalog (Product Backlog, agile requirements, unit of work, task, epic, theme) and it is worth using its Spanish lexicon when writing for the organization. Two helpful clarifications: **unit of work** is the generic term when you do not want to prejudge the type, and **task** refers to the internal technical work of a story, not to a backlog item.

What you should be able to do

Having mastered this chapter, a professional is able to:

- List the item types that coexist in a backlog and say how each one is confirmed.
- Decide the type of an item before deciding its format.
- Order different types in a single list by their value, without creating parallel lists.
- Use the Scrum Manager BoK lexicon when writing for the organization.

Common mistakes and antipatterns

Turning every defect into a story. A defect is confirmed by reproducing the failure; wrapping it in «as a user, I want it not to fail» adds noise and hides the reproducing case.

Spikes without a question. A spike with no explicit question becomes open-ended investigation time with no decision attached.

Parallel lists by type. If the types do not compete in the same list, the order stops reflecting value and decisions get made separately.

Confirming a constraint only once. Performance or accessibility degrade with every change; their home is the Definition of Done, verified in every increment.

Further reading

- [The Scrum Guide 2020](#)
- [Scrum.org — Myth 4: the Product Backlog need not consist of user stories](#)
- [Scrum Manager BoK — Product Backlog \(in Spanish\)](#)
- [Scrum Manager BoK — Agile requirements \(in Spanish\)](#)
- [Skill Arena — Product Owner](#)

BLOCK F · NOT EVERYTHING IS A STORY

Chapter 23. Technical work without fake users · ESTABLISHED

The three honest ways out for work with no visible user, and how an honest technical story is told from a disguised task.

Introduction

«As a server, I want an index on the invoices table, so that I can respond faster.» That sentence, or one like it, exists in almost every backlog in the world, and it communicates nothing. In this chapter we look at the three honest ways of handling work that has no visible user, and how to choose among them.

23.1 Why the fake user appears

The fake user is not a drafting slip: it is the consequence of a misunderstood rule. When the organization requires every backlog item to have the shape of a story, the team complies with the shape and gives up on the content. The responsibility, then, lies with the rule.

The damage is twofold. On one hand, the story does not allow ordering, because it does not say who it serves or how much it is worth. On the other, it hides the conversation that was actually needed, which was technical and with technical criteria.

23.2 The three honest ways out

Find the real beneficiary and the value. Often it exists and nobody looked for it. The index on the invoices table does not serve the server: it serves the freelancer who waits four seconds every time they open their list, and the value can be stated in abandonment or in support queries. When the beneficiary appears, the item becomes a legitimate story again.

State it plainly as technical work. If there is no distinguishable beneficiary, write what it is, with its economic justification: «reindex the invoice store to sustain next year's expected growth». It can be called a technical item, an FDD-style feature, a spike or an enabler, as the case may be, and none of those forms needs a template.

Turn it into a constraint or a Definition of Done criterion. When the work expresses a quality that must always hold (response time, accessibility, encryption), its home is not a backlog item done once but a criterion verified in every increment.

The dishonest way out. Inventing a user («as a system...», «as a database...») ticks the form's boxes and switches off the three decisions that had to be made: who benefits, how much it is worth and how it will be checked.

23.3 Honest technical story versus disguised task

This is the distinction most used in refinement, and it has a simple test. An **honest technical story** produces a change in the system that somebody identifiable needs and that can be prioritized against other things: it can be postponed, it can be discarded and it can be explained to a non-

technical person. A **disguised task** is a necessary step of something else: it cannot be postponed without blocking the story it serves, and its home is inside that story.

Humanizing Work publish the criterion with examples and add a useful guide: if the item makes no sense outside the context of another item, it is a task. The practical consequence is that the backlog gets much cleaner when this test is applied: a good share of the technical items cluttering it are tasks of stories that are already on the list.

23.4 Technical debt and its prioritization

Technical debt deserves a mention of its own because handling it as a backlog item is a known source of arguments. Stated as technical work with no beneficiary and no consequence, it always loses against a new feature. Stated with its observable effect (how much it slows every change, how many incidents it produces, what risk it introduces), it competes on equal terms.

The formulation that works is the one about **interest**: instead of «we need to refactor the invoicing module», «every change in invoicing costs twice as much and produces an incident one time in three, and that will stay the case until it is reorganized». With that formulation, the decision belongs to product and is not a concession to the technical team.

23.5 The nuance of platform teams

There is one case where the «technical user» is not a fiction: teams whose product is an internal platform. There, whoever develops is a real customer, with real needs and alternatives (they can build their own solution or use another), and stories with developers as users are legitimate and necessary.

The team topologies literature has sorted that case out, and its criterion for the platform is demanding: it must be the **thinnest viable platform** that solves the problem of its user teams, with an explicit service relationship. In that context, everything this guide teaches applies just the same, with the developer as user and their experience as value.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Choose among the three honest ways out for an item with no visible user.
- Find the real beneficiary of technical work and state its value in observable terms.
- Tell an honest technical story from a disguised task with the context test.
- Formulate technical debt with its interest, so it competes on equal terms in the backlog order.
- Recognize the case of platform teams, where the developer is a real user.

Common mistakes and antipatterns

Inventing a technical user. It ticks the form's boxes and switches off prioritization; it also prevents the technical conversation that was actually needed.

Putting tasks in the backlog. It inflates the list with items that cannot be postponed and makes the order stop meaning anything.

Asking for a fixed percentage for technical debt. It turns a product decision into a negotiated quota; with the interest well stated, debt earns its place on value.

Treating the internal platform as work with no customer. Its users are identifiable and demanding, and their experience can be measured like that of any end user.

Further reading

- [Humanizing Work — Dealing with technical stories](#)
- [SAFe — Enablers](#)
- [Cohn — Not everything needs to be a user story: using FDD features](#)
- [Team Topologies — key concepts](#)
- [Scrum Manager BoK — Unit of work \(in Spanish\)](#)

