USER STORIES

A USER STORY IS A SIMPLE, INFORMAL WRITTEN EXPLANATION OF AN INTERACTION THAT PRODUCES VALUE

A USER STORY IS A DESCRIPTION OF A NEED IN SIMPLE TERMS FROM THE CUSTOMER'S POINT OF VIEW

ACCEPTANCE CRITERIA

I'LL KNOW IT'S DONE WHEN THESE CONDITIONS ARE MET:

S – SPECIFIC
THEY ARE CONCRETE AND CLEAR.

M – MEASURABLE
I CAN CHECK THAT THEY HAVE BEEN MET USING OBJECTIVE EVIDENCE.

A – ACHIEVABLE
THEY ARE REALISTIC AND ATTAINABLE.

R – RESULT-FOCUSED
THEY DETERMINE THE BENEFITS OR VALUE TO BE ACHIEVED.

T – TIME-BOXED
THEY RELATE TO A SPECIFIC TIME FRAME.

HELPFUL TIPS

USER STORY

As a...
I want...
So that...

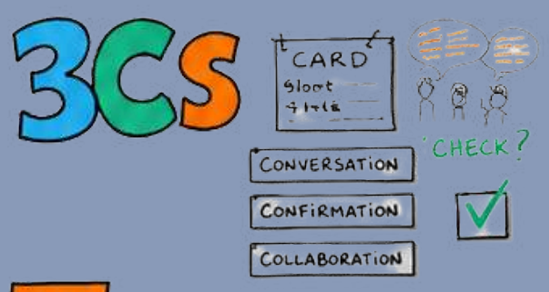
STRUCTURE

AS A USER / ACTOR
I WANT
TO [OBJECTIVE]
SO THAT [REASON]

ACCEPTANCE CRITERIA
[RESULT / OBJECTIVE]

GOOD PRACTICES

3Cs



INVEST

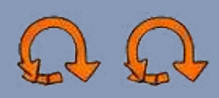
- INDEPENDENT
- NEGOTIABLE
- VALUABLE
- ESTIMABLE
- SMALL
- TESTABLE

KEEP IT SHORT AND SIMPLE

DELIVER VALUE TO THE CUSTOMER

IF WE DON'T, LET'S NOT DO IT. LET'S DELETE IT.

SPRINT 1



YOU SHOULD BE ABLE TO DELIVER IT.

WE CAN ESTIMATE IT.

IT SHOULD BE AS SMALL AS POSSIBLE.

YOU SHOULD BE ABLE TO TEST IT.

@MIKE_NEPHEW

User stories

Complementary training

Version 5.0.

Scrum Manager ®

Version date: august 2026.
Cover design: Miguel Ángel Sobrino.
Authors, in collaboration with Scrum Manager: Alexander Menzinsky, Gertrudis López, Juan Palacio, Miguel Ángel Sobrino, Rubén Álvarez, and Verónica Rivas.

Index

Agile requirements engineering

User stories

Requirements engineering has two means of communication for capturing and conveying requirements, each with its own characteristics:

1. Written communication.

- It records information permanently.
- It is easier to share with groups and remote staff.
- It can be thought through and reviewed carefully and in full.
- It is more prone to misinterpretation.

2. Verbal communication.

- It provides immediate feedback.
- It is dynamic: the conversation adapts to maximize its efficiency.
- It adapts easily to new developments.
- It generates new ideas.
- It makes it easier to reach shared understanding and clarity with less effort.

In agile requirements engineering, user stories are used as a communication tool that combines the strengths of both means: written and verbal. In one or two sentences, they describe a piece of software functionality from the user's point of view, in the language the user would use (Cohn, 2004). The focus is on the needs or problems that what is being built will solve.

They originate from the Extreme Programming (XP) methodology, where user stories are meant to be written by the customers. XP was created by Kent Beck and first described in 1999 in his book *Extreme Programming Explained*. Today they are used in most agile methods, including Scrum.

User stories streamline requirements management, reducing the amount of formal documentation and time involved. They are part of the three Cs formula, defined by Ron Jeffries in 2001, for capturing functionality:

- **Card:** each user story is pared down until it is easy to remember and to capture on a card or sticky note. The card serves as a reminder and a promise of a later conversation.
- **Conversation:** the development team and the Product Owner add acceptance criteria to each story shortly before it is implemented. Agility welcomes change, so there is no point in fleshing out these details any earlier. A great deal can shift between the moment the functionality is captured on the card and the moment it is implemented.

- **Confirmation:** the Product Owner or business user reviews the acceptance criteria to confirm that the development team has correctly understood and captured their requirements. These criteria are sometimes expressed as test scenarios.

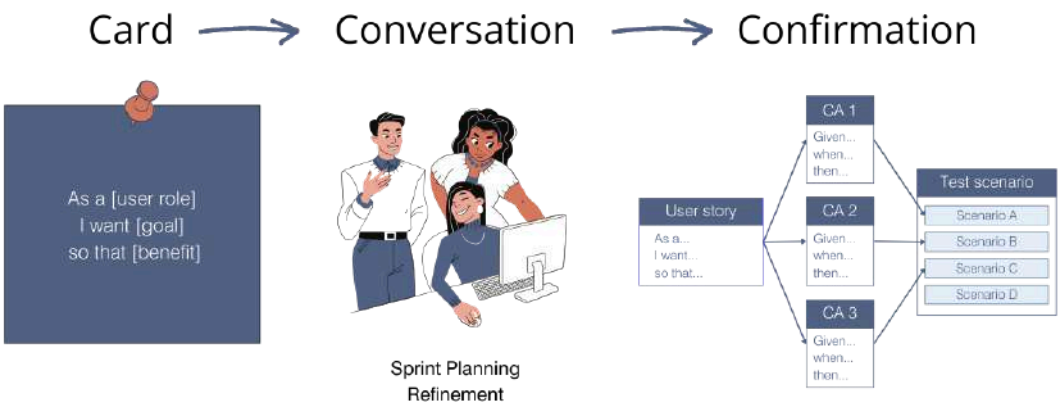


Figure 1: The three aspects of the three Cs formula.

Advantages of user stories

- They provide the necessary documentation while encouraging discussion.
- They foster collaboration between all stakeholders and the agile team.
- They are written in the user's language, keeping a close relationship with the customer.
- They engage and involve the customer in both the process and the product.
- By nature, they are independent.
- They make planning and implementation easier.
- They are ideal for projects with volatile or unclear requirements.
- They encourage deferring non-essential details.
- They are small and therefore easy to work with.
- They allow projects to be split into small deliveries.
- They make it easy to estimate the development effort.
- They work well for iterative development: because they are small, they represent business-model requirements that can be implemented in a short time (days or weeks).
- They require little maintenance.

In today's context

Since their origins in Extreme Programming, user stories have evolved to fit contemporary software development contexts:

- **Distributed and remote teams:** user stories support asynchronous collaboration through digital tools (Jira, Azure DevOps, Trello), keeping the balance between written documentation and conversation.

- **DevOps and continuous delivery:** acceptance criteria are automated as tests, enabling continuous integration and deployment.
- **Agile scaling:** some scaling frameworks introduce additional levels (for example, features and capabilities in SAFe), while others keep the Scrum artifacts and coordinate multiple teams around a shared backlog (for example, LeSS and Nexus).

Despite these technological advances, the core principle remains: user stories are communication tools that spark conversations and build shared understanding between business and technology.

In other words, **they are not complete specifications;** they are reminders to have a conversation. In keeping with that purpose, estimation relies on relative scales, and AI can be used as an assistant to speed up mechanical tasks—always keeping human validation and conversation in the loop.

From themes and epics to tasks

In the hierarchy of agile requirements, epics and themes sit above user stories. What they have in common is that they all focus on describing what will be built.

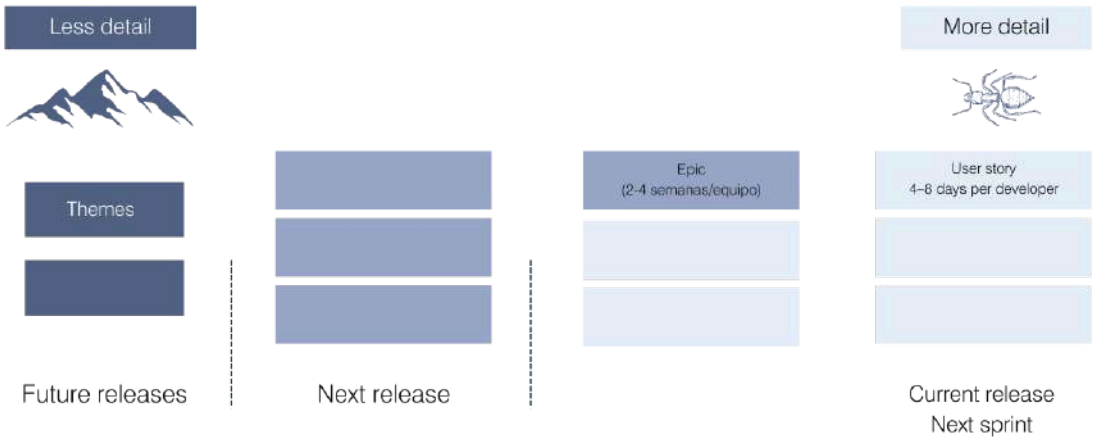


Figure 2: Product Backlog granularity.

Epics

An epic is a large user story that carries a greater degree of uncertainty[1]. It is like a label we assign to a story whose effort makes it impossible to complete in one go or within a single sprint.

Marking a story as an epic means it cannot be completed in one go or within a single sprint. The development team will normally break it down as its implementation draws near. The resulting user stories will be closely related, and their smaller size will

[1] Granularity refers to the grain size into which the work in the Product Backlog is divided: epics are coarse-grained (large, low-detail items) and tasks are fine-grained (small, detailed items). The higher an item's priority, the finer its grain should be.

make them easier to manage in an agile way—estimating the time needed to complete them more accurately and tracking their progress in detail.

Every epic and story is, at heart, a hypothesis: until the user sees or uses it and gives us feedback, we won't know whether it is the right solution. From this perspective, we can think of an epic as a series of experiments—future user stories—that build the functionality incrementally until the desired outcome is reached.

This way of developing epics is known as the Lean Startup cycle. It sets out to develop new ideas, products, and services through hypothesis-driven development (HDD). Thanks to the feedback gathered as hypotheses are validated, the Product Owner gains a clearer understanding of the most suitable solution. Barry O'Reilly proposes the following pattern in his article "How to Implement Hypothesis-Driven Development":

> *We believe [this capability]*
> *Will result in [this outcome].*
> *We will have confidence to proceed when [we see a measurable signal].*

1. We believe [this capability]

What functionality will we develop to test our hypothesis? By defining the testable capability of the product or service we are trying to build, we identify the functionality and the hypothesis we want to test.

2. Will result in [this outcome]

What is the expected outcome of our experiment? What specific result do we hope to achieve by developing the testable capability?

3. We will have confidence to proceed when [we see a measurable signal]

What signals will indicate that the capability we have built is effective? Which key leading indicators—qualitative and/or quantitative—will we measure to know the experiment has succeeded and move on to the next stage?

> ***We believe*** *that increasing the number of restaurant photos on the booking page* ***will result in*** *better customer engagement and conversion.* ***We will have confidence to proceed when*** *we see a 5% increase in customers who review the restaurant images and then go on to book within the next 2 minutes.*

Theme

A collection of related epics and user stories that describe a system or subsystem. It is part of the product vision rather than a single piece of functionality.

For example, in an accounting software system, the set of epics "adding, removing, and maintaining customers," "one-off and recurring billing," "browsing queries and loyalty actions," "orders," and "returns" could be grouped under the theme "customer management."

Tasks

Tasks sit below user stories. They describe how to build rather than what. They result from breaking user stories down into work items suitable for managing and tracking execution progress.

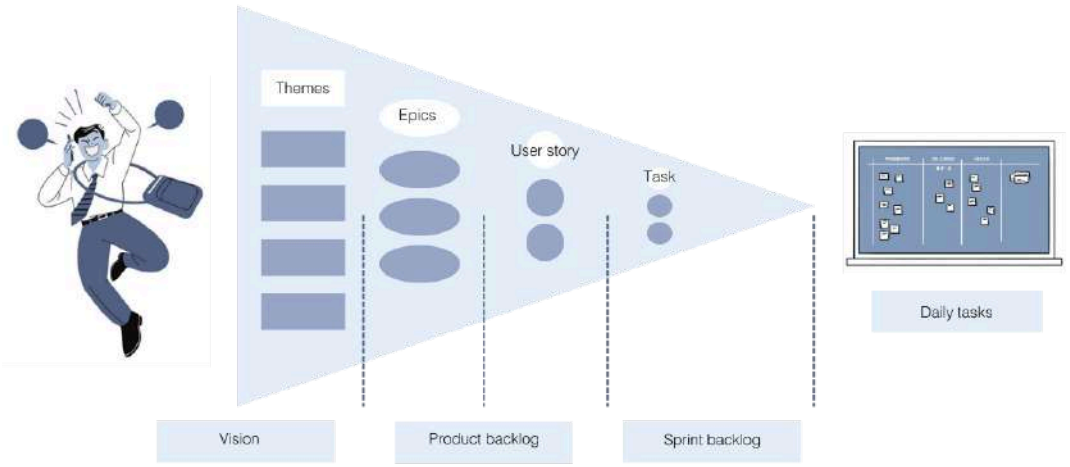


Figure 3: The four size levels at which agile management handles requirements.

***

In Scrum, and in agile methods generally, a Product Backlog can contain user stories as well as themes and epics.

At the bottom of the Product Backlog sits the lowest-priority work: epics (large user stories) and themes, which describe the more general requirements of the product vision. Each of these is broken down into smaller items as it moves up the backlog.

Stories that may enter the next sprint need more detail so the team can break them into tasks during Sprint Planning. User stories are therefore the items that should sit at the top of the Product Backlog.

Anatomy of a user story

To decide what information to include in a user story, it is best not to adopt rigid formats. The results of Scrum and agility do not depend on forms, but on institutionalizing their principles and implementing them appropriately for the characteristics of the company and the project. That said, keeping a consistent structure within the team and organization makes communication and shared understanding easier. Flexibility does not mean arbitrariness.