BLOCK F · NOT EVERYTHING IS A STORY

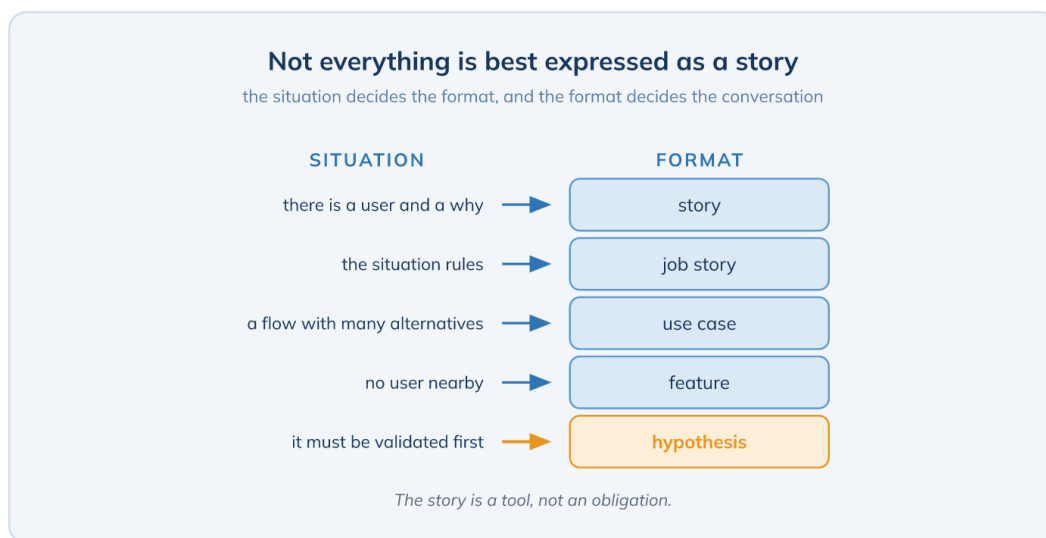
Chapter 24. Alternative representations and how to choose the format

CONSOLIDATING

The available formats with their author and their use, and the five questions that decide which one fits.

Introduction

This chapter closes the block and, in a way, the classic part of the guide: the catalog of ways to express work and the criterion for choosing among them. Whoever masters it stops arguing about templates and starts deciding with arguments.



24.1 The job story

Alan Klement proposed in 2013 replacing the classic template with one centered on the situation: «**When [situation], I want to [motivation], so I can [expected outcome]**». His argument is that the user's persona tells you less than the context in which the need appears, and that the classic template invites inventing personas.

The experience of Intercom, which popularized it, and its authors' later review agree on something important: the job story **complements** rather than replaces. It works better when the situation rules (the freelancer at eleven at night, with tomorrow's deadline) and worse when what matters is telling apart two kinds of user with opposing needs.

The Vantia backlog item we have been quoting is written that way, and that is why it works: «when I get stuck at night with a filing and have nobody to ask, I want a clear explanation there and then, so I can finish that same night without losing heart».

24.2 The use case and its slice

The **use case** describes an actor's goal with its main flow and its alternatives, and it is still the best tool when the flow is the complicated part: many branches, many exceptions, or expensive consequences if one is forgotten.

Its agile version, **Use-Case 2.0** (Jacobson, Spence and Kerr), answers the monolithic-document objection with **use-case slices**: you identify the complete use case to have the map, and you deliver it slice by slice, each with its story and its tests. It is the point where the use case and the story stop being alternatives.

Situations where it pays off: systems under regulations that require traceability, distributed teams that cannot talk daily, domains where the cost of error is high. The Scrum Manager basic manual this guide supersedes already offered that table of contexts, and it still holds.

24.3 Feature, hypothesis, scenario and structured notation

The FDD-style feature. Palmer and Felsing proposed a syntax for work with no nearby user: action, result and object («calculate the total withholding of an invoice»). Cohn expressly recommends it for APIs and internal components.

The hypothesis with its outcome. For what has to be validated before building: «we believe that [this change] will produce [this outcome]; we will know when we see [this measure]». Its full toolkit belongs to Product Owner training and to the discovery skill.

The Gherkin scenario. It is a **confirmation**, not a card. Writing the backlog item directly as a scenario confuses the what with how it will be checked, and produces unreadable backlogs.

EARS notation. Alistair Mavin proposed five templates for conditional requirements («when..., the system shall...», «while...», «if..., then...»). It is useful when a machine will read the requirement or when all ambiguity must be avoided, which is why it reappears in block H, in specifications for agents.

The pitch and the design document. Ryan Singer (*Shape Up*) proposes the pitch: problem, appetite, sketched solution, risks and what is out of scope. Google's design docs serve a similar purpose for technical decisions. Both work when the decision needs an argument and does not fit on a card.

24.4 The five questions that decide

Faced with an item, five questions are enough to choose the format:

Who benefits? If there is an identifiable user with a need, a story. If the beneficiary is the team itself or the system, technical work stated plainly.

What is unknown? If uncertainty is high, a hypothesis or a spike before any solution format.

What conversation is needed? If rules and examples have to be agreed, a story with Example Mapping. If a threshold has to be agreed, a constraint. If a flow has to be agreed, a use case.

Who will read it? If it is read by the team that talks daily, little text is enough. If it is read by an auditor, another team or an agent, precision and notation are needed.

What does being wrong cost? The more expensive the error, the more explicit the format.

The rule that sums up the block. The format serves the conversation that needs to happen. Chosen the other way round, the conversation adapts to the format, and then you only talk about what the format allows you to say.

24.5 Knowing when not to write a story

The exercise to close the block with is one of reclassification: take a batch of ten or fifteen real items from your own backlog and decide, with the five questions, what each one is. The typical result is revealing: between a third and a half are not stories, and a good part should not be in the backlog at all.

And there is a final case that deserves its own name: when the story **adds noise**. A change to a legal text, an update to a tax rate or the correction of a typo need no user, no motivation and no benefit. «Update the reduced VAT rate to 5 % from 1 January» is a perfect item exactly as written.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Choose the format of an item with the five questions and justify the choice.
- Write a job story and say when it fits better than the classic template.
- Use use-case slices when the flow has many alternatives or the error is expensive.
- Express work with no nearby user with the feature syntax, and what is uncertain as a hypothesis.
- Recognize items that need no format at all and leave them as they are.
- Reclassify a batch of badly formulated items and decide which ones should not be in the backlog.

Common mistakes and antipatterns

Replacing every story with a job story. Swapping one universal template for another solves nothing: the problem was the universality, not the template.

Writing the item as a Gherkin scenario. The scenario confirms, it does not describe; used as a card it produces backlogs nobody can order.

Dismissing use cases out of prejudice. In domains with many branches or required traceability they are the right tool, and their sliced version is fully compatible with incremental delivery.

Wrapping in a template what does not need one. A regulatory change written as a story adds three lines of fiction to something that was already clear.