So, aside from three fields considered necessary, you can include any field that provides useful information for the project. Remember that **the goal of user stories is to build shared understanding.**

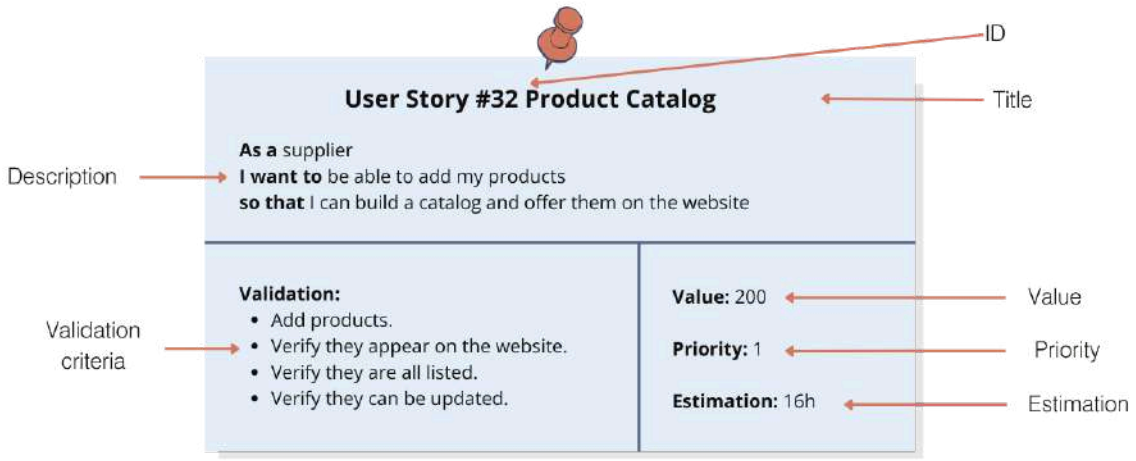


Figure 4: Example of a user story card.

Essential fields

The fields considered most necessary to describe user stories are:

- **Description:** a summary of the user story. The style can be free-form, but it must answer three questions: who benefits? what do they want? and what is the benefit? Mike Cohn recommends following the pattern below to keep the functionality described at a high level and concisely.

> **As a** [user role], **I want** [goal] **so that** [benefit].

- **Estimate:** an approximation of the relative size of the work needed to implement the story. It can be estimated on a relative scale (story points[2]) if the team prefers and is familiar with this system.

[2]Story points are units that express the relative size of the work (complexity, uncertainty, risk, and effort) compared against a reference agreed by the team.

- **Priority:** indicated using a system that establishes the order in which stories are implemented.

Other fields

Depending on the type of project, how the team works, and the organization, other fields may be advisable, such as:

- **ID:** a unique identifier for the user story, feature, or piece of work.
- **Title:** a descriptive title for the user story.
- **Business value:** the value (usually numeric) that the user story provides to the customer or user. The team's goal is to maximize the value and satisfaction perceived by the customer in each iteration. Together with the estimate, this field helps decide implementation priority.
- **Acceptance criteria:** acceptance tests agreed with the customer or user. They are sometimes turned into tests the code must pass for the implementation to be considered complete.
- **Non-functional requirement:** general qualities and constraints, such as usability or security, that affect entire applications and systems, and therefore individual user stories.
- **Definition of Done (DoD):** the activities or criteria needed to consider a user story finished (developed, tested, documented, and so on), as agreed by the Product Owner and the team.
- **Dependencies:** a user story should not depend on another, but sometimes the relationship must be kept. This field would contain the identifiers of the other stories it depends on.
- **Assignee:** used when we want to suggest who might implement the user story. Remember that in Scrum the team is self-managing and is the one that distributes and assigns tasks.
- **Sprint:** it can help the Product Owner stay organized to include the number of the sprint in which the story is expected to be built.
- **Risk:** the technical or functional risk associated with implementing the user story.
- **Module:** the system or product module it belongs to.
- **Notes:** to enrich or clarify the information, or for any other necessary use.

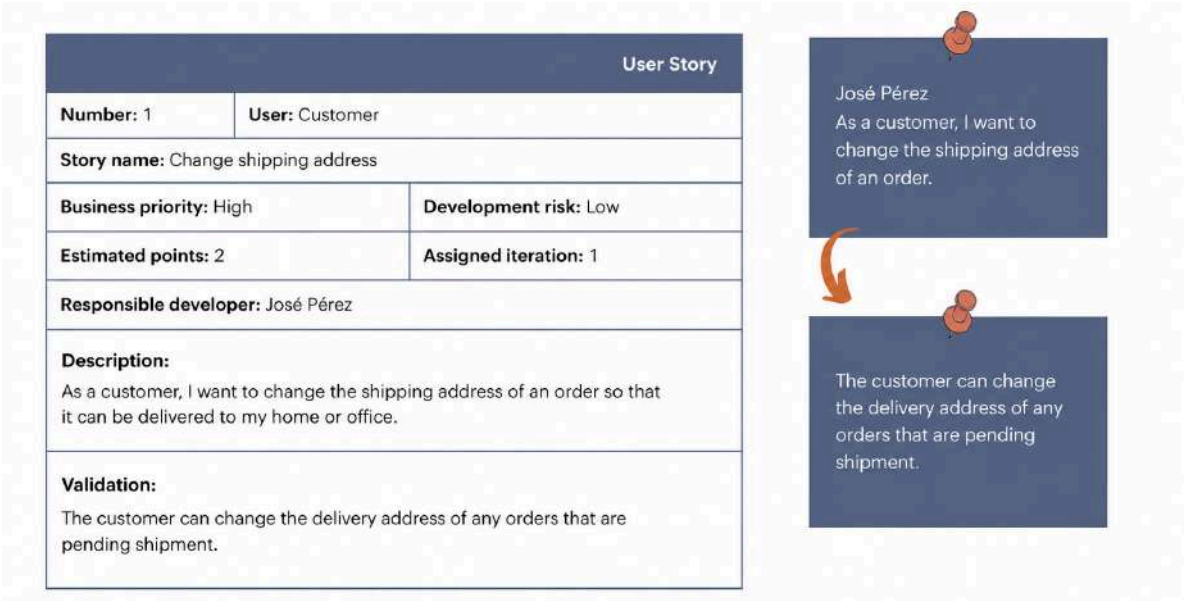


Figure 5: Examples of a user story.

Mike Cohn notes that, while user stories are flexible enough to describe the functionality of most systems, they are not suitable for everything. If for any reason a requirement needs to be expressed differently, it is advisable to do so.

For example, the layout of an interface is usually described with screenshots—the best way to convey the design we want to give an application.

Description

People's thinking is structured as a narrative, a story; that is how we understand the world. It is also an effective way to retain knowledge and make decisions. By using characters and empathizing with them, we can understand their point of view and prioritize more accurately.

A useful technique here is the creation of user personas, widely used in the field of user experience (UX). The goal is to understand users' points of view: to build a set of fictional characters that represent, through archetypes, a realistic sample of the key audience of our website or application.

These user personas will serve as the "user role" in the pattern we saw earlier for describing the story:

> **As a** [user role], **I want** [goal] **so that** [benefit].

- **As a [user role]:** who are we building for? We do not mean a job title or a professional role, but the person behind the described need, as represented in

their user persona. The team must have a shared understanding of who they are and how they work, think, and feel—in short, empathy.

- **I want [goal]:** what are they really trying to achieve? It is important to remember that this field explains their intention, not the functionality they will use.
- **So that [benefit]:** how does their immediate need fit into the bigger picture? What is the overall benefit they are trying to achieve? What is the larger problem they need to solve? This is a benefit tied to the product vision.

These characters make it easier to keep conversations focused on users, giving them a name and a personality so we can empathize with the group they represent. If our website's target audience has certain characteristics, creating its user persona and referring to it by nickname lets us picture more clearly how it interacts with the system, helping the team make better decisions and offer solutions better suited to real needs.

Users are usually represented on a fact sheet, tri-fold style, including for example:

- **Name and nickname:** a quick way to identify and recognize the character.
- **Demographics:** help classify the character within a market segment.
- **Description:** a short paragraph that helps us get to know the character.
- **Goals:** the user's needs that set their behavior apart from the rest.

Figure 6: Example of a user persona.

Business value

The "value" field of a user story represents the business value it will provide once completed. It offers deeper information than the story's rationale, since it is tied to the business logic. One way to estimate it is to consider how much the customer would be willing to pay for the functionality.

Given how Scrum and other agile methodologies work, delivering value to the customer means focusing on solving someone's problems or providing benefits to someone. Knowing the business value is very important for prioritizing correctly and focusing the team. There are also other factors to consider, which is why value and priority are two independent pieces of information.

Value is measured on an arbitrary, usually numeric, scale. It should be one the Product Owner and the users feel comfortable with—for example, the Fibonacci series, or ranges of natural numbers such as 1 to 10, 1 to 100, or 1 to 1,000,000.

One very engaging approach is to hand out Monopoly money equal to the project's total value to each of the business users involved, and have them distribute the money according to the value they assign to each user story. The final value is the sum of everyone's notes. The result can also be divided by 100 or 1,000 to give a more manageable number.

Another technique, known as **value poker**, consists of estimating value the same way the team estimates effort: with Planning Poker cards using the Fibonacci series. Value, like effort, is additive and relative between stories. The idea is for the Product Owner and the business experts to meet and estimate the value of the stories through Planning Poker, creating a shared knowledge base for users and the business side, just as a common base is built in the team's meetings.

This practice has **two very interesting side effects:** it fosters the team's respect for and understanding of the Product Owner, and it helps the Product Owner better understand why the team sometimes needs to re-estimate certain stories.

It is worth remembering that **a story, as an agile requirement, is only a hypothesis to validate that we estimate will benefit the user.** We assess it first from the business point of view, with the information available. But we won't know whether the hypothesis was correct until we see the customer's real response. This is why using the business value delivered by a team in each sprint or release as a metric is not recommended. Business value should be understood as an input for prioritizing the Product Backlog.

Estimation

As we have seen, this is part of the essential information in a user story. Estimating effort helps the Product Owner prioritize and helps the team decide which stories from the backlog fit into the sprint.

One thing to consider is that the uncertainty in estimating small user stories is much lower than for large ones. Moreover, although the size of stories grows linearly, uncertainty grows exponentially.

A numeric series well suited to this task is the Fibonacci series, made up of numbers that are the sum of the two preceding ones. The distance between the numbers in the series reflects the fact that the uncertainty inherent in estimation grows with the size of the story. So the differences between 1, 2, and 3 story points are probably better understood than the differences between 13, 21, and 34.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

There are also reasons rooted in human nature for using this series. We find it easy to estimate by determining how much smaller or larger something is compared to something else, which is why Scrum and other agile methods use relative estimation techniques. But we tend to think in multiples of two. For example, we compare a user story with a 2-point one, see that it is larger, and automatically think of 4, 8, or 16. The Fibonacci series forces us to break this habit and look for the most suitable estimate.

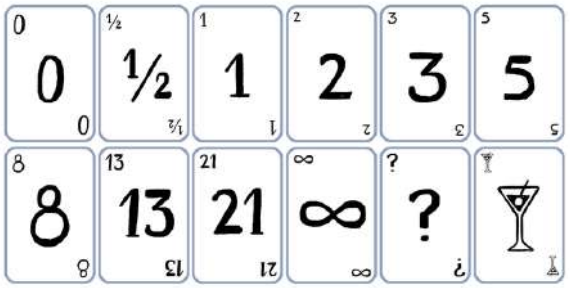


Figure 7: Planning Poker cards with the Fibonacci series.

The decks typically used in Fibonacci-based estimation may include variations. Some add 0 for stories that require almost no effort and 1/2 for very small stories. Some decks replace 21 with 20, since saying a story has a size of 21 gives a false sense of precision. Beyond 21, decks usually include an infinity symbol (∞) and, if they continue, round values such as 40 and 100.

The goal of Planning Poker is not to make predictions that can be confirmed later, but to offer a quick estimation process that gives the team useful values for estimating the Product Backlog or planning the sprint. More important still is getting everyone to understand the item being estimated: if participants estimate with widely differing values, it means someone knows something the others don't. Remember that teams

include specialists from different areas, optimists and pessimists, and that makes some see solutions others don't. These discussions help align the whole team's knowledge.

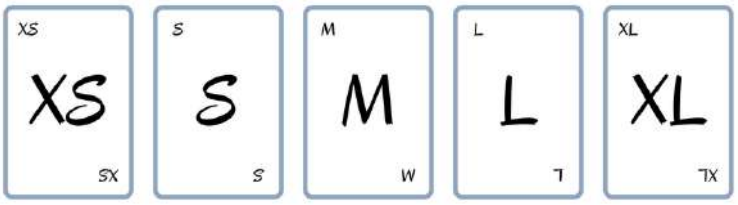


Figure 8: Estimation cards with T-shirt sizes.

For estimating large items such as epics and themes, the Fibonacci series is not appropriate: its numbers would give a sense of precision that does not exist. One solution is to use a technique based on T-shirt sizes. Sizes serve to sense rather than estimate, with the high degree of uncertainty that overly large, loosely defined items call for. This technique has the advantage of being purely relative, since its values cannot be translated into time, workdays, or hours.

With both techniques we can estimate the entire Product Backlog in a coherent, agile way. The items in the priority part of the backlog are detailed, including the estimate, while the more distant items are coarser-grained, with a vaguer, broader estimate.

The Product Owner relies on these results to prioritize the Product Backlog. They can see the most immediate items more precisely while still having enough detail for the distant, low-priority ones.

Priority

Although every user story may be important, to focus efficiently we must highlight the ones that bring the most value to the system. The Product Owner must assign a value to each story that takes part in the prioritization system, taking the following variables into account:

- The benefits of implementing the functionality.
- The loss or cost of postponing its implementation.
- The risks of implementing it.
- Consistency with the interests of the business.
- The differential value compared to competitors' products.

One aspect to consider is that the definition of value can differ for each customer. It is advisable to use a qualitative scale, with intrinsic meaning, that provides more information than simply whether the priority is high, medium, or low.

That is the case with the **MoSCoW technique**, in which the user responsible for assigning priority is aware of the real effect their choice will produce. This technique was first defined in 1994 by Dai Clegg of Oracle UK Consulting in the book *Case Method Fast-Track: A RAD Approach*. Its purpose is to reach a common understanding of the project between customer and team, specifically about the importance of each user story. The classification is:

- **Must have:** the functionality must be implemented in the solution; otherwise it will fail or cannot be considered a success.
- **Should have:** it should be implemented, as it is high-priority functionality. The solution won't fail without it, but there should be justified reasons for leaving it out.
- **Could have:** it is desirable and would be convenient to have in the solution, but it will depend on the project's timeline and budget.
- **Won't have (for now):** this is very low-priority or discarded functionality that may become relevant in the future. In that case, it would move to one of the other states.

It is important to distinguish between priority and value. A user story may have no value for the customer or user yet still be essential and high priority. One example is the infrastructure needed to implement a piece of software, which provides no value to the customer in itself but without which the solution cannot be developed or run.

Another tool that helps the Product Owner prioritize the Product Backlog appropriately is calculating the Return on Investment (ROI).

ROI = Business value / Size

- **Business value:** the relative value of the item to the business or the customer.

- **Size:** the estimate, in story points, of the effort needed based on complexity and size.

Applied in descending order, ROI is an excellent guide for prioritizing the backlog: the higher the return, the higher the item's priority.

WSJF = Cost of delay / Size

A highly recommended and comprehensive lean prioritization technique is **WSJF (Weighted Shortest Job First)**, created by Don Reinertsen and described in 2009 in his book *The Principles of Product Development Flow: Second Generation Lean Product Development*. It is based on the economics of product development flow and is calculated by dividing the cost of delay by the duration.

What is interesting about cost of delay is that even technical stories have it, despite having no intrinsic business value. Cost of delay represents the value we fail to obtain if we do not build a given story. In the case of a technical story, we may miss out on the business value of other user stories tied to it.

Since duration is impossible to estimate, we calculate the cost of delay using the size of the item:

Cost of delay = Business value + Time criticality + Risk reduction and opportunity value

- Business value.
- **Time criticality:** this measure addresses the need to deliver the item within a timeframe and is associated with the progressive decline in value. The greater the need to deliver, the higher this factor.
- **Risk reduction and opportunity value:** this measure assigns a relative value to eliminating one or more risks, or a value for the potential business opportunities the item provides.

The WSJF prioritization technique consists of applying the following process to the Product Backlog:

- Features, user stories, and epics are estimated relative to one another.
- The unit scale is the Fibonacci series: 1, 2, 3, 5, 8, 13, 21.
- The list is filled in column by column.
- A 1 is placed on the item with the smallest value. There must be a 1 in each column.
- The rest of the items in the column are filled in with a relative estimate against the 1.

- Once the last column is done, the WSJF is calculated and sorted from highest to lowest.
- The highest-priority item is the one with the highest WSJF.

Feature / User story / Epic	Business value	Time criticality	Risk reduction and opportunity value	Size	WSJF
As a member of the administration department, I want a screen to manage the company's product catalog and link to the corresponding Google+ photo.	1	5	1	1	**7**
As a salesperson, I want a catalog with an online ordering system so that I can train customers and place orders for those who need them.	5	13	8	8	**3.25**
As a member of the administration department, I want a screen to manage customer data.	8	2	3	5	**2.6**
As a manager, I want monthly information on products sold and invoices issued so that I have reliable, on-demand reports.	3	1	1	3	**1.67**

Acceptance criteria

After 50 years of software engineering history, we have concluded that acceptance criteria—which are sometimes turned into tests—are an excellent language for detailing functional requirements, and that is why they take on such importance in user stories.

To measure the quality of an acceptance criterion, the SMART method is used, by which they should, as far as possible, be:

- **Specific**.
- **Measurable**.
- **Achievable**.
- **Relevant**.
- **Time-boxed**.

Acceptance criteria are usually expressed as a checklist or as a flow once user stories become ready to implement, and they are refined during Sprint Planning. They help the development team understand how the product works, so they can better estimate the size of the underlying story.

During development, they also serve as a guide for decision-making. During the Sprint Review, the Product Owner will use these acceptance criteria and the Definition of Done (DoD) to check whether each user story can be accepted and considered finished.

Acceptance criteria can be written in natural language, as the Product Owner expresses them, and can take different formats. For example:

> ***Check*** *[criteria].*
> ***Demonstrate*** *[expected behavior].*
> ***Verify that when*** *[role]* ***does*** *[action],* ***they get*** *[result / expected behavior].*
> ***Given*** *[context] and* ***additionally*** *[context],* ***when*** *[event],* ***then*** *[result / expected behavior].*

An excellent option is to write them using the scenario-based behavior technique from BDD (Behavior-Driven Development), with Gherkin, a language created for describing software behavior. The Gherkin syntax is as follows:

> ***Scenario*** *[scenario number] [scenario title]:*
> ***Given*** *[context] and* ***additionally*** *[context],*
> ***When*** *[event],*
> ***Then*** *[result / expected behavior].*

The elements of the acceptance criteria derived from this syntax are:

- **Scenario number:** an identifier for the scenario associated with the story.
- **Scenario title:** describes the scenario that defines a behavior.

- **Context:** details the conditions that trigger the scenario.
- **Event:** represents the action the user performs, in the context defined for the scenario.
- **Result / expected behavior:** the system's behavior in that situation.

```
Scenario: Some determinable business situation
  Given some precondition
  And some other precondition
  When some action by the actor
  And some other action
  And yet another action
  Then some testable outcome is achieved
  And something else we can check happens too
```

Figure 9: Gherkin example.

Example of acceptance criteria written in Gherkin:

User story

As a customer,
I want to withdraw money from the ATM
so that I can avoid queuing at the bank.

Acceptance criteria

Scenario 1: account is in credit.
Given the account is in credit
And the card is valid
And the ATM has cash available
When the customer requests money
Then the amount is charged to the account
And the money is dispensed to the customer
And the customer's card is returned.

Scenario 2: the account exceeds the overdraft limit agreed with the bank.
Given the account exceeds the overdraft limit agreed with the bank
And the card is valid
When the customer requests money
Then the ATM displays a message denying the request
And no money is dispensed to the customer
And the customer's card is returned.

Non-functional requirements

Non-functional requirements (NFRs) represent general qualities and constraints that affect entire applications and systems, unlike epics and user stories, whose acceptance criteria focus on specific behaviors and functions. However, a story or epic cannot be considered done if it does not meet the associated non-functional requirements. These guarantee that the system meets highly important qualities such as usability, security, and others such as:

Accessibility	Dependability	Availability	Interoperability	Operability
Extensibility	Traceability	Performance	Recoverability	Maintainability
Compatibility	Conformance	Scalability	Usability	Configurability
Reliability	Certification	Efficiency	Security	Robustness

Implementing new NFRs usually requires new items in the Product Backlog in the form of technical stories. Once included in a sprint, these evolve into new constraints for the system.

NFRs are also usually reflected in the acceptance criteria of user stories. And if they are persistent constraints, they are added as new items in the Definition of Done (DoD).

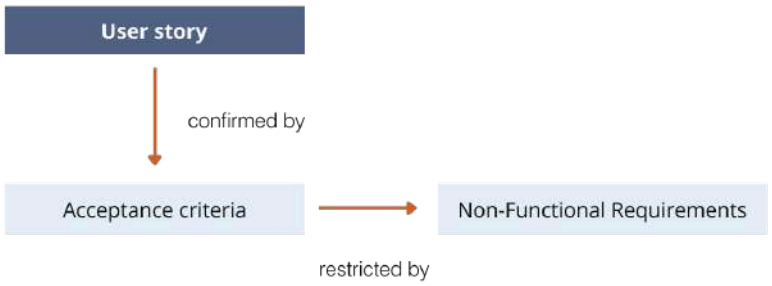


Figure 10: Non-functional requirements as constraints.

Like any requirement, NFRs must be described and quantified to be understood better. Let's look at an example following this pattern:

Step 1
Name: [in the form quality.subquality].
Scale: [what to measure (unit)].
Meter: [how to measure (method)].

Step 2
Target: [level of success to reach].
Constraint: [level of failure to avoid].
Baseline: [current level].

We start from the following user story, related to an online bookstore:

As a customer,
I want to browse books
so that I can choose which one to buy.

For which we have set response time as a non-functional requirement, since if the page takes more than 3 seconds to load, the buyer will look for other options:

Step 1
Name: usability.performance.
Scale: milliseconds between the buyer clicking "search" and being shown the results page.
Meter: average of book-search times.

Step 2
Target: <200 milliseconds (what Google considers the maximum response time).
Constraint: >3000 milliseconds (when the buyer starts to feel anxious).
Baseline: 1800 milliseconds.

Definition of Done

The Definition of Done, or DoD, is an explicit quality agreement that answers the question "what does a user story have to meet to be done?". It is a tool related to software quality that must be agreed jointly by the team and the Product Owner. It consists of a set of criteria and actions that add verifiable, demonstrable value to the product.

The Definition of Done is a document that explains what is needed to consider a story done, and it lets everyone involved share this definition. It should be distinguished from acceptance criteria, which are specific to each story.

Without a Definition of Done, a cloud of unfinished work can build up that complicates planning, causes delays, and creates hidden risks for the release. As we will see next, the definition applies at the level of task, user story, sprint, and release. At the sprint level, it usually includes that the user story meets its acceptance criteria.

Task:

- Implemented.
- The written code follows styles and guidelines.
- Unit tests done.
- Integrated into the repository.
- Integrates with the rest in the build.
- Task manager updated.
- Certain metrics met: coverage, static analysis, etc.
- Approved by the tester.

Sprint:

- All planned stories are done.
- Sprint Review done and feedback obtained from stakeholders.
- Retrospective done and an improvement action identified.
- Approved by the Product Owner.

User story:

- All associated tasks are done.
- Issues found during development resolved.
- Code documented, with relevant comments where they add clarity.
- Project/product documentation and other requirements done.
- Meets the acceptance criteria.
- Integration tests done.
- Accumulated unit tests and acceptance tests passed, manually or automatically.
- Follows engineering/architecture standards.
- Meets the non-functional requirements.

- Installed in a test, staging, or pre-production environment and has passed smoke tests.
- Approved by the Product Owner.

Release:

- All planned stories are done.
- Distribution media produced.
- User documentation checked.
- Meets the non-functional requirements and standards.
- Release installation/deployment instructions done.
- Security, deployment, and regression tests done.
- Critical issues and those needing resolution for deployment resolved.
- Release communication and user training done.
- Approved by the Product Owner.

INVEST and quality user stories

The INVEST method was formulated by Bill Wake in his article "INVEST in Good Stories, and SMART Tasks," published in 2003, and later popularized by Mike Cohn in his book User Stories Applied (2004). The method is used to check the quality of a user story by verifying that it meets a set of characteristics:

- **Independent**.
- **Negotiable**.
- **Valuable**.
- **Estimable**.
- **Small**.
- **Testable**.

Independent: being able to plan and implement user stories in any order has many advantages and makes the team's later work easier. For this to be possible, each story should be independent. One way to reduce dependencies is to combine stories or split them differently.

Negotiable: a user story is a short description of a need, without details. Stories should be negotiable, since the specifics will be agreed with the customer or user later on. The story should be open enough to allow conversation and negotiation (without imposing the "how"), but not ambiguous: it must clearly express who, what they need, and why.

Valuable: a user story must provide value to the customer or user. One way to achieve this is to have them write it.

Estimable: a good user story must be estimable with enough precision for the Product Owner to prioritize and plan its implementation. The team usually does the estimating, and the estimate will be affected by the size of the story—the larger it is, the greater the uncertainty—and by the team's knowledge of the need expressed. Where that knowledge is lacking, more rounds of conversation will be needed.

Small: user stories should span at most a few weeks of work per person. Some teams even restrict them to days per person. A short description helps reduce the size of a user story, making it easier to estimate.

Testable: the user story should be testable in the confirmation phase. If the customer or user doesn't know how, it means the functionality is not entirely clear or not valuable; and if the team cannot test it, there is no way to know whether it is finished.

Thomas Wallet has devised a set of questions that help evaluate the INVEST criteria:

Independent

- Are the internal dependencies resolved?
- Are the dependencies visualized?
- Are the external dependencies agreed?
- Are the external dependencies resolved?

Negotiable

- Is the user need to be solved understood?
- Is the description open to future refinements?
- Can the team define the "how"?
- Is the "how" formulated by the team alone?

Valuable

- Have the key users been identified?
- Is the business value defined?
- Is the right group involved in defining the business value?
- Does the business value allow for negotiation?

Estimable

- Has the item been clearly defined and recorded?
- Is the team ready to estimate?
- Was it estimated on a relative reference scale?
- Did everyone take part in the estimate?

Small

- Does it have fewer than <N> acceptance criteria?
- Is the estimated size less than <N>?
- Does it no longer make sense to subdivide further?
- Are there no more risks of size growth?

Testable

- Are the acceptance criteria defined and clear?
- Are the acceptance criteria agreed with the users?
- Has the necessary documentation been produced?
- Has the provision of critical test resources been agreed?
- Has the provision of critical test resources been resolved?

Some companies have designed their own cards, like these from Braintrust. Besides giving a professional look, they are complete in information and consistent in the space allotted to each field:

Figure 11: Example of cards designed by Braintrust.

Best-practice tips:

- Always write stories in terms of the what; avoid the how.
- Don't write an exhaustive description—just enough.
- Write the acceptance criteria explicitly enough.
- Estimate every story. Failing to do so can create false expectations.
- Don't entrust all the information to the cards. External documentation, such as a wiki, can be used.
- Never consider a story finished before it meets all the requirements.

Splitting user stories

The refinement meeting is part of the continuous flow of requirements gathering through user stories. It is a meeting to keep the Product Backlog up to date, in which the Product Owner, with the team's help, adds, re-prioritizes, removes, and splits backlog items. Its goal is to ensure that user stories likely to be developed soon have enough detail and estimation for the team to be able to commit to them.

Experience shows that smaller user stories improve flow, and that large user stories entail greater functional uncertainty and are harder to estimate.

Smaller user stories are simpler, more understandable, and clearer to develop. This results in fewer issues per story, so the team's work can focus on developing new stories rather than resolving setbacks.

When unexpected problems or blockers arise, the impact on delivery at the end of the sprint is directly proportional to the size of the stories. Small stories have small scopes, so if they cannot be finished, the scope of what is not delivered will be small.

Suppose we have the following story:

> *As a marketing content creator,*
> *I want to manage the news on the website*
> *so that I can announce our latest updates to the customer.*

The story could be split into the following sub-features:

- Create a news item.
- Edit a news item.
- Delete a news item.
- Publish a news item.
- Unpublish a news item.

Leaving out the original story does not have the same impact as leaving out one or two of its sub-features.

Moreover, having small user stories gives us a competitive advantage in the market. The example story spans five possible smaller stories. Do we need to implement all the sub-features to bring the news functionality to market? No. Creating news items and being able to publish and unpublish them (in case we make a mistake) will probably be enough. This way, we leave room in the sprint to release other requirements that provide value to the user.

Splitting stories improves understanding, makes estimates more accurate and prioritization easier, reduces the number of issues, gives greater stability in delivery times, and increases productivity and market competitiveness.