Further reading

- [Klement — Replacing the user story with the job story](#)
- [Adams — How we accidentally invented job stories \(Intercom\)](#)
- [Jacobson, Spence and Kerr — Use-Case 2.0](#)
- [Mavin — EARS notation](#)
- [Singer — Shape Up: write the pitch](#)
- [Ubl — Design docs at Google](#)

BLOCK G · ARTIFICIAL INTELLIGENCE IN STORY WORK

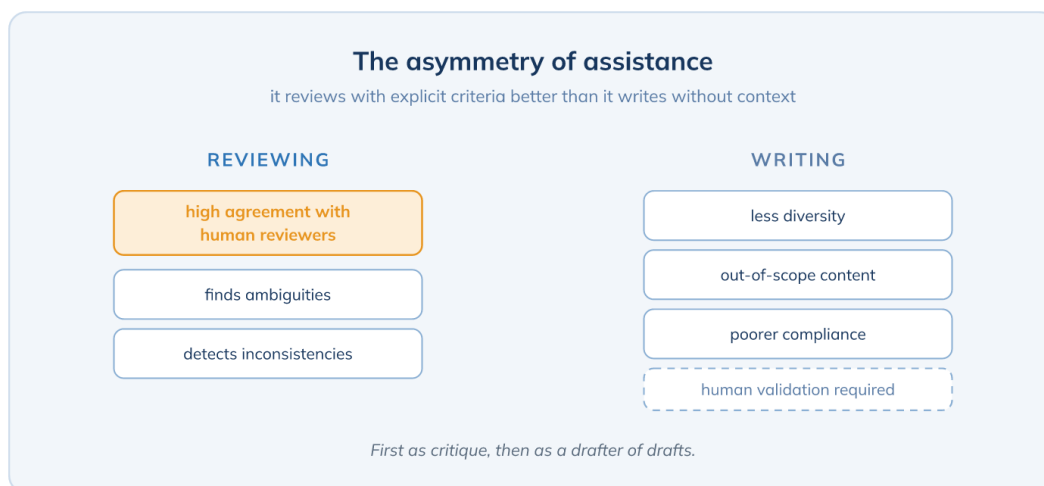
Chapter 25. The asymmetry: artificial intelligence reviews better than it writes · CONSOLIDATING

What the evidence says about evaluating versus generating stories, and the order of use that follows from it.

Introduction

Almost everyone's first impulse, with a language model at hand, is to ask it for stories. It is the lowest-yield and highest-risk use, and the evidence says so fairly clearly. In this chapter we look at what has been measured, what result it gives and what working order follows.

It is worth starting with a warning about the nature of what follows. Much of the available research is recent, done in the lab and with small samples; there is little field evidence. When something is a Scrum Manager proposal rather than a confirmed finding, we will say so.



25.1 As an evaluator, with explicit criteria

The work of Ronanki, Cabrero-Daniel and Berger (2023) compared story quality evaluation by a model with that of human reviewers, and found high agreement, around 70-75 %, rising above 85 % when taking the best of three answers. The research of Quattrocchi et al. (2025) confirms the evaluation side: with clear criteria, the model evaluates well.

In detecting ambiguities and inconsistencies, comparative studies against rule-based tools favor the model, which finds problems the rules do not cover, including some that manual analysis misses. That is consistent with the split in chapter 5: what can be decided by reading can be detected mechanically, and a language model is today the best available detector.

The condition: explicit criteria. The model evaluates well when it is given the criteria (INVEST, the thirteen quality criteria, the list of risky words, the team's own rules). Without criteria, it produces textbook opinions.

25.2 As an author, without context

The same work by Quattrocchi found the opposite when generating: the stories produced have comparable surface quality but are **less diverse**, comply **worse** with quality criteria and contain **more out-of-scope content**. The ALAS project, with agents specialized in improving stories, raised team satisfaction, and its conclusion includes an explicit condition: it requires manual validation by the Product Owner.

The most informative field study is Steghöfer's (2026), with Product Owners already working this way. His two findings are common sense and that is why they hold: the output is **too generic because the context is missing**, and it **does not reference the underlying source material**, so verifying it costs almost as much as having written it.

The conclusion is not that it should not be used for drafting. It is that drafting is the step where it contributes least and the one that demands the most verification.

25.3 The order of use that follows

With that asymmetry, the sensible order is the reverse of the one almost everybody applies.

First as critique. Give it what the team has already written, with the criteria stated explicitly: INVEST, ambiguities, duplicates and overlaps across the whole, coherence between items, missing edge cases. The result is a list of candidates to decide in conversation.

Then as a drafter. With enough context (chapter 26), to produce a first text the team corrects, variants of the same story, proposals for criteria and examples, or candidate slices to be judged afterwards against the criteria of chapter 17.

Never as a source of knowledge about users. The Scrum Manager BoK puts it in a sentence worth remembering: artificial intelligence does not understand real users; it generates stories that sound right because it has seen thousands of similar examples, without direct experience of this product's users.

25.4 What Scrum Manager already holds

This reading is not new in the house. The Agile Inception skill already holds that assistance is a better reviewer than author of the backlog, and the BoK carries the same criterion on its user story and acceptance criteria pages, with its explicit limit. The Product Owner 2.0 core guide applies it to the role's work and adds the pertinent warning: an impeccably drafted story that nobody has talked through is a specification in disguise, with all its misunderstandings intact.

Tools that take on precisely that critique role have appeared on the market, such as Mountain Goat Software's story reviewer, which assesses outcome clarity, scope focus, testability and backlog fit. It is a sign that the asymmetry is starting to be recognized in products too.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain the asymmetry between evaluating and generating stories with the evidence behind it.
- Use assistance as critique with explicit criteria and obtain a list of candidates.
- Decide at which point in the work a draft is worth asking for, and with what verification.

- Reject the use of assistance as a source of knowledge about the product's users.
- Count the cost of verification as part of the gain calculation.

Common mistakes and antipatterns

Asking for «twenty stories» up front. It produces a generic, undiverse batch whose verification eats the gain; it also fixes a scope nobody has decided.

Critiquing without criteria. Without explicit criteria, the review returns textbook advice instead of concrete defects in these stories.

Accepting the critique unfiltered. The list is one of candidates, not decisions: discarding, merging or rewriting remains product work.

Confusing finish with quality. Well-written text does not indicate that the content is right, and that confusion has a name in chapter 27.