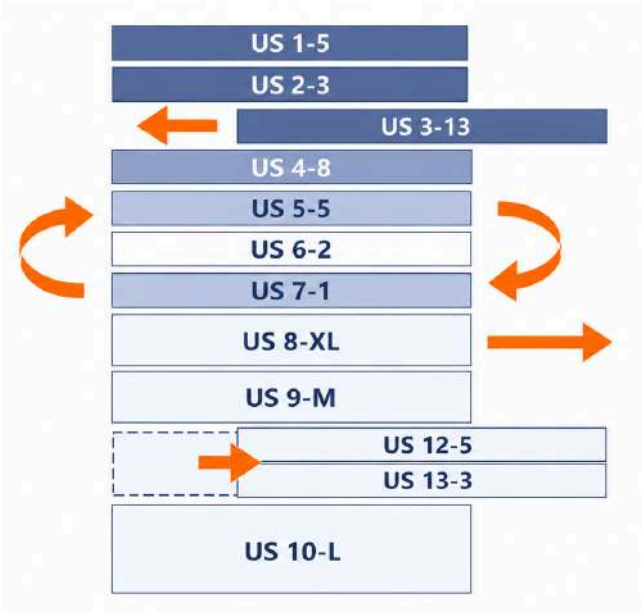


Figure 12: Product Backlog refinement.

User stories can be split horizontally and vertically.

Splitting horizontally means doing so by type of work—for example, by technology. This is typical of traditional management methodologies. It produces stories that have no business value individually, only together. This makes them impossible to prioritize and also encourages each team member to focus only on stories in their specialty, which tends to create bottlenecks. Agile requirements engineering avoids these problems through the cross-functionality of its members: everyone takes part, to a greater or lesser extent, in the different types of task.

Vertical splitting is more useful. A vertically split story produces stories that each have business value. It is divided not along technical layers but along functional ones. Like the increments resulting from a sprint, these stories are like a slice of cake that includes all the necessary technical layers.

The 10 strategies by Verwijs

In 2009, Richard Lawrence documented recurring patterns he observed while working with multiple development teams. His work *Patterns for Splitting User Stories* identifies specific techniques applicable to different contexts.

In 2015, Christiaan Verwijs popularized them and presented them as **10 strategies for splitting user stories vertically**, to obtain smaller, functional stories:

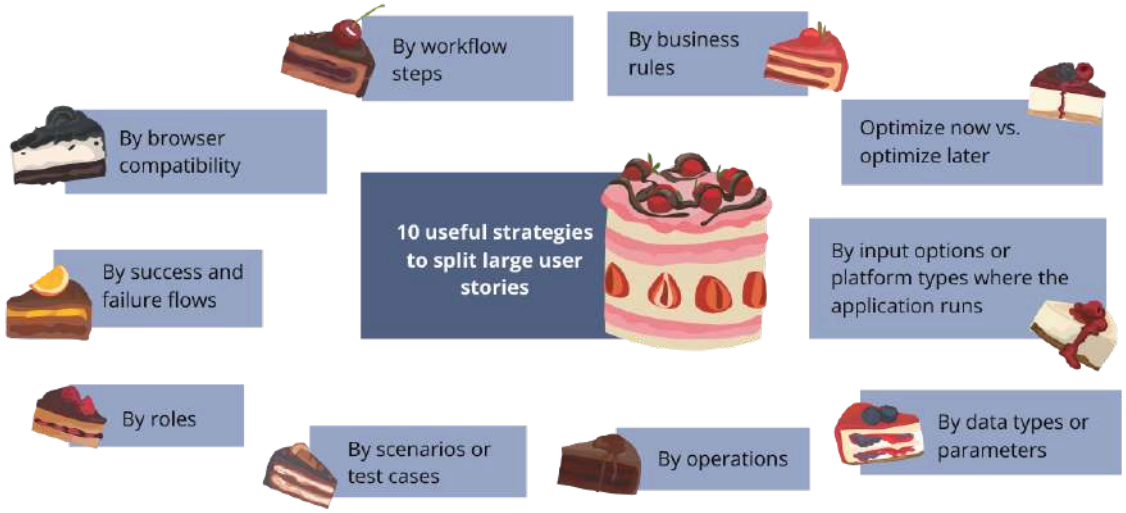


Figure 13: Diagram of the 10 user story splitting strategies.

To the 10 strategies we must add one more, for cases where implementing the story is complex and hard to understand and requires a prior exploratory activity (a spike).

- **Strategy 0:** extract a spike.
- **Strategy 1:** splitting by workflow steps.
- **Strategy 2**: splitting by business rules.
- **Strategy 3:** splitting by happy/unhappy flow.
- **Strategy 4:** splitting by input options/platforms.
- **Strategy 5:** splitting by data types or parameters.
- **Strategy 6:** splitting by operations.
- **Strategy 7:** splitting by test cases/scenarios.
- **Strategy 8:** splitting by roles.
- **Strategy 9:** splitting by optimize now or later.
- **Strategy 10**: splitting by browser compatibility.

In all these strategies, explained below, splitting reduces uncertainty, lets us focus on the most important stories, and leaves the rest for future development.

Strategy 0: build a spike

A spike is essentially a development activity, in the form of a technical story, that is added to the Product Backlog. It may involve gathering information for a later decision or for designing a solution.

When to apply this strategy?

Building a spike serves to acquire the prior knowledge needed to better understand the solution or the need associated with a user story, reducing uncertainty about its implementation. This technique takes shape as proofs of concept or prototypes that make it possible to assess the story's feasibility. The results of these exploratory activities allow sound decisions to be made to refine or define the scope of a piece of functionality.

Spikes are written in the form of a technical story. Suppose the following user story:

> *As a cyclist,*
> *I want to pay for my purchase by credit card*
> *so that I can buy accessories for my bike.*

The corresponding spike could be:

> *Explore our bank's credit card payment gateway*
> *to determine the feasibility of including it in our product.*

The acceptance criteria refer to the questions that need to be answered. Besides generating knowledge and clearing up doubts, spikes make it easier for teams to give solid estimates for later user stories.

Strategy 1: splitting by workflow steps

When refining a user story with an associated workflow, we often find that the workflow makes it more complex than expected. When this happens, this strategy identifies the MVP (minimum viable product) and the incremental stories needed to build the original story.

Sometimes the greatest value lies in the first and last steps of the flow; the intermediate ones add value but can be postponed. The MVP would consist of those first and last steps, splitting the rest into steps or groups of steps that can be added independently later.

In other user stories, the entire workflow is necessary, and the MVP must be a thin slice through the whole flow. The splitting strategy would then be based on adding complexity through stories that add layers to the workflow.

We will also find stories with several alternatives or possible paths. We can approach the MVP by starting with a single path—the highest-value one—and base the splitting strategy on the different possible paths.

Splitting by workflow brings the great benefit of improving understanding of the functionality, which in turn makes it easier to define the MVP.

When to apply this strategy?

When the user story involves a workflow of some kind. Let's look at an example with this story from an online store:

> *As a shopper,*
> *I want to pay for the items in my shopping cart*
> *so that I can receive them at home.*

Splitting by workflow steps:

> ***MPV***
> *As a shopper, I want to confirm and review my shopping cart so that I can confirm before paying.*
> *As a shopper, I want to use PayPal so that I can automatically provide my personal details and shipping address and make the payment.*
>
> ***Increment 1***
> *As a shopper, I want to log in with my Facebook account so that I can automatically enter my details and shipping address.*
> *As a shopper, I want to use a credit card so that I can make the payment.*
>
> ***Increment 2***
> *As a shopper, I want to use a bank transfer so that I can make the payment.*

Strategy 2: splitting by business rules

This strategy proposes incorporating special business rules gradually, as incremental extensions of the initial functionality. These rules may be industry standards, legal regulations, or any kind of complex constraint.

This way, the Product Owner can obtain initial user stories that cover the essential rules, implementing the others later or in a simplified form. A simplified rule such as a message in an online store saying "we don't ship orders outside the EU" may be enough to start with.

When to apply this strategy?

When the user story involves a set of business rules, implicit or explicit, and some can be incorporated gradually later on.

> *As a shopper,*
> *I want to pay for the items in my shopping cart*
> *so that I can receive them at home.*

Splitting it produces other, equally complex stories using different business rules:

> *As the store owner,*
> *I want to reject orders under 10 euros*
> *so that I can avoid unprofitable purchases.*
>
> *As the store owner,*
> *I want to reject customers outside the EU with orders under 100 euros*
> *so that I can avoid shipping costs that make the purchase unprofitable.*
>
> *As the store owner,*
> *I want to reserve stock of ordered products for 48 hours*
> *so that I can offer customers accurate stock levels.*
>
> *As the store owner,*
> *I want to automatically cancel orders not paid within 48 hours*
> *so that I can sell those products to other customers.*

Business rules are often implicit, so to discover them it can be useful to first split by test cases/scenarios.

Strategy 3: splitting by happy/unhappy flow

Good user stories and their acceptance criteria involve success flows (happy flows) and failure flows (unhappy flows). The success (or happy) flow describes the ideal scenario, with few or no options—how the functionality behaves when everything goes well. Failure (or unhappy) flows deal with any alternative flow: deviations, exceptions, or problems that may arise.

Identifying the flows inherent to a piece of functionality lets us understand it in depth and makes prioritization easier for the Product Owner. For example, blocking a

user after three failed attempts (an unhappy flow) may not be important while we have few users.

When to apply this strategy?

When the functionality involves an ideal flow in which everything goes right and one or more alternative flows.

Example of a bank login user story:

> *As a user,*
> *I want to access my online banking*
> *so that I can securely access my bank transactions.*

Splitting by the various possible flows:

> ***Happy flow***
> *As a user,*
> *I want to log in with my email account*
> *so that I can access my online banking.*
>
> ***Unhappy flow***
> *As a user,*
> *I want to reset my password when my login fails*
> *so that I can try to log in again.*
>
> *As a user,*
> *I want the option to register a new email account if I can't log in with the current one,*
> *so that I can access my online banking.*
>
> *As the bank's security officer,*
> *I want to block users who log in with invalid credentials three times in a row*
> *so that I can protect the system from attackers.*

Strategy 4: splitting by input options/platforms

This strategy proposes splitting user stories by the interfaces or means through which a user can interact with the system—for example, different devices such as desktop computers, laptops, and mobile phones, and their operating systems, which may require customizations.

This split makes it easier for the Product Owner to prioritize the most important input options or platforms at any given time.

Sometimes the user interface is more complex than the functionality itself. In that case, the splitting strategy involves a user story with the simplest possible interface, which is later enriched with incremental stories.

When to apply this strategy?

When the story must support several input methods and/or platforms, and we can ask ourselves: is there a simpler version we could build first?

Example of a user story for an online travel agency:

> *As a traveler,*
> *I want to buy flights between two destinations*
> *so that I can enjoy my holidays seeing the world.*

Stories resulting from this strategy:

> ***By device***
> *As a traveler,*
> *I want to buy flights between two destinations through a mobile app*
> *so that I can enjoy my holidays seeing the world.*
>
> *As a traveler,*
> *I want to buy flights between two destinations through a computer*
> *so that I can enjoy my holidays seeing the world.*
>
> ***By complexity***
>
> *As a traveler,*
> *I want to choose flights by typing in the date I'm interested in*
> *so that I can find the ticket.*
>
> *As a traveler,*
> *I want to choose flights by selecting the date through an elegant calendar-style user interface*
> *so that I can find the ticket.*

For now the traveler has to print from the browser; later we'll add:

> *As a traveler,*
> *I want to print the purchased flight*
> *so that I can board.*

Strategy 5: splitting by data types or parameters

This strategy splits user stories into smaller items based on the complexity of their data or parameters. It proposes incorporating them incrementally through smaller stories, which also helps us understand them better.

Imagine a book-search feature that we split into stories by search type; this lets us estimate and prioritize the functionality more precisely. Perhaps the relevant searches are by title and ISBN, if September is approaching and it's textbook-buying season; other search types can be implemented in a simplified way for now or left for later.

You can also split by the types of data the functionality returns. Even if the idea is to present them visually with charts, for now we can decide to show them in tabular format and create the charts by hand in Excel.

Another option is to use ERDs (entity-relationship diagrams), which suggest several different groups of data to work from. If an ERD has entities such as books, categories, and authors, we can expect related user stories.

When to apply this strategy?

When the story can be split by the types of data it returns or by the parameters it is supposed to handle.

Example of a user story for a restaurant booking website:

> *As a diner,*
> *I want to find a restaurant*
> *so that I can take my partner out for a romantic dinner.*

The split produces as many stories as there are valuable search criteria:

> *As a diner,*
> *I want to search for restaurants by cuisine type*
> *so that I can quickly find a restaurant that suits my wishes.*
>
> *As a diner,*
> *I want to search for restaurants by price range*
> *so that I can get more relevant search results.*
>
> *As a diner,*
> *I want to search for restaurants by a group of main ingredients*
> *so that I can get more relevant search results.*

Strategy 6: splitting by operations

This strategy offers a natural way to split the operations of a user story into independent implementations.

We are referring to user stories about entities such as items, orders, rates, users, and so on, all associated with create, read, update, and delete operations (CRUD). The Product Owner can bring the team the highest-priority or most complex features, such as creating an item, leaving less frequent ones for later, such as deleting or updating an item, which can be done directly on the database in the meantime.

When to apply this strategy?

When a user story involves operations. The word "manage" is often a sign that it covers multiple operations, such as the typical CRUD ones or others, for example configuration.

Let's look at a user story for a shopping cart:

> *As an online book shopper,*
> *I want to manage the books I'm choosing in a shopping cart*
> *so that I can complete the purchase with my final decision.*

Splitting by individual CRUD operations:

> *As an online book shopper,*
> *I want to add new books to my shopping cart*
> *so that I can see the books I've chosen and, once my budget is used up, buy those books.*
>
> *As an online book shopper,*
> *I want to view my shopping cart*
> *so that I can see what I've chosen so far.*
>
> *As an online book shopper,*
> *I want to update the quantity of each book*
> *so that I can buy several copies of the same book.*
>
> *As an online book shopper,*
> *I want to remove books from my shopping cart*
> *so that I can buy the books that matter most to me and keep the purchase within my budget.*

From a business-value perspective, adding books to the shopping cart is the highest-priority feature.

Strategy 7: splitting by test cases/scenarios

This split is based on all the test cases and scenarios derived from the story's acceptance criteria. It is especially useful when it is hard to split the story based on its functionality alone. Relevant test scenarios can easily be translated into user stories and implemented incrementally.

Test scenarios in automated environments are usually written in Gherkin, a language created for the purpose so that tests can be understood by development teams and technical staff as well as by the Product Owner, business stakeholders, analysts, and even the customer.

If a test scenario is not very common or does not pose a high enough risk, the Product Owner may decide to omit the functionality for now and focus on others that provide more value. They may also decide to simplify a test scenario to cover the most urgent problems.

When to apply this strategy?

When we can answer the question: which scenarios need to be checked to know that the functionality is correctly implemented?

We'll split this bank user story:

> *As a customer,*
> *I want to withdraw money from the ATM*
> *so that I can avoid queuing at the bank.*
>
> ***Scenario 1: account is in credit***
> *Given the account is in credit*
> *And the card is valid*
> *And the ATM has cash available*
> *When the customer requests money*
> *Then the account is debited*
> *And the money is dispensed to the customer*
> *And the customer's card is returned.*
>
> ***Scenario 2: the account exceeds the overdraft limit agreed with the bank***
> *Given the account exceeds the overdraft limit agreed with the bank*
> *And the card is valid*
> *When the customer requests money*
> *Then the ATM displays a message denying the request*
> *And no money is dispensed to the customer*
> *And the customer's card is returned.*
>
> ***Scenario 3: the customer has started the process on their phone***
> *Given the customer has entered the amount to withdraw in the mobile app*
> *When they place the phone with the QR code in front of the ATM scanner*

Then the ATM dispenses the money directly to the customer.

Splitting this story by test scenarios produces new stories:

Scenario 1
As the bank manager,
I want to check that the customer's account is in credit
so that I don't dispense money by mistake.

As the bank manager,
I want to check that the customer's card is valid
so that I don't dispense money by mistake.

As the bank manager,
I want to check that the ATM has cash available
so that I can ensure the money is dispensed.

Scenario 2
As the bank manager,
I want the ATM to deny the money via a message
so that I can prevent the account from exceeding the agreed overdraft limit.

Scenario 3
As a customer,
I want to start the ATM cash-withdrawal process in the app
so that I can avoid queuing and get the money directly.

As we can see, this strategy implicitly helps apply others. Scenarios 1 and 2 lead us to use the splitting-by-business-rules strategy, while scenario 3 lends itself to splitting by input options/platforms.

Strategy 8: splitting by roles

One of the first considerations when writing a user story is the role we are writing it for. Roles are usually defined through user personas, and the story is likely to benefit each one in different ways. Splitting by roles can be applied when more than one receiving role benefits.

Imagine a restaurant booking website aimed at two audiences or roles: regular customers and VIP customers. The booking process is likely to differ. With this strategy, a Product Owner finds a way to split, prioritize, and deliver value sooner.

When to apply this strategy?

When the story involves several roles or groups that carry out certain parts of the functionality.

Example of a user story for an online newspaper:

> *As the news department,*
> *I want to publish new articles on our website*
> *so that we can attract our customers and give them a reason to come back.*

Splitting by the different roles involved:

> *As a customer,*
> *I want to read new articles*
> *so that I can stay informed about important events.*
>
> *As a journalist,*
> *I want to write new articles*
> *so that I can attract our customers.*
>
> *As a designer, I want to apply the corporate styles to new articles so that I can integrate them into our website's look and feel.*
> *As an editor, I want to review new articles before publishing them on our website so that I can prevent errors.*
> *As the audiovisual manager, I want to include photos and videos in new articles so that I can capture the scenes of the story.*

Strategy 9: splitting by optimize now or later

In some stories, the complexity lies not in their functional implementation but in non-functional requirements such as performance. This splitting strategy aims to implement the functionality at various degrees of refinement and optimization, which we can summarize as "make it work" and, later, "make it fast."

A lot can be learned from a slow initial solution, and it may even have some value for the user. The Product Owner can prioritize the resulting stories based on what most customers need first.

It is important to stress that this strategy refers to functional optimization, not code optimization that could serve as an excuse to incur technical debt.

When to apply this strategy?

When the functionality can be implemented at different levels of refinement and optimization: make it work first, then improve it to meet the non-functional requirements.

Example of a hotel-search story:

As a visitor,
I want to search for hotels in a neighborhood
so that I can narrow my search to the area where it is located.

Splitting by the different degrees of optimization:

As a visitor,
I want to search for hotels within a radius from a base address
so that I can narrow my searches.

As a visitor,
I want to enter the postal code and search the neighborhood
so that I don't have to fill in the base address.

As a visitor,
I want to use my GPS location and search the neighborhood
so that I don't have to fill in the base address.

As a visitor,
I want to get the most searched-for hotels in the neighborhood immediately while other hotels load in the background
so that I can get results faster.

Strategy 10: splitting by browser compatibility

This strategy proposes splitting user stories by the internet browsers that may interact with the page. The most modern browsers, such as Chrome, Firefox, and Edge, keep up with the standards, while older ones usually need customizations to make everything work.

This strategy makes prioritization easier for the Product Owner and helps the team focus on the most valuable browser at any given time. A high percentage of an online store's users may access it from Chrome, whereas the users of a large multinational's legacy-heavy intranet may access it through Internet Explorer. By focusing on the most-used browser, we can deliver value sooner.

When to apply this strategy?

When we have stories for web applications that must work in different browsers.

Example of an online beauty-products store:

As a shopper,
I want to see the product details
so that I can decide whether it's what I want to buy.

Splitting by the different browsers in which the functionality must be viewable:

As a shopper,
I want to see the product details in Edge
so that I can decide whether it's what I want to buy.

As a shopper with a vintage computer,
I want to see the product details in Internet Explorer 7
so that I can decide whether it's what I want to buy.

As a shopper with an iPhone,
I want to see the product details in Safari
so that I can decide whether it's what I want to buy.

SPIDR model

Mike Cohn proposes that roughly 80% of user stories can be split using just five techniques, grouped under the acronym SPIDR:

- **S – Spike** (extract research).
- **P – Path** (split by alternative paths).
- **I – Interface** (split by interface complexity or platforms).
- **D – Data** (split by data types or parameters).
- **R – Rules** (split by business rules).

S - SPIKE

A spike is a time-boxed research activity that generates the knowledge needed to better understand a story or reduce technical or functional uncertainty. It may include prototypes, proofs of concept, or feasibility research.

When to apply this technique?

When the story contains significant technical or functional uncertainty that prevents estimating it confidently or implementing it effectively. Spikes are especially useful when:

- We don't know whether a technology or integration is viable.
- We need to evaluate different technical approaches.
- The technical complexity is unknown.
- We need user research or concept validation.

Example

Original story
As a cyclist,
I want to pay for my purchase by credit card
so that I can buy accessories for my bike without using cash.

Splitting by extracting a spike
Spike (technical story): Investigate the integration with our bank's payment gateway to determine whether it meets our requirements for security, response times, and transaction costs.

Spike acceptance criteria (questions to answer):
- *Does the gateway support the card types we need?*
- *Are response times under 3 seconds?*
- *Is it PCI-DSS compliant?*
- *What is the cost per transaction?*

Resulting story (after completing the spike):
As a cyclist,

I want to pay by credit card using the [name] gateway
so that I can complete my purchase safely and quickly.

P - PATH

If a user can achieve a goal in multiple ways (for example, paying by card, PayPal, and/or bank transfer; sharing on Facebook, Twitter, and/or email), each alternative path can become a separate story.

When to apply this technique?

When the story involves:

- Multiple different workflows.
- Different ways to achieve the same goal.
- Several input or interaction methods.
- Happy-path and unhappy-path routes.
- Different CRUD operations (Create, Read, Update, Delete).
- Multiple roles that perform the same functionality differently.

Example

Original story
As a platform user,
I want to share a video
so that I can recommend it to my contacts.

Splitting by paths

As a platform user,
I want to copy the video link to the clipboard
so that I can share it through any medium I choose (WhatsApp, email, etc.).

As a platform user,
I want to share the video directly on Facebook
so that I can quickly reach my friends on that social network.

As a platform user,
I want to share the video directly on Twitter
so that I can spread it among my followers immediately.

As a platform user,
I want to email the video from the platform itself
so that I can share it with contacts who don't use social networks.

I - INTERFACE

This technique consists of first implementing a simple version of the interface and then adding complexity, or supporting different platforms incrementally.

When to apply this technique?

When the story:

- Must work on multiple devices or platforms (web, mobile, tablet).
- Has an interface that can be implemented in versions of increasing complexity.
- Requires support for different browsers.
- Includes advanced UI features that are not essential initially.

Example

Original story
As a shopper,
I want to search for products by category
so that I can quickly find what I need.

Splitting by complexity (simple → complex):

As a shopper,
I want to search for products by typing the category into a text field
so that I can find products quickly and directly.

As a shopper,
I want to select categories from a dropdown menu with icons
so that I can explore the options more intuitively.

As a shopper,
I want to browse an interactive catalog with images for each category
so that I can discover products in a more engaging, exploratory way.

D - DATA

This technique consists of implementing the functionality for a subset of data, categories, or parameters first, and then gradually expanding to other data types.

When to apply this technique?

When the story handles:

- Multiple data types or categories of information.
- Different search or filter parameters.
- Several input formats.
- Different data sources.

- Different attributes or properties of an entity.

Example

Original story
As a diner,
I want to search for restaurants
so that I can find options that suit my preferences and situation.

Splitting by search data types:
As a diner,
I want to search for restaurants by cuisine type (Italian, Japanese, Mexican, and so on)
so that I can find the kind of food I feel like today.

As a diner,
I want to search for restaurants by price range (€, €€, €€€)
so that I can stay within my available budget.

As a diner,
I want to search for restaurants by distance from my location
so that I can find nearby, convenient options.

As a diner,
I want to search for restaurants by user rating (stars)
so that I can choose places well rated by other diners.

As a diner,
I want to search for restaurants by table availability
so that I can book immediately without waiting.

R - RULES

Business rules are constraints, conditions, or special logic that determine how the system should behave in different situations. They include legal regulations, company policies, complex calculations, or special conditions.

When to apply?

When the story involves:

- Multiple business rules that can be implemented incrementally.
- Legal or regulatory constraints.
- Complex conditional logic.
- Special cases or exceptions.
- Different policies depending on customer type, region, and so on.

The recommended approach is to first implement a simple version of the functionality with basic or even simplified rules (for example, a manual message), and then add more complex rules progressively.

Example

Original story
As a shopper,
I want to complete my order
so that I can receive the products I've selected at home.

Splitting by business rules
As the store owner,
I want to set a minimum order of €10
so that I can keep the business profitable by avoiding very small orders.

As the store owner,
I want to apply a €5 shipping charge for orders between €10 and €50
so that I can cover the logistics costs of small shipments.

As the store owner,
I want to offer free shipping for orders over €50
so that I can encourage larger purchases.

As the store owner,
I want to apply a €15 surcharge for shipments outside the EU
so that I can cover the additional costs of international shipping.

As the store owner,
I want to automatically reserve stock for 48 hours after the order
so that I can ensure availability while the customer completes payment.

As the store owner,
I want to automatically cancel orders not paid within 48 hours
so that I can release the reserved stock and offer it to other customers.

Practical application of the SPIDR model

When you face a large story during Product Backlog refinement, ask yourself these questions in order:

1. Is there significant technical or functional uncertainty? → Consider extracting a **SPIKE.**
2. Can the user achieve the goal in multiple ways? → Split by **PATH** (alternative paths).
3. Does the interface have multiple levels of complexity or platforms? → Split by **INTERFACE.**

4. Does the story handle different data types or parameters? → Split by **DATA.**
5. Does it involve multiple business rules or constraints? → Split by **RULES.**

The same story can be split using multiple SPIDR techniques in succession. For example, you first split by PATH (CRUD operations), and then split each operation by DATA (product types). Splitting is an iterative process.

Other ways of gathering requirements

User stories are not the only technique for capturing software requirements. Throughout the history of software engineering, various approaches have been developed, each with its own strengths and contexts of application. Two of the most common techniques are use cases and traditional functional requirements.

Use cases

Use cases are another technique that is sometimes incorporated into agile processes, but they are not equivalent to user stories. It is worth distinguishing between the two to avoid confusion.

What are use cases?

A use case captures the desired functionality from the perspective of the users (actors) and how they interact with the system. They are written using UML (Unified Modeling Language) in use case diagrams. UML is a modeling language for describing a system simply, from both its static and dynamic perspectives.

Unlike user stories, which are written in colloquial language to recall the conversation with the customer, UML is not meant to sound natural. Use cases include formal elements such as:

- **Actors:** the roles that interact with the system.
- **Preconditions:** the state of the system before the use case runs.
- **Main flow:** the sequence of steps in the successful scenario.
- **Alternative flows:** variations on the main flow.
- **Exception flows:** what happens when something fails.
- **Postconditions:** the state of the system after the use case runs.

Differences from user stories

As Alistair Cockburn notes in his 2001 book *Agile Software Development*, user stories are closer to requirements capture—the phase used to draw out the user's needs—whereas use cases focus on requirements specification. We could say that a user story states **what the customer or user wants**, and a use case gets into how they want it.

Aspect	Use cases	User stories
Language	UML (technical, formal)	User's natural language
Focus	Requirements specification (the how)	Requirements capture (the what)
Level of detail	High (flows, extensions, preconditions)	Low (placeholder for conversation)
Documentation	UML diagrams + extensive descriptions	Card + simple acceptance criteria
When produced	Extensive upfront analysis	Just-in-time, shortly before development
Acceptance criteria	Tracking matrices with percentages	Binary (meets or doesn't)
Maintenance	Requires updating UML diagrams	Minimal (discarded after implementation)
Main author	Functional analyst or architect	Product Owner + team + user
Flexibility to change	Requires formal reworking	High, changes easily incorporated

They also differ in their acceptance criteria. Use cases require requirements-tracking matrices with percentages to mark progress; a user story's acceptance criterion does not use percentages but is binary—it either passes or it doesn't.

User stories are living documents. The functional and technical analysis is done shortly before development—in the sprint kickoff meeting in the case of Scrum—and the breakdown into tasks is done by the team. The level of detail and foresight far exceeds what a single architect or functional analyst can achieve with use cases. When a project adopts agile methods, it is generally advisable to prioritize user stories over traditional use cases, given their focus on conversation and iterative delivery. However, certain regulated contexts or complex systems may benefit from a documented hybrid approach.

When can use cases still be useful?

Although user stories are generally more effective in agile contexts because of their simplicity and focus on conversation, use cases can still add value in specific situations:

- **Systems with complex**, interdependent logic where it is crucial to formally document all possible flows before starting development.
- **Highly regulated projects** (healthcare, aeronautics, finance) that require formal documentation for audits or certifications.
- **Highly distributed teams** where synchronous communication is difficult and more thorough documentation is needed.
- **Integration projects with legacy systems,** where mapping existing use cases helps to understand the current system.
- **Fixed-scope contracts** that require detailed specification before development.

In pure agile contexts, where requirements are volatile, the team is cross-functional, and collaboration is continuous, user stories tend to be more appropriate because of their lightness and adaptability.

Functional requirements

User stories are also sometimes confused with functional requirements. The former are understood to be an artifact of agile methods, and the latter to play the same role in traditional methodologies. However, there are significant differences.