Further reading

- [Quattrocchi et al. — Can LLMs generate user stories and assess their quality?](#)
- [Ronanki et al. — ChatGPT for user story quality evaluation](#)
- [Zhang et al. — ALAS: agents for improving story quality](#)
- [Steghöfer — Faster than the Team, Faster than the Customer](#)
- [Scrum Manager BoK — User story \(in Spanish\)](#)

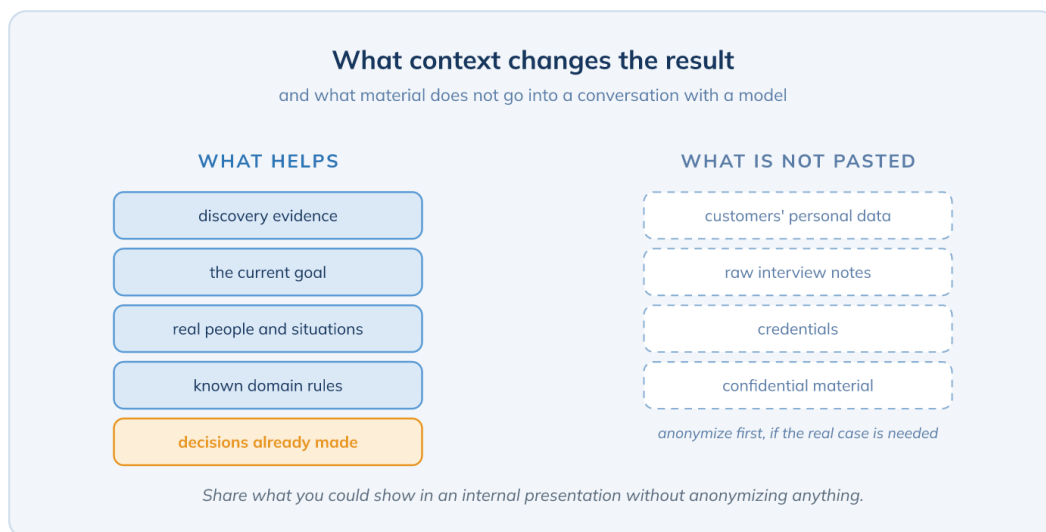
BLOCK G · ARTIFICIAL INTELLIGENCE IN STORY WORK

Chapter 26. Context, invention and provenance · EMERGING

What context changes the result, what a model invents, how to verify it cheaply and how provenance is preserved.

Introduction

If the previous chapter decides **when** to use assistance, this one decides **with what**. The difference between generic and useful output lies almost entirely in the context you give it, and the difference between useful output and a problem lies in knowing which part was invented. In this chapter we cover both, and close with provenance.



26.1 What context changes the result

The context that genuinely improves a generated story is concrete and limited: the **discovery evidence** (what has been observed, with what data), the **epic or goal** it serves, **real people** with their situations (not generic archetypes), the **known domain rules** and the **decisions already made** that will not be reopened.

At Vantia, that context takes little more than a page: the night-time abandonment problem with its figure, the current goal, a description of the freelancer who works at night and of the accounting firm's advisor, the assistant's three rules and the decision that a made-up answer is worse than admitting the limit. With that page in front of it, the output changes in kind.

Persistent context, not repeated. What is stable about the product (vision, goal, personas, rules, glossary) is best kept in a reusable document attached to every task, rather than rewritten in every prompt. That discipline reappears in chapter 28 as context layers.

26.2 What not to paste

Some material must not go into a conversation with a model, and it is worth deciding before you need to: **customers' personal data** (names, tax identifiers, addresses, the content of their

documents), **raw interview notes** with identifiable data, credentials, and any material under contractual confidentiality.

The operating rule is simple: share what you could show in an internal presentation without anonymizing anything. When the real case is needed, anonymize first. The privacy and responsible-use toolkit belongs to the AI applied to agile work skill, which treats the question with the detail it deserves.

26.3 The taxonomy of what gets invented

Synthetic users and needs. The model produces plausible profiles that correspond to nobody. Nielsen Norman Group have documented the synthetic users problem with a clear conclusion: they are useful for practice, not for deciding.

Plausible but false rules. The most dangerous kind, because they sound like domain knowledge: thresholds, deadlines, regulatory exceptions. In a tax product, this category can produce incorrect statements about the regulations themselves.

Criteria without evidence. Numbers appear (three seconds, 95 %) that nobody has decided and that are later defended as if they were agreements.

False consensus. The text presents as agreed what is only one possible interpretation, and it reaches the team in the form of a decision already made.

Systematic reviews of the field confirm that most research is at an early stage and document how often invented material appears, in a proportion no organization should assume without verification.

26.4 How to verify cheaply

Full verification is expensive and not always necessary; three low-cost techniques catch most of the problems.

Demand the source of every criterion. Ask, alongside each criterion or rule, where it comes from (an interview, an earlier decision, the regulations, or «proposed by me»). Whatever has no source is marked as an assumption.

Mark assumptions explicitly. Have the output distinguish what it knows from what it has inferred. The specification tools of block H use a literal marker for this, and the practice carries over to any story.

Cross-check with whoever knows the domain. Half an hour of the accounting firm's advisor is worth more than two hours of your own review. And it is the only filter that catches plausible but false rules.

The cost of these three techniques is real and has to be counted: part of the gain from generation goes into verifying. Acknowledging that is what tells professional adoption from enthusiasm.

26.5 Provenance and traceability

With assistance in the process, the story no longer has a single author, and it is worth making that visible. The minimum useful distinction has three categories: what was **contributed** (the team's

evidence and decisions), what was **inferred** (what the model deduced from the context) and what was **generated** (what it produced without support), plus the fact of **who validated it**.

No system is needed: a single line at the foot of the item is enough. At Vantia, the ordered backlog excerpt includes an origin column for precisely this reason, and in it sit «freelancer interviews», «refinement conversation», «team proposal» and «automated usage analysis». Having the origin visible lets you discuss the item with the confidence its provenance deserves.

There is also an external reason. Article 50 of the European AI Act, applicable since August 2026, sets transparency obligations for generated content, with an exemption where there is **human review or editorial control and an identifiable responsible party** for the content. For story work, that condition coincides with good practice: somebody answers for every backlog item. The regulatory detail belongs to other skills; here it is enough to know that traceability is no longer merely a professional courtesy.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Prepare the product context that makes assisted output useful, and keep it as a reusable asset.
- Decide what material does not go into a conversation with a model and anonymize when needed.
- Recognize the four classes of invention in generated material.
- Apply the three cheap verification techniques and count their cost in the gain calculation.
- Record the provenance of an item, telling apart what was contributed, inferred and generated, with its responsible party.

Common mistakes and antipatterns

Prompting without context and blaming the result. Without evidence or domain rules the output can only be generic; the defect is in the request.

Repeating the context in every conversation. Besides being costly, it produces diverging versions of the same context; better to keep it in one document.

Pasting raw interview notes. They contain identifiable data and are rarely needed: what helps is the distilled evidence, not the transcript.

Accepting numbers nobody decided. Invented thresholds become commitments the moment they enter an acceptance criterion.

Presenting what was inferred as agreed. The team inherits decisions it does not know exist, and finds out when one of them turns out to be wrong.