What are traditional functional requirements?

A traditional functional requirement is a formal, detailed statement of a capability the system must provide. They usually follow standards such as IEEE 830 (now IEEE 29148) and are documented in software requirements specifications (SRS).

A typical functional requirement has this structure:

ID: REQ-FUN-001
Title: user authentication
Priority: high
Description: the system shall allow users to authenticate with a username and password. The system shall validate the credentials against the user database. If the credentials are correct, the system shall create a session valid for 24 hours. If the credentials are incorrect, the system shall display the message "Incorrect username or password" and lock the account after 5 consecutive failed attempts.
Preconditions: the user must be registered in the system.
Postconditions: the user has an active session.
Dependencies: REQ-FUN-015 (Session management).

Key differences from user stories

We must be aware that, although user stories describe functionality that will be useful to the customer or user, and are usually written on cards or sticky notes, they are much more than that. They imply a later conversation in which the team, together with the user or customer, works out the details of the functionality to be developed.

As with functional requirements, user stories state the what but not the how of developing the functionality. Stories should not have the level of detail that a functional requirement's specification does.

Aspect	Functional requirements	User stories
Format	Detailed formal specification	Simple sentence in natural language
Initial level of detail	Very high (complete behavior)	Low (only the essentials)
Production	Extensive upfront analysis (requirements phase)	Continuous conversation (progressive refinement)
Author	Requirements analyst	Product Owner with input from the end user

Audience	Technical (developers, testers, architects)	Mixed (business and technical)
Documentation	Extensive formal document (SRS)	Simple physical or digital cards
Traceability	Unique IDs, traceability matrices	Prioritized backlog
Change management	Formal change request process	Changes welcome at any time
Success criteria	Compliance with the specification	Conversation → acceptance criteria
Philosophy	"Comprehensive documentation"	"Conversation over documentation"

Transition and coexistence strategies

Adopting agile approaches does not require abandoning earlier techniques entirely; the advisable thing is **to select and combine practices** according to context, keeping traceability or formal documentation when it adds value or is mandatory.

Layered hybrid approach

- High-level requirements or epics documented formally (for regulatory or contractual compliance).
- User stories for the iterative implementation of those high-level requirements.
- Traceability between traditional documents and user stories.

Gradual transition

1. Start with user stories in low-risk pilot projects.
2. Keep formal requirements in critical or regulated systems.
3. Progressively increase the use of stories as the team matures.

Just-enough documentation

- Keep formal documentation where it is legally required or adds real value.
- Use user stories where flexibility and speed matter more.
- Generate formal documentation automatically from stories where possible (traceability tools).

When to use each approach

The decision about which technique to use should be based on a pragmatic assessment of the context:

Project context	Most appropriate technique
Iterative digital product, startup	User stories
Medical system with FDA certification	Formal requirements + use cases
Mobile app with continuous user feedback	User stories
Aeronautical control system	Formal requirements + use cases
Constantly evolving web platform	User stories
Integration with multiple legacy systems	Use cases for mapping + stories for new features
Project with a fixed-scope contract	Formal requirements (with the option of stories internally)
Internal development of corporate tools	User stories (more agile and collaborative)

There is no single correct approach

The debate between use cases, functional requirements, and user stories should not be framed in terms of "right vs. wrong" or "modern vs. obsolete." Each technique has its place and value depending on the context.

User stories excel in environments where adaptability, continuous collaboration, and incremental value delivery are priorities. Traditional use cases and functional requirements remain valuable in contexts where formal documentation, thorough traceability, and detailed upfront specification are necessary or required by regulation.

Professional maturity consists of knowing all the available tools well and knowing when to apply each one. As the *Agile Manifesto* states, we value "customer collaboration over contract negotiation" and "responding to change over following a plan"—but that does not mean contracts and plans have no value; simply that we value the former more.

Technical stories

The Product Backlog contains all the items to be built: user stories, epics, and themes. But to be able to plan both sprints and releases, it must also contain all the pending work, including technical needs. Technical stories usually provide indirect value (sustainability, risk reduction, performance, security), even though they are not always value that is visible to the end user.

To give some examples, technical stories include actions such as setting up a web server, implementing a set of tables in a database that will be consumed by several features, and elements of security, scalability, and performance. Other types of technical stories focus on resolving technical debt and on refactoring; others are exploratory stories, such as a technical or functional analysis that serves to clear up uncertainty about a user story.

Characteristics of technical stories

Writing and format

Technical stories are written directly in clear, **precise technical text**, without a pattern like the one used for user stories.

Technical acceptance criteria

They also have associated acceptance criteria that are checked in the Sprint Review, but **these are verified by the relevant technical audience** (the team, the architect, other developers), not by the Product Owner or end users. The criteria are usually technical and measurable:

- "The migration completes without data loss (verified with checksums)."
- "Query response time is reduced from 2s to <200ms."
- "Test coverage stays above 80%."
- "The solution is backward-compatible with previous versions of the API."

Ownership and responsibility

The Product Owner is responsible for the Product Backlog, including the technical stories. However, given the specialized nature of these stories, the technical team plays a leading role in defining, estimating, and relatively prioritizing them.

It is advisable for the Product Owner to trust the team's technical judgment to understand the why and when of these stories, while retaining ultimate responsibility for prioritizing them in the context of the whole backlog.

Importance of visibility

It is valuable for the business and the Product Owner to have contact with the technical side, to forge a common language and to make everyone involved aware of its importance. Without this visibility, technical stories are often perceived as a "waste of time" or "unnecessary perfectionism" by non-technical stakeholders, which creates friction and accumulated technical debt.

Types of technical stories

Different types of technical stories can be identified:

Architecture

They build elements such as APIs, which create the structure, operation, and interaction between different parts of the software. These stories define fundamental architectural decisions that affect multiple features.

Example

> *Implement a secure login system.*

Product infrastructure

Stories that are consumed directly by user stories. This could include new and/or modified infrastructure, and refactoring opportunities arising from some functional need.

Example

> *Set up the database and web servers.*

Team infrastructure

Stories that support the team's ability to deliver software. They are usually stories for tools and testing frameworks, metrics, design, planning, and so on. They may also involve the team developing, or buying and installing, something.

Example

> *Set up a continuous integration system.*

Refactoring

Stories that represent refactoring candidates, such as technical debt. But it is not only code that needs refactoring; it can also include designs, automation, tools, and any process documentation.

Example

> *Standardize the code of the loan-calculation function.*

Spikes

Time-boxed exploratory stories that answer a question, or gather information for a later decision or for designing a solution.

Example

> *Evaluate Oracle versus SQL Server.*

Technical spike: if we are not sure how to develop something from a technical point of view, we create this type of spike—a short activity focused on finding a development approach and on determining the feasibility and impact of design strategies.

Functional spike: it helps teams discover the details of features and designs by creating prototypes, and reach an exact understanding of what the customer needs.

Technical debt

The term "technical debt" was coined by Ward Cunningham in 1992 with the following metaphor: accumulating technical debt is like taking out a financial loan. You gain speed now (you can launch sooner), but you pay interest later in the form of greater difficulty, slowness, and cost to make changes.

Types of technical debt

- **Deliberate debt:** in exceptional, critical situations, the team may consciously decide to take very specific, bounded shortcuts to meet a strategic business window of opportunity. This practice should be exceptional, not routine. If a team incurs deliberate debt frequently, it points to planning or estimation problems that need to be addressed.
- **Accidental debt:** arises from a lack of knowledge or understanding of the problem, and is discovered when the code makes future changes difficult.
- **Debt through negligence:** ignoring known good practices due to pressure or lack of discipline (not writing tests, unjustified duplicated code).

Managing technical debt

As in finance, technical debt accrues "interest": decreasing speed, growing bugs, lower team morale, and a higher cost of change. The interest is paid every sprint, not only when we decide to tackle the debt.

Technical debt can be acceptable when it allows hypotheses to be validated quickly, critical windows of opportunity to be met, or the domain to be learned. What matters is that it be conscious, documented, and backed by a clear repayment plan.

Management strategies

- Refactor progressively whenever legacy code is touched.
- Reserve a fixed percentage of sprint capacity for technical stories.
- Prioritize according to impact on future speed and risk.

Prioritizing technical stories

One of the biggest challenges in agile teams is balancing functional stories (direct value to the user) and technical stories (system sustainability). As a general reference, mature teams on established products usually devote 20–30% of sprint capacity to technical stories. However, this ratio is highly contextual. For example, in new projects with a solid architecture it may be 10–15%; in products with high accumulated technical debt it may temporarily be 40–50%.

Prioritization techniques

- **Technical cost of delay:** how much does it cost us NOT to do this technical story? Debt in critical modules that constantly generate bugs is high priority; "nice to have" improvements are low priority.
- **Impact-effort matrix:** classify by high/low impact and high/low effort. Prioritize high-impact, low-effort stories first (quick wins), and plan the high-impact, high-effort ones (strategic projects).
- **Fixed reserved time:** set a team policy such as "25% of every sprint is devoted to technical stories," giving predictability to the business and sustainability to the system.
- **Warning signs of insufficient technical investment:** decreasing velocity sprint after sprint, growing bugs, increasing development time, a frustrated team, and fear of changing fragile code.

User Story Mapping

Jeff Patton describes this technique in his book *User Story Mapping: Discover the Whole Story, Build the Right Product* (2014). It is very useful for building a Product Backlog that goes beyond a one-dimensional list of user stories and epics.

It is a collaborative technique for building the Product Backlog. It involves everyone connected with defining, using, and building the product: stakeholders, customers or end users, the Product Owner, and the team. There is also a facilitator, who in most cases is the Scrum Master or the agile coach. The goal is to jointly define, discover, prioritize, and estimate the user stories and epics envisioned as part of the product to be built.

Figure 14: A flat Product Backlog and a Product Backlog built with the User Story Mapping technique

This technique allows us to obtain:

- **Shared vision:** everyone involved takes part in creating the user story map, building a shared vision of the product to be delivered. This makes it easier to keep the focus on the solution throughout its construction.
- **Alignment:** the team and the business (the Product Owner and stakeholders or customers) can be aligned on what will be built, knowing the why and the underlying reasons.
- **A better understanding of customers' wants and needs:** through modeling the customers and what they will do with the product.
- **A better understanding of the problems customers face:** taking part in this exercise—modeling what the customer will do with the product, and how—makes it easier to empathize, reflect, and propose better solutions.
- **A user story map ordered by version:** by modeling a backbone of the customer's flow through the product, we can define the set of stories that will make up the minimum viable product to bring to market. A minimum that lets users carry out

the whole flow in a basic way (at least one feature for each mandatory activity in the flow). It also helps define an initial idea of the upcoming versions envisioned, which may change depending on the results obtained once the MVP and each new version go into production.

Elements of User Story Mapping

To use User Story Mapping, the following elements must be defined:

1. **The backbone of the user story map:** the backbone, or spine, of the user story map captures the high-level activities a user will perform when using the product. For example, the backbone for buying a digital book on a website would be:

Figure 15: Example of a user story mapping backbone.

2. **User stories associated with each activity in the process, ordered by value:** the user stories that make each activity possible are defined. They are then ordered from top to bottom, placing the most valuable or highest-priority ones higher up. Continuing with the example, a set of user stories associated with each activity and prioritized by value would be:

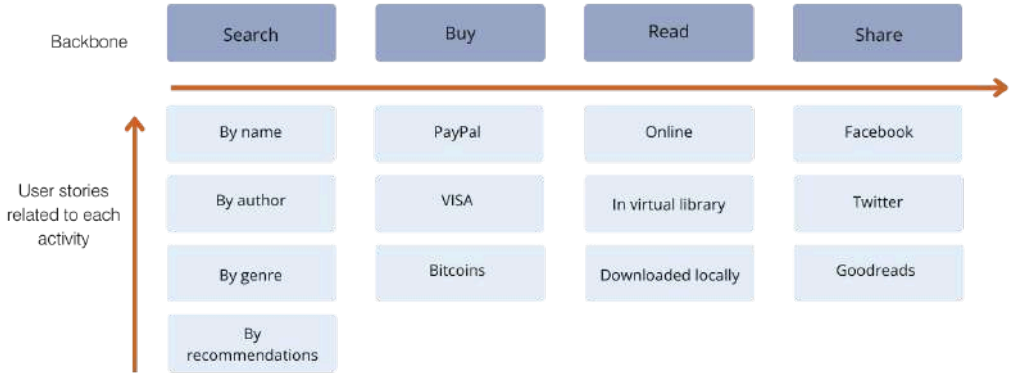


Figure 16: Backbone and user stories of a user story mapping.

3. **Minimum viable product (v1.0) and upcoming planned versions:** the user story map shows which user stories will make up the MVP (v1.0) and the following versions.

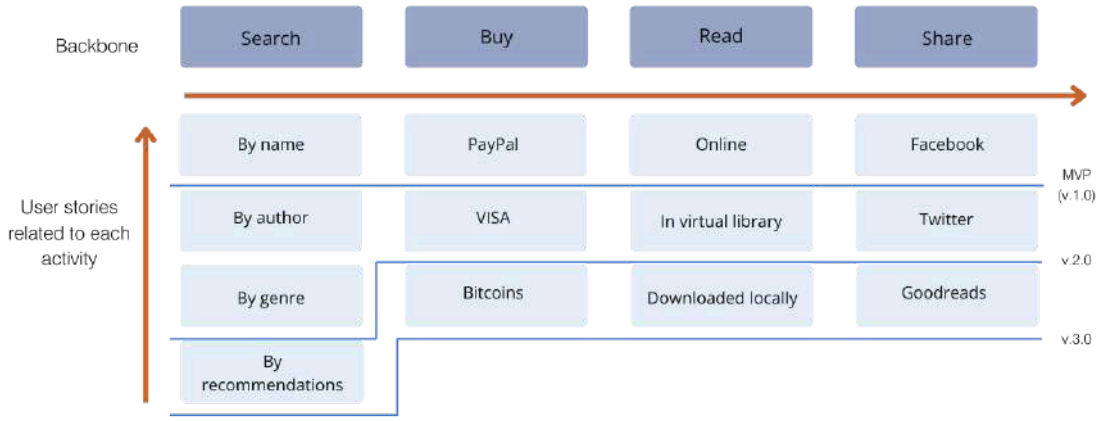


Figure 17: Backbone, user stories, and versions of a user story mapping.

Once we reach this point, the team can make an initial estimate of the effort needed to build each user story.

This way, the Product Owner can find out the approximate effort to develop the MVP and each of the versions defined in the user story map. If they know the team's velocity—that is, how many story points the team can complete per sprint—and the length of the sprints, they can produce a product graph and obtain an initial estimate of when each version is likely to be delivered.