Further reading

- [Steghöfer — Faster than the Team, Faster than the Customer](#)
- [Cheng et al. — Generative AI for requirements engineering \(review\)](#)
- [Nielsen Norman Group — Synthetic users](#)
- [European AI Act, Article 50](#)
- [Skill Arena — AI applied to agile work](#)

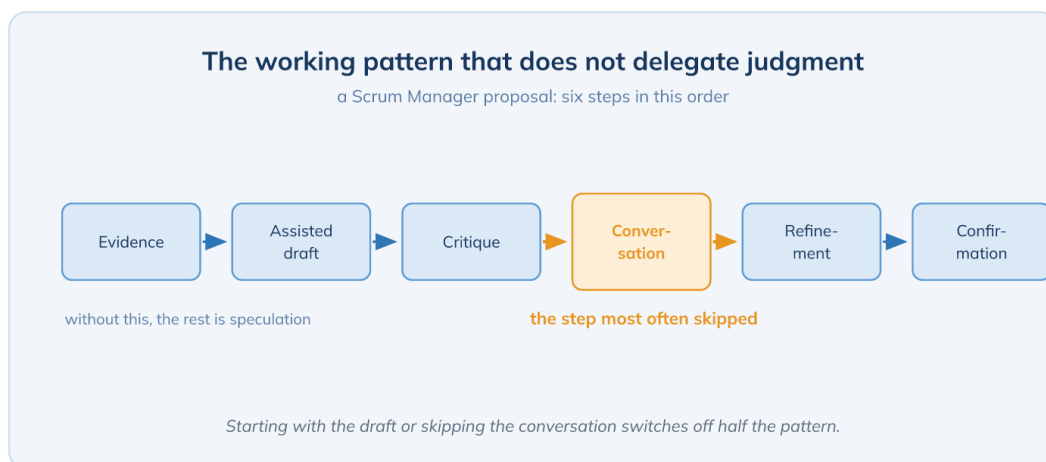
BLOCK G · ARTIFICIAL INTELLIGENCE IN STORY WORK

Chapter 27. Risks and the working pattern that does not delegate judgment · EMERGING

The five risks with their evidence, Scrum Manager's six-step pattern and the role left to the professional.

Introduction

The risks of working with assistance are not the ones usually cited. Whether the model makes mistakes is not the issue, because those can be caught; the issue is that its mistakes arrive with a good finish, in large quantity and at the moment when least review happens. In this chapter we sort them out with the available evidence and propose a working pattern that contains them.



27.1 The five risks

Invention with the appearance of solidity. Already covered in chapter 26: rules, criteria and users that do not exist, written as if they did.

Precision bias. The finish of the text grants authority to the content. Humanizing Work have documented it in product management: a story with well-formatted criteria and round numbers gets questioned less than the same story written by hand, even when its grounding is worse.

Homogenization. Studies on assisted creativity find that individual proposals improve while the set becomes more alike. It is worth stating precisely: that evidence comes from other domains (creative writing, idea generation) and carrying it over to story work is a **reasonable inference, not a measured finding**.

Backlog overproduction. Generating costs less than deciding, so the backlog grows faster than the capacity to order it, and the decay of chapter 7 arrives sooner.

The replacement of conversation by chat. The biggest risk, because it strikes at the technique's reason for being: if the story arrives drafted and with criteria, the conversation looks unnecessary and gets skipped.

Automation bias. Classic research on automation (Parasuraman and Manzey) holds something uncomfortable: the tendency to trust the automated system ****cannot be prevented by training or instructions****. What contains it is process design, not individual prudence.

27.2 Scrum Manager's working pattern

Against the antipattern of «ask for stories and call the result a backlog», we propose a six-step pattern. It is a **Scrum Manager proposal**, built on the evidence of the previous chapters and on the house's practice, not a consensus of the craft:

- 1. Evidence.** Start from what has been observed: interviews, usage data, incidents, experiment results. Without this step, everything else is speculation with a good format.
- 2. Assisted draft.** With the context of chapter 26, ask for a first text: candidate stories, criteria, examples, slices.
- 3. Critique.** Ask for a review against explicit criteria, of your own draft and of the generated one, and obtain a list of defects and gaps.
- 4. Conversation.** The material goes into refinement with the necessary perspectives, including the assumptions and the open questions, not just the result.
- 5. Refinement.** Decide: what goes in, what gets split, what gets investigated, what gets discarded, and fix the agreed rules and examples.
- 6. Confirmation.** Close the observable criteria and record the provenance and the decisions (chapter 13).

The order matters more than the names. The two frequent mistakes are starting at step 2 (without evidence) and skipping step 4 (without conversation), and each one switches off half the pattern.

27.3 The role left to the professional

With drafting made cheap, product work concentrates on six tasks that assistance does for nobody: **giving context** (which requires knowing the product and its evidence), **asking** (which requires knowing what is missing), **cross-checking** (with whoever knows the domain and with the data), **finding the slice** (which requires the judgment of block D), **deciding** (which requires accountability) and **verifying intent** (which requires knowing what was sought).

Put another way: the craft does not shift toward improved writing, it shifts toward judgment. That is this guide's thesis and the reason its classic part takes up six of eight blocks.

Product literature points in the same direction from different angles, with proposals such as the Product Owner as orchestrator and with warnings about the effect of assistance on decision quality. None of those proposals is settled, and it is worth following them with critical attention.

27.4 The cost you do not see

There is one last documented effect worth keeping in mind because it is long term: research on assistance use and critical thinking points to a reduction in cognitive effort and in confidence in one's

own judgment when assistance is used steadily. These are self-reports and do not allow causal inference, so they should not be overstated.

The prudent reading is about work design: it pays to deliberately reserve judgment for the decisions that matter (what goes in, how it is sliced, what gets discarded) and to automate the mechanical without scruples (making formats consistent, detecting duplicates, formatting). Delegating in the reverse order is what produces teams that draft a lot and decide badly.

What you should be able to do

Having mastered this chapter, a professional is able to:

- List the five risks and tell those with measured evidence from those that are inference.
- Explain why automation bias is not corrected by training and what does contain it.
- Apply the six-step pattern and recognize the two frequent order mistakes.
- State the six professional tasks that assistance does not replace.
- Distribute the work between what is worth automating and what is worth reserving for judgment.

Common mistakes and antipatterns

Trusting the finish. An impeccable format tells you nothing about the grounding, and it also discourages the reviewer's criticism.

Generating backlog in bulk. The whole grows faster than it can be ordered and the decay accelerates; better to generate little and decide a lot.

Skipping the conversation because the text is complete. It is the risk that does the most damage, because it removes exactly what makes a story useful.

Presenting the result without the assumptions. The team inherits invisible decisions; bringing the open questions to refinement takes a minute.

Automating judgment and doing the mechanical by hand. It is the reverse of the sensible split: the mechanical is where assistance pays off without risk.

Further reading

- [Parasuraman and Manzey — Complacency and bias in human use of automation](#)
- [Humanizing Work — Precision bias and AI in product management](#)
- [Humanizing Work — Two AI + backlog pitfalls](#)
- [Lee et al. — The impact of generative AI on critical thinking](#)
- [Skill Arena — AI applied to agile work](#)

BLOCK H · THE STORY AS THE INPUT TO AN AGENT WORKFLOW

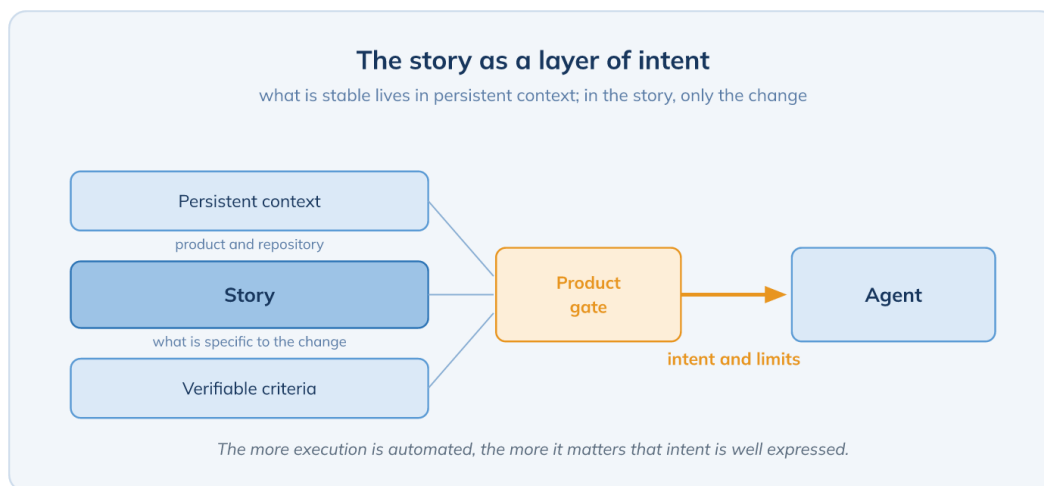
Chapter 28. From story to spec: intent as the source of truth · EMERGING

What agent development tools do today, what relationship the spec bears to the story and how context is distributed.

Introduction

This last block deals with what changes when the implementer of a story is, wholly or partly, an artificial intelligence agent. It is the most recent material in the guide, with tools that are months old and opposing positions, so we will treat it with the evidence labels up front and without taking as settled what is not.

It is worth saying at the outset what we will not cover here. The complete spec-driven development flow, with its phases, gates and tools, has its own skill in Scrum Manager, as does the coexistence of stories and specs according to risk. This block looks at one thing only: what happens to the story, and what product must do.



28.1 What the tools do today

Agent development tools work on an intermediate artifact, the **specification** (spec), and it is worth knowing what it contains, because it is not what many assume. Kiro's documentation (Amazon, July 2025) describes specs that include user stories with their acceptance criteria in structured notation, and a specific type for bug fixing that explicitly declares the behavior that **must not change**.

GitHub Spec Kit's template (September 2025, version 1.0 in August 2026) is even more telling: it contains prioritized stories marked as independently testable, acceptance scenarios, edge cases, declared assumptions and a literal marker for what needs clarification. That is, everything this guide teaches, arranged in a document.

The spec contains the story. It does not replace it. What happens is that product work (intent, rules, examples, limits) becomes the critical part of a document that used to be technical. The Scrum Manager BoK fixes it this way: spec-driven development does not replace user stories, it complements them.

28.2 The doctrine and its critics

The strong formulation of the trend says the specification is the new code and intent is the source of truth: if code can be regenerated from intent, what is valuable is well-expressed intent. It is a suggestive idea with obvious consequences for product work.

It has serious critics and they are worth knowing. The main objection is that a flow requiring the complete specification to be written before building reintroduces the waterfall cycle under another name, with its same weakness: the detailed specification of what is not yet understood.

Thoughtworks's field analyses are more nuanced and note that these tools can be a sledgehammer to crack a nut, with specifications out of all proportion to small changes.

The prudent position, and the one this guide adopts, is to treat it as a **trend under evaluation**: it is worth knowing the format because it already reaches teams, worth contributing from product what only product knows, and not worth rebuilding the way of working around a tool that is months old.

28.3 Context layers

There is one practical finding that is well established and that does change how a story is written in this setting: context is organized in layers, and repeating what is stable in every prompt **degrades** the result.

Persistent layer. What does not change between tasks: what the product is, who its users are, the domain glossary, the repository conventions, the current decisions. It lives in the context files the tools read automatically and is maintained as a team asset.

Task layer. Only what is specific to this change: the intent, the rules that apply here, the examples, the limits and the doubts. That is what goes in the story.

The reason for that separation is measured: instruction following degrades as their number grows, so a story that repeats the product's context competes with itself. It is exactly the same principle as the card with «just enough to remember», twenty-five years later and for a different reason.

28.4 What this means for story work

Three consequences, and none of them forces a change of method. First, the detail of the story stops being optional in the stretch the agent will execute: not because more has to be written, but because **what used to be said out loud** has to be written. Second, the conversation does not disappear, it moves earlier: it happens before the story is written, and its result is what gets written. Third, verification gains weight, and that is the subject of chapter 30.

At Vantia, the first story the team handed to an agent was the tax-deadline warning. The story that had served to talk with the team (one sentence, three criteria) turned out to be insufficient for the agent at one specific, predictable point: it did not say what **must not** change. The agent rewrote the email template already used by the rest of the platform.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Explain what a specification in today's tools contains and what part the story plays inside it.
- Place spec-driven development as a trend under evaluation, with its doctrine and its criticisms.

- Distribute context between the persistent layer and the task layer, and justify why it is not repeated.
- Anticipate which part of what used to be talked through has to be written when an agent executes.

Common mistakes and antipatterns

Writing the complete specification up front. It is the well-founded criticism of this trend: specifying in detail what is not yet understood reproduces the waterfall problem.

Repeating the product context in every story. It degrades instruction following and multiplies diverging versions of the same context.

Rebuilding the process around the tool. The tools are months old; product judgment is decades old, and that is what has to be contributed.

Assuming the spec replaces the conversation. The conversation moves earlier, it is not removed: its result is what the specification records.