This initial estimate is adjusted as working software is delivered, providing feedback about the product and its features.

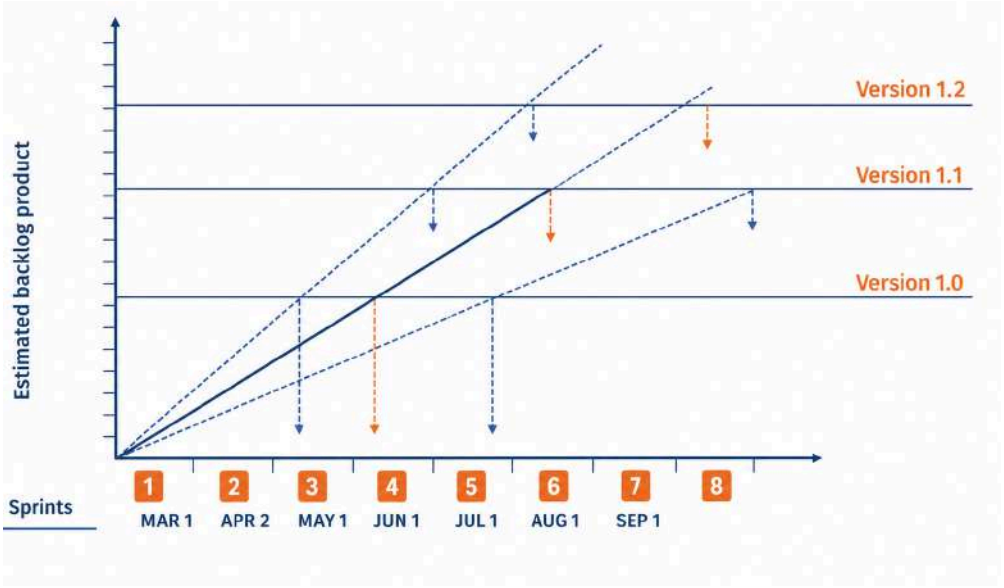


Figure 18: Version release forecast on a product graph.

Related techniques

User Story Mapping is not the only technique for visualizing and prioritizing work in agile environments. Other complementary or alternative techniques include:

Impact Mapping

Developed by Gojko Adzic, Impact Mapping helps connect business goals with product deliverables. It is structured in four levels:

- **Why?** – The business goal.
- **Who?** – The actors who can produce the desired impact.
- **How?** – Impacts on the behavior of those actors.
- **What?** – The deliverables (features) that can produce those impacts.

Impact Mapping is especially useful when you need to ensure that every feature in the Product Backlog is justified by a clear business goal, avoiding building features "just because." It complements story mapping well: Impact Mapping helps decide WHAT to build, and story mapping helps organize HOW to build it sequentially.

Opportunity Solution Trees

Popularized by Teresa Torres, Opportunity Solution Trees help product teams connect desired outcomes with specific solutions through identified opportunities:

- The desired outcome (the root of the tree).
- Opportunities discovered through user research.
- Possible solutions for each opportunity (which may be experiments, features, and so on).

This technique is valuable when you work in continuous product discovery and want to make sure you are solving real user problems, not just implementing features. Like Impact Mapping, it can be used before or in combination with story mapping.

Artificial intelligence and user stories

The arrival of generative artificial intelligence tools (ChatGPT, Claude, Gemini, and others) has raised new possibilities and risks in managing user stories. These technologies promise to speed up processes, generate content, and analyze large volumes of information in seconds. However, it is essential to set a clear stance on their role in agile development.

Artificial intelligence is a support tool, not a substitute for human collaboration or a member of the team. User stories exist to facilitate conversations between people. AI can optimize certain repetitive processes and provide useful suggestions, but it must never displace genuine dialogue with real users, nor the team's strategic creativity and critical judgment.

User stories are based on Ron Jeffries's three Cs formula. Conversation is the central element: the moment when the team, the Product Owner, and the users build a shared understanding. An AI can generate cards and suggest criteria, but it cannot replace the human conversation that brings stories to life.

Appropriate uses of AI

Artificial intelligence can add value in specific phases of working with user stories, as long as it is used as a complement to—never a replacement for—the team's collaborative work.

It is also essential to consider the information you enter into AI tools. It is best to avoid entering sensitive data: personal data, trade secrets, incidents containing customer data, credentials, or contractual details. Whenever possible, use approved corporate environments and anonymize or minimize the data before analyzing it.

Generating initial drafts

One of the most obvious applications of AI is quickly generating drafts of user stories from short descriptions or identified needs.

Tools (examples)

There are general-purpose models, assistants integrated into management suites, and extensions specific to the backlog. The specific availability changes quickly; what matters is validating outputs and governing their use.

Practical application

Imagine that during a discovery session with users we identify the following need: "Customers want to be able to recover their password when they forget it." We can use AI to quickly generate variations of user stories:

Prompt to the AI

Based on this need: "Customers want to recover their password," generate 3 user story variations following the format "As a [role], I want [goal] so that [benefit]."

Possible AI response

Variation 1:

As a registered customer,
I want to be able to request a password reset via my email
so that I can recover access to my account if I forget it.

Variation 2:

As a platform user,
I want to receive a temporary password-recovery link
so that I can create a new password securely.

Variation 3:

As a customer,
I want to verify my identity through security questions
so that I can reset my password without relying on email.

These variations can serve as a **starting point for the conversation with the team and the Product Owner**. However, they require **mandatory human validation.** Stories created solely by AI should not be implemented without this validation step.

Refining and improving existing stories

AI can act as a quality assistant, helping to identify problems in stories already written and suggesting improvements.

AI can help to:

- identify ambiguities in the wording that could lead to misunderstandings,
- suggest additional acceptance criteria based on the story's description,
- detect inconsistencies between related stories in the backlog,
- propose ways to split large stories by applying strategies,
- check compliance with INVEST (the user story quality model).

Practical example

As a user, **I want** to manage products.

This story is too large, ambiguous, and does not specify the benefit. We can ask the AI for help:

Prompt

This user story seems too large and ambiguous: "As a user, I want to manage products." Suggest how to split it following the CRUD pattern (create, read, update, delete) and add specific benefits.

Possible AI response

Story 1:

As the catalog administrator,
I want to be able to create new products
so that I can expand the range available to customers.

Story 2:

As the catalog administrator,
I want to view the complete list of products
so that I can check which items are active in the system.

Story 3:

As the catalog administrator,
I want to update data for existing products
so that I can keep the information up to date (prices, descriptions, stock).

Story 4:

As the catalog administrator,
I want to remove discontinued products
so that I can keep the catalog clean and avoid confusion.

The team can then review these suggestions, adjust the role ("catalog administrator" instead of a generic "user"), prioritize which are most valuable to the business, and refine the benefits according to their organization's specific context.

Analyzing and organizing the backlog

When a backlog grows considerably, AI can assist with analysis and pattern-detection tasks that would be laborious by hand.

AI can:

- identify duplicate or very similar stories that could be consolidated,
- detect gaps in functional coverage ("we have stories for creating and viewing, but not for updating"),

- analyze implicit dependencies between stories (e.g., one story requires another to be implemented first),
- suggest logical groupings by theme, module, or user flow,
- propose an initial prioritization based on business-value patterns.

Example

In a backlog with 150 stories, we can ask the AI:

> *Analyze this backlog and group the stories by functional module: authentication, product catalog, shopping cart, payments, and administration.*

The AI can process the full list and provide an initial categorization that the team then reviews and adjusts. This saves time in Product Backlog refinement sessions.

Extracting information from multiple sources

One of AI's capabilities is natural language processing (NLP) applied to large volumes of unstructured text.

AI can process:

- **User interview transcripts:** identify patterns, recurring needs, and pain points mentioned.
- **Customer support tickets:** analyze frequent complaints and requests that can become stories.
- **Survey or NPS feedback:** extract qualitative insights about which features users value.
- **Product reviews on marketplaces:** understand which aspects of the product are most appreciated or criticized.
- **Scattered technical documentation:** consolidate information about legacy systems or necessary integrations.

Example

After conducting 20 user interviews, we have 20 transcript documents. We can use AI to:

> *Analyze these 20 user interview transcripts and extract the 10 needs or problems mentioned most frequently. For each one, suggest a preliminary user story.*

This analysis provides a starting point based on real data, which the Product Owner then validates and prioritizes according to the business strategy.

Critical risks and limitations of AI

Despite the apparent advantages, using AI in user stories entails serious risks that can compromise product quality and the effectiveness of the agile process. It is essential that teams are aware of these limitations to avoid costly mistakes.

AI does not understand real users

Artificial intelligence generates text based on statistical patterns learned from vast corpora of data during its training. It has no direct experience with your product's users, does not understand their specific context, and cannot genuinely empathize with their frustrations and needs.

When AI generates a user story, it is producing text that sounds good because it has seen thousands of similar examples in its training, but it has no real understanding of the problem it is trying to solve or of the value it would bring to your specific user.

You can end up with a backlog full of stories that sound professional, follow the "As a... I want... so that..." format perfectly, and comply with INVEST, but that do not solve your users' real problems. This leads to building features that no one uses or values.

AI cannot replace "getting out of the building" and talking to real users. AI can help process and organize information, but **never generate it from scratch without real human input.**

"Perfect" stories that inhibit conversation

AI-generated user stories can seem too polished and complete, creating a false sense that there is nothing left to discuss or refine. This contradicts one of the fundamental principles we saw at the beginning: user stories are placeholders for conversations, not final, closed documents.

When a story reaches the team with impeccable wording, detailed acceptance criteria, and a perfect structure (all AI-generated), the team can fall into the trap of assuming it is already "done" and simply proceed to implement it without the necessary dialogue.

Let's recall the three Cs:

- **Card:** a brief reminder.
- **Conversation:** the central element where understanding is built.
- **Confirmation:** validation that it was understood correctly.

If AI generates a "card" so complete that it appears to need no "conversation," we are perverting the model.

Loss of organizational context

AI operates without knowledge of your organization's particularities, which severely limits its usefulness for strategic decisions.

AI does not know:

- your company's specific strategy and its short/medium-term goals,
- the technical limitations of your legacy system or current architecture,
- the tacit knowledge your team has accumulated through previous iterations,
- the regulatory constraints specific to your industry (GDPR, PCI-DSS, sector regulations),
- the organizational culture and internal dynamics that affect the feasibility of certain solutions,
- the history of past decisions and why certain paths were taken.

Example

AI might suggest: "As a user, I want social media integration to share my profile." This sounds reasonable and is a common feature. However, your team knows that:

- your company's privacy strategy prohibits integrations with third-party social networks,
- it was tried in the past and caused support problems,
- your target users (the corporate B2B sector) do not value this feature.

This AI-generated story is technically correct but strategically wrong for your context.

Stories that are technically "correct" but strategically irrelevant

AI can generate stories that follow all the good practices (INVEST, correct format, acceptance criteria) but that do not provide real business value or that prioritize low-importance features.

Example

You ask the AI to "generate 20 user stories for a mobile banking app." The AI will generate stories that sound logical based on banking apps it knows from its training. It may include things like:

- "As a user, I want to be able to change the app's color theme."
- "As a user, I want to see animations when making transactions."
- "As a user, I want to be able to give my accounts nicknames."

These stories are valid from a technical point of view, but are they a priority for your bank right now? Perhaps your strategy is focused on attracting young customers who value investing in cryptocurrencies, or on improving the experience of older people who struggle with the app's complexity.

AI-generated stories do not reflect these strategic priorities unless you specify them in great detail in the prompt (and even then, the AI will interpret them according to its training biases).

As a result, the team may waste valuable time implementing features that, although correct, do not move the needle on business goals.

Overconfidence in AI outputs

An important psychological risk is the overconfidence that AI responses can create. Because the responses are usually well written and presented with confidence, it is easy to assume they are correct or complete.

The following cognitive biases are likely to be triggered:

- **Authority bias:** we tend to trust sources that appear "authoritative."
- **Automation bias:** we over-trust automated systems, assuming they are more accurate than human judgment.

The reality is that generative AIs can produce incorrect information with complete confidence (the phenomenon known as "hallucinations"). In the context of user stories, this can translate into:

- suggesting integrations with APIs that do not exist,
- proposing technically unfeasible features,
- inventing "best practices" that are not backed by the industry,
- creating dependencies between stories that make no sense.

It is critical to maintain a healthy skepticism and always validate AI outputs with the team's expert knowledge.

Atrophy of professional skills

Excessive use of AI as the "first line" for generating stories can atrophy the team's skills. Agile professionals should think critically about the problem first, and only then use artificial intelligence as a refinement tool.

The recommended approach is for the team to first write a draft without using artificial intelligence. They can then turn to it to review, suggest improvements, and identify gaps; the final decision about what to incorporate must also rest with the team.

Appendices

User stories in other fields

Agility is being adopted in new areas of the company, and one of the most fitting is marketing. Micro-campaigns with fast feedback in the form of sprints make Scrum an ideal way of working. Jim Ewel, an expert in Agile Marketing, compares marketing user stories with IT ones, focusing on their differences:

IT user stories	Marketing user stories
Low-level	High-level
A large number of them in the backlog	Relatively few in the backlog
Focused on features	Focused on results

IT user stories are detailed and very fine-grained, whereas marketing ones are broader and their Product Backlog tends to be short.

But the most interesting difference is what they focus on: IT stories, when talking about features, are often pinned down early into capabilities and verifiable criteria; marketing stories focus on outcomes (impact on behavior/perception). For example, when people remember certain ads, they tend to talk about how they made them feel, how they changed their day, or how they made them laugh, rather than about features.

The solution a marketing story points to speaks to what customers want to achieve—whether solving a problem, easing a pain, or aspiring to something better through our product. The "so that" part of the story must answer this central marketing question: what's in it for me? (WIIFM).

> ***As*** *[my ideal customer],*
> ***I want*** *[our solution]*
> ***so that*** *[my problems disappear].*

Digital tools

Many companies prefer to use digital tools instead of physical boards. The ecosystem of tools for managing user stories has grown significantly, offering options for different needs and work contexts.

Main tool categories

Atlassian suite

- **JIRA:** a robust, highly configurable tool, the market leader for agile project management.
- **Trello:** a visual, Kanban-board-based interface, simpler and more intuitive.
- **Confluence:** complements JIRA/Trello with documentation and wikis.

Modern specialized tools

- **Linear:** designed specifically for software development teams, with a fast, minimalist interface.
- **ClickUp:** an all-in-one platform with high customizability.
- **Notion:** combines project management with collaborative documentation.
- **Monday.com:** focused on visualization and workflow automation.