Further reading

- [GitHub Spec Kit — spec-driven development](#)
- [Kiro — feature specs](#)
- [Böckeler — Understanding spec-driven development](#)
- [Zaninotto — Spec-driven development: the waterfall strikes back](#)
- [Scrum Manager BoK — Spec-Driven Development \(in Spanish\)](#)
- [Skill Arena — SDD in agile teams](#)

BLOCK H · THE STORY AS THE INPUT TO AN AGENT WORKFLOW

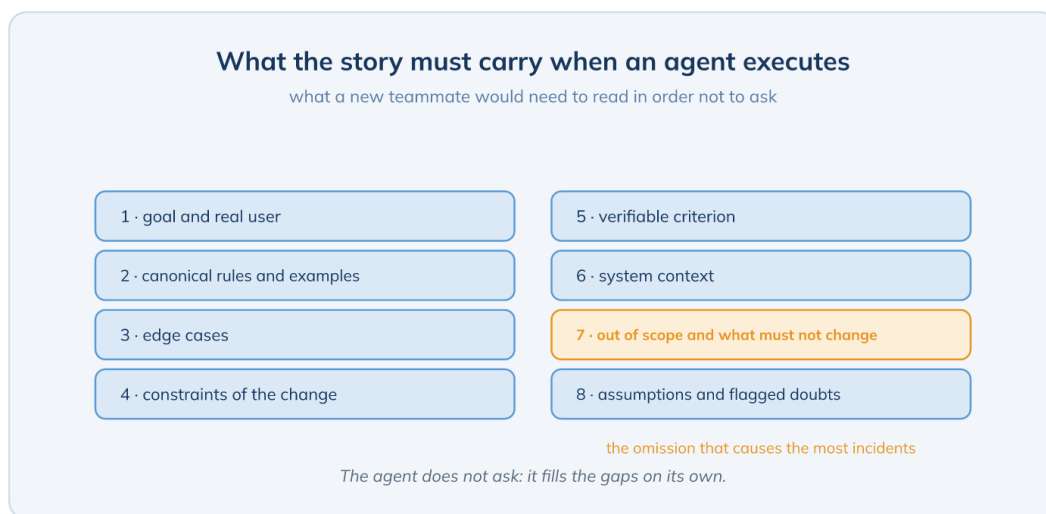
Chapter 29. What the agent needs and the criterion that verifies itself

EMERGING

The eight elements the story must carry, doubts as a deliverable and the rereading of INVEST under agents.

Introduction

There is one rule that sums up this chapter and it is worth keeping in mind before the detail: the story must contain what **a competent new teammate** would need to read in order to do the task without asking. No more, because excess degrades; no less, because the agent does not ask.



29.1 The eight elements

- 1. Goal and real user.** What it is for and for whom, with the same criterion as chapter 1. Without this, the agent optimizes what it can measure.
- 2. Canonical rules and examples.** The rules that apply to this change and two or three examples that pin them down, not an exhaustive battery.
- 3. Edge cases.** The boundary values and the negative cases that matter (chapter 11).
- 4. Constraints and non-functional requirements of the change.** The ones that apply here, not the system's general ones (those are in the persistent layer).
- 5. A verifiable success criterion.** How it will be checked, preferably by machine (29.3).
- 6. System context.** Which files, components or existing patterns are relevant, and where to find a similar example already solved.
- 7. Out of scope and behavior that must not change.** The most forgotten element and the one that prevents the most incidents. It is what was missing in Vantia's deadline-warning story.
- 8. Declared assumptions and flagged doubts.** What has been assumed and what is not known, marked as such.

Doubts are a deliverable. The agent does not ask: it fills gaps on its own and inconsistently. Recent work on coding agents confirms that, faced with ambiguous statements, they do not request clarification. That is why a flagged doubt is worth more than an invented decision.

29.2 Few criteria, and canonical

The natural temptation is to load the story with criteria, and it backfires for a measured reason: joint instruction following **falls** as their number grows. With twenty criteria, the probability that all of them are respected is low, and there is no way of telling which one was ignored.

The practice that works is chapter 10's taken to the extreme: few criteria, canonical (fixing the rule rather than enumerating cases) and verifiable. Particular cases go in as examples, which illustrate without multiplying instructions.

The tool guides agree. GitHub's coding agent documentation asks for complete acceptance criteria on what a good solution looks like; Claude Code's insists on stating what is out of scope and on not shipping what you cannot verify; and Devin's bounds task size to a few hours of work, which is the scale of chapter 14.

29.3 From observable criterion to test

A well-written acceptance criterion is the seed of machine verification, and that is where the work of block C pays off most in this setting. The chain is direct: the agreed rule and its concrete example become a declarative scenario (chapter 12), and the scenario becomes a test the agent can run to check itself.

Research on generating acceptance tests with language models gives promising results in terms of the usefulness of the generated scenarios, with the expected caveat that some need regenerating or discarding. That is: generation helps, and approving the scenario remains human, because the scenario **is** the criterion.

For probabilistic functionality, chapter 10 applies: rubric, evaluation set and threshold. It is the only way for an acceptance criterion to be verifiable when there is no single correct answer.

29.4 INVEST when an agent executes

Chapter 4's acronym still serves, with shifts we propose as a **Scrum Manager rereading** rather than as craft consensus:

Independent and **small** reinforce each other. With fast execution and a low cost per attempt, the small, independent piece is even more advantageous: it gets verified sooner and discarded without regret.

Testable becomes **machine-verifiable**. Being able to write an example is not enough: it pays to have an executable check, because that is what allows delegating execution without delegating judgment.

Negotiable moves earlier. Negotiation stops happening during implementation (the agent does not negotiate) and shifts to the prior conversation and to the gate of chapter 30. Here lies the open debate worth declaring: moving negotiation earlier looks like fixing the detail too soon, which is

precisely what the technique set out to avoid. There is no settled answer, and the prudent criterion is to move earlier only what the agent needs in order not to invent.

Valuable and **estimable** do not change in kind, although sizing becomes more important: tasks that fit in a few hours of agent work are the ones that work best, and that pushes toward the fine slicing of block D.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Prepare a story with the eight elements and check that out-of-scope is not missing.
- Flag doubts as a deliverable instead of resolving them by guesswork.
- Write few canonical criteria and move particular cases into examples.
- Turn an observable criterion into a declarative scenario and into an executable check.
- Reread INVEST in this setting and explain the shift in each property.
- Recognize the open debate about moving negotiation earlier and adopt the prudent criterion.

Common mistakes and antipatterns

Omitting what must not change. It is the omission that produces the most incidents: the agent rewrites what was there, because nobody said it was fine as it was.

Loading the story with criteria. Joint compliance degrades with the number, and there is no way of telling which one was ignored.

Resolving doubts by guesswork «so as not to block». A written guess becomes a requirement; a flagged doubt becomes a conversation.