Specialized tools

- **Azure DevOps:** full integration with the Microsoft ecosystem.
- **Rally (Broadcom):** for large organizations with multiple teams.
- **Taiga:** an open-source platform for agile management.

Visual collaboration tools

- **Miro:** a collaborative digital whiteboard, ideal for story mapping and remote workshops.
- **Mural:** similar to Miro, focused on visual facilitation.
- **Excalidraw:** an open-source collaborative digital whiteboard.
- **FigJam:** from Figma, integrated with product design.

Artificial intelligence features

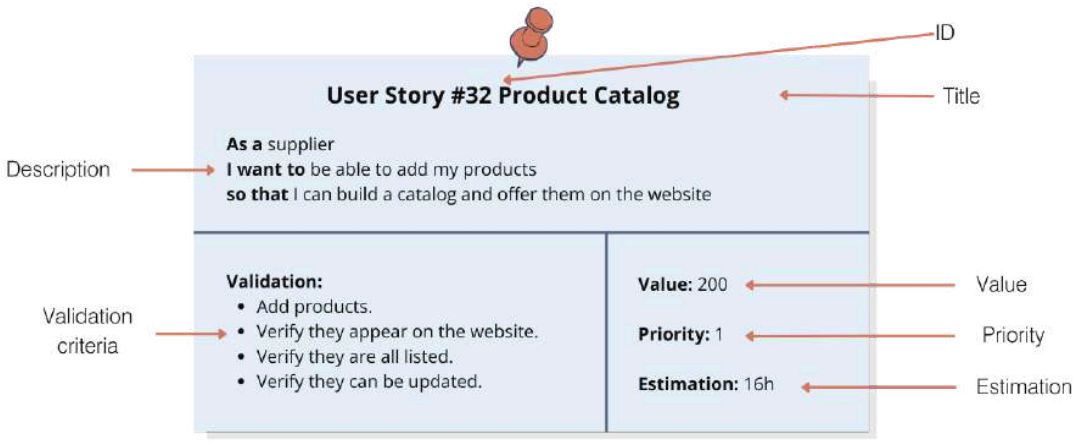
Many of these tools are incorporating artificial intelligence capabilities to assist with writing, refining, and managing user stories. For example:

- **Atlassian Intelligence (Jira):** can suggest acceptance criteria, split large stories automatically, generate progress summaries, and automate answers to questions about the project.
- **Linear AI:** assists with writing descriptions and suggests labels and priorities.
- **ClickUp AI:** generates story drafts and summarizes comment threads.
- **Notion AI:** helps with supplementary documentation and summaries.

These features should be used as support tools, not as a replacement for human judgment and collaboration. AI can speed up mechanical tasks (initial drafting, formatting, suggestions), but conversation and collaborative refinement remain essential.

Examples of use

Let's look at our example of a user story card:



Now we'll design the same card in JIRA, a tool that belongs to the Atlassian suite. We have all the information available, just as on the model card: identifier, description, estimate, acceptance criteria, and so on.

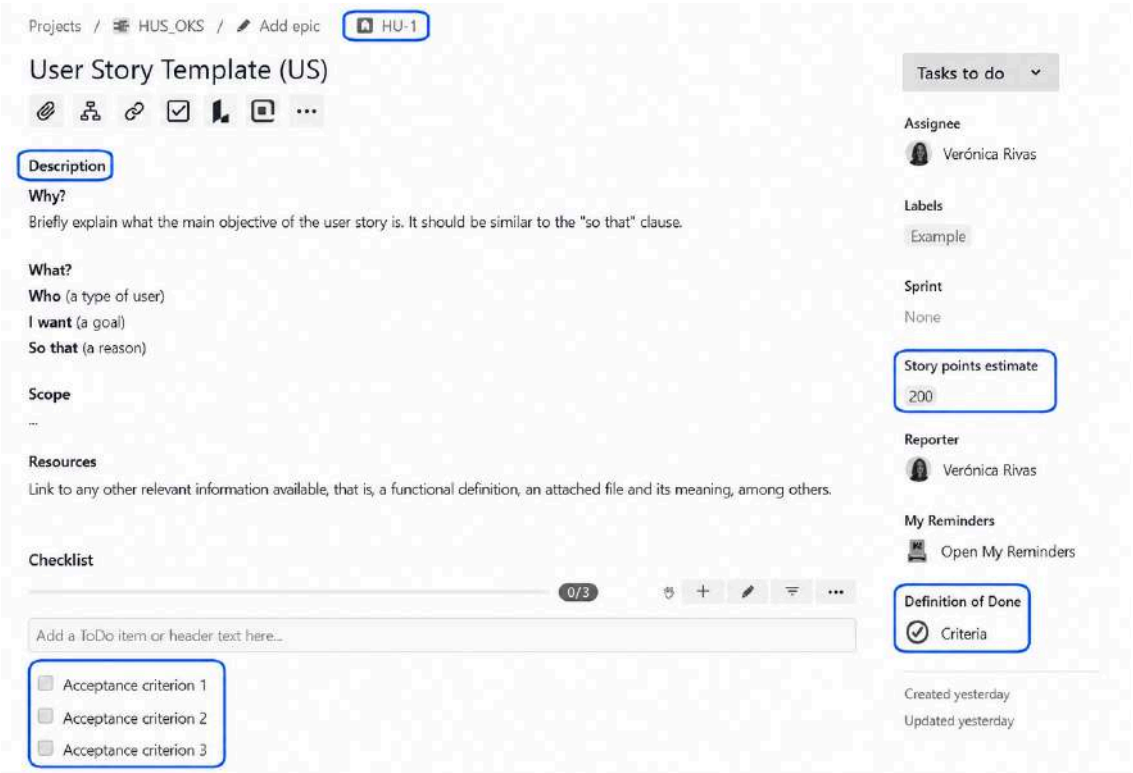


Figure 19: Example of a user story in JIRA.

The wonderful thing about digital tools is that they let us expand or enrich the traditional card. We can include images, documentation, and other data so that they are available at a glance:

- The teammate developing the user story.
- The sprint it is in.
- The status of the user story, which will depend on those defined in the tool.
- Who created the user story.

Let's look at another example with a different digital tool, in this case Trello (also from the Atlassian suite):

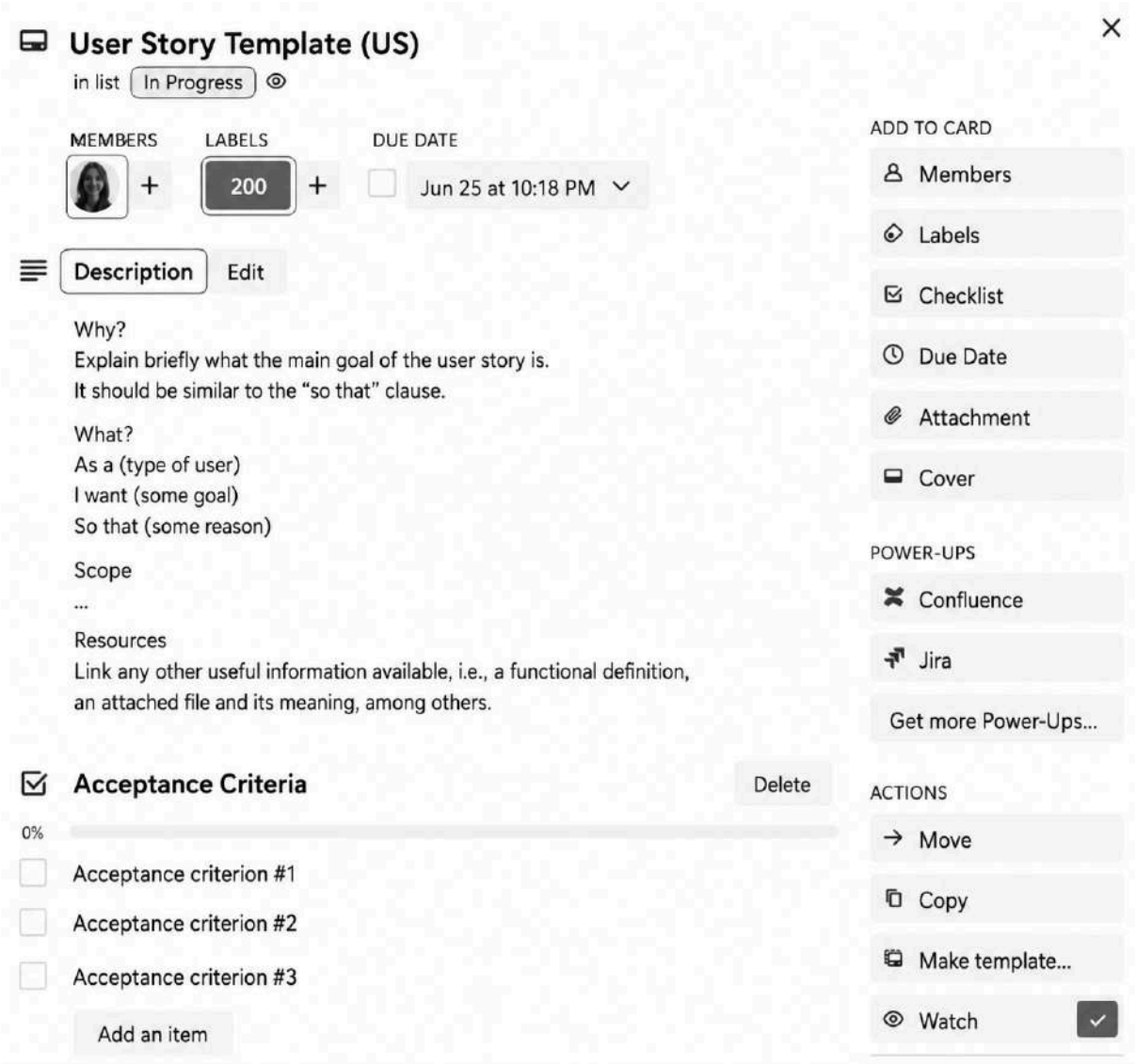


Figure 20: Example of a user story in Trello.

Once again, we have all the user story's information available on the tool's card:

- Description.
- Estimate.

- Acceptance criteria, and so on.

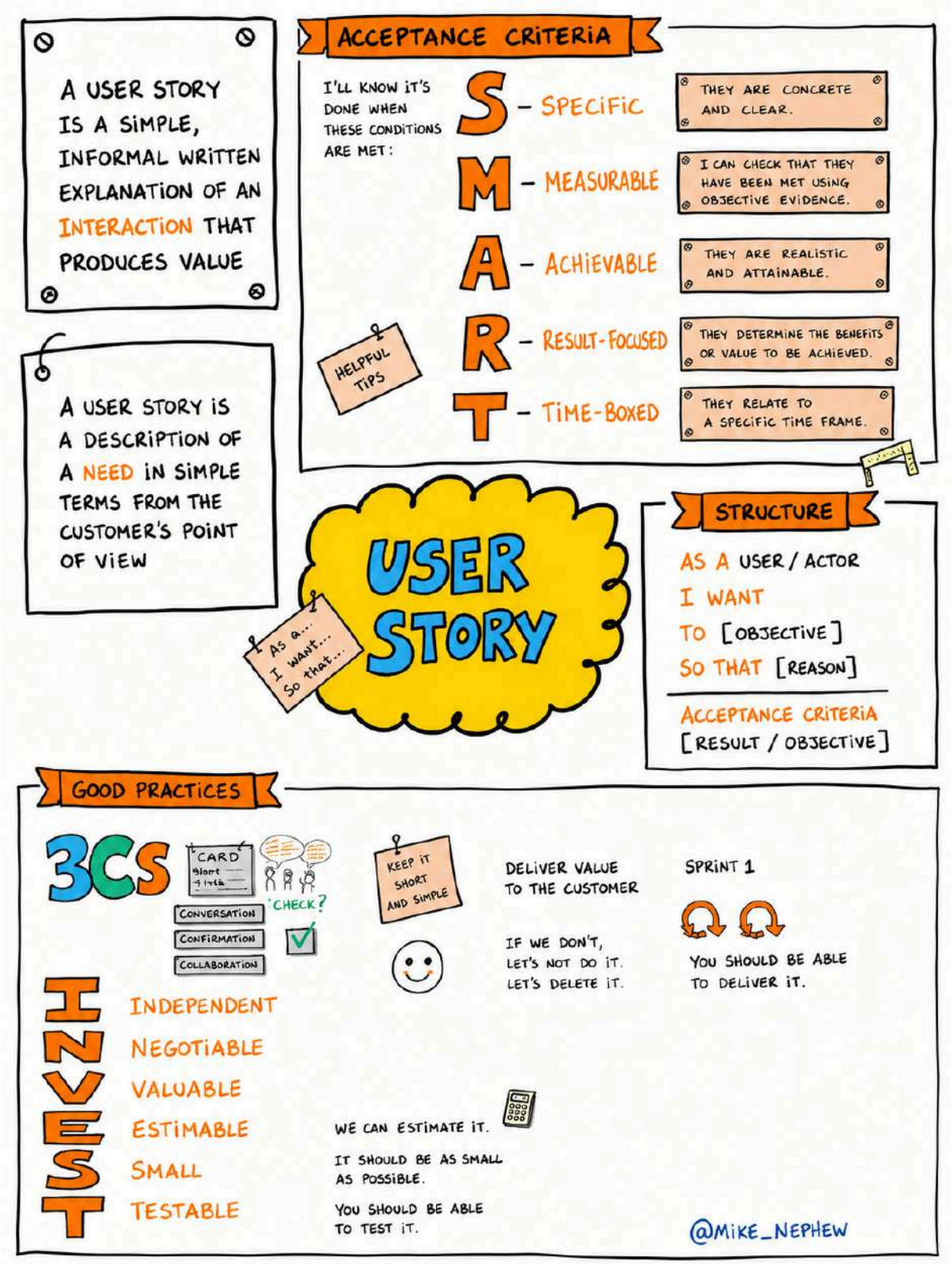
The user story can also be linked to and complemented with alternative tools such as JIRA or Confluence. Like any digital tool, they can be further enhanced with plugins or add-ons, which enable more features for our teams.

A very important **advantage** of this type of tool is that it makes it possible to run all the Scrum events, tracking, and artifacts with remote teams, and provides great freedom in distributing the work.

They provide useful reports and charts for tracking epics, stories, and releases—long-lived items that give value to the business. These can be observed from their entry into the backlog through to their deployment, and usage, quality, and outcome metrics can be analyzed.

Among the **disadvantages** that digital tools can have is that they are not equipped to give a complete view of the virtual board with all its user stories. And they can lead people to focus on learning the process of the specific tool, rather than encouraging the discovery of their own process and its continuous improvement.

Summary of user stories



Continuous improvement and quality control at Scrum Manager

Thank you for choosing