Repeating the system context in every story. Its home is the persistent layer; repeated, it competes with what is specific to the change.

Delegating the approval of scenarios. The scenario is the acceptance criterion: approving it is a product decision, even if a model drafted it.

Further reading

- [Spec Kit — specification template](#)
- [GitHub — getting the best results from the coding agent](#)
- [Claude Code — best practices](#)
- [Ferreira et al. — acceptance test generation with LLMs](#)
- [Report on the curse of instructions](#)
- [Kiro — spec best practices](#)

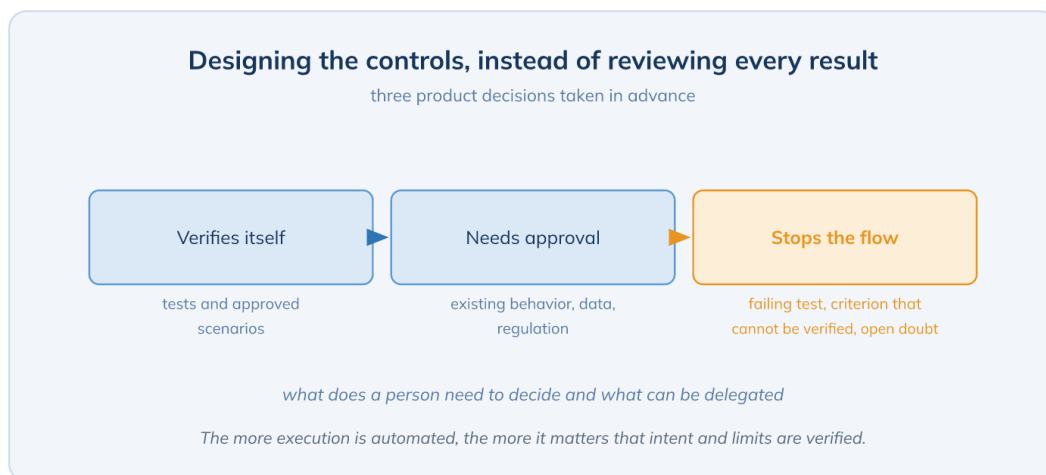
BLOCK H · THE STORY AS THE INPUT TO AN AGENT WORKFLOW

Chapter 30. The requirements gate from product · EMERGING

What product reviews before an agent starts, how the loop's controls are designed and what the evidence says about productivity.

Introduction

We close the guide with the part that belongs to product when execution is automated: a light gate before the agent starts, and a new role that consists of designing the controls of the process rather than reviewing every result.



30.1 The six points of the gate

The gate is not a Definition of Ready under another name (chapter 7 explains why that drift is damaging): it is six checks done in two minutes, and its purpose is to prevent automated work on the wrong intent.

- 1. The intent is the real one, not the attributed one.** That what the story asks for is what is actually wanted, and not the solution somebody assumed.
- 2. The constraints and what must not change are stated.** The most forgotten point of chapter 29.
- 3. The criteria are observable.** And, if possible, machine-verifiable.
- 4. The scope is manageable.** Of the size the agent completes reliably, which today is a few hours of work.
- 5. The what carries no technical decisions inside.** Except those that are genuine system constraints.
- 6. The doubts are flagged.** And the blocking ones resolved before starting.

The question that organizes refinement. Dave West put it precisely for this setting: what does a human need to decide here, and what can we delegate to AI? It is the question to review any story with before handing it to an agent.

30.2 On the loop, not in the loop

With a volume of automated execution, reviewing every result stops being viable, and the role shifts to **designing the controls**: what gets verified automatically, what requires human approval, what can be undone at no cost and what cannot, and at which point the flow stops if something does not add up.

Applied to story work, that means deciding three things in advance. Which checks accompany each story (tests, approved scenarios, a person's review). Which changes require explicit approval before being integrated (those affecting existing behavior, customer data or regulatory obligations). And which signal stops the process (a failing test, a criterion that cannot be verified, a doubt that reappears).

It is a product responsibility rather than an engineering one, because the three decisions are about risk and value, not about implementation.

30.3 The risks of this setting

More documentation without more understanding. The biggest risk: extensive specifications nobody has talked through, with the same misunderstanding as always and better presentation.

Automated assumptions. An unflagged assumption becomes implemented behavior, and is discovered when somebody uses it.

Fast propagation of errors. A badly written criterion runs in minutes and in many places; the cost of a misunderstanding is no longer cushioned by the slowness of the process.

False precision. Structured text and formal notations lend an appearance of exactness to decisions nobody has made.

30.4 What the evidence says about productivity

Here it is worth being especially careful, because exaggeration is the norm. The METR study (July 2025) measured expert developers working in their own repositories with and without assistance, and found they were **19 % slower** with assistance, while they believed they had been faster. It is a study with a small sample and in a specific context, and it does not license the conclusion that assistance is useless; it does license distrust of perception as a measure.

The DORA 2025 report provides the panorama: adoption is near universal (around 90 %) and confidence is much lower (close to a third report little or none). And it offers one useful fact for this guide: among the capabilities it associates with AI performance are **working in small batches** and **user-centric focus**, which are exactly the content of blocks D and A.

The conclusion this guide holds, and the one it closes with: the more execution is automated, the more it matters that intent and limits are well expressed and verified. The technique this material teaches does not become obsolete with agents; it moves to the center.

What you should be able to do

Having mastered this chapter, a professional is able to:

- Review a story against the six points of the gate before handing it to an agent.

- Formulate the question that splits what a person decides from what can be delegated.
- Design the controls of the process: what verifies itself, what requires approval and what stops the flow.
- Recognize the four risks of this setting and the practices that contain them.
- Read the evidence on productivity prudently and distrust perception as a measure.

Common mistakes and antipatterns

Turning the gate into a Definition of Ready. Chapter 7's drift returns: a veto form instead of two minutes of checking.

Reviewing every result. It does not scale and shifts product work into inspection; what has to be designed are the controls.

Trusting the feeling of speed. The available evidence shows perception can run contrary to the measurement.

Specifying more in order to understand better. Shared understanding does not grow with document length, and that is the thread the guide began with.

Further reading

- [METR — study with experienced developers](#)
- [DORA — 2025 report](#)
- [DORA — capabilities](#)
- [Böckeler — humans and agents in the loop](#)
- [Scrum.org — Is your Product Backlog ready for AI agents?](#)
- [Skill Arena — Scrum with AI teams](#)

© 2026 Scrum Manager®. This work is published under a Creative Commons Attribution-NonCommercial 4.0 International licence (CC BY-NC 4.0). Official Scrum Manager trainers and centres are licensed under the terms of CC BY 4.0 for their training activity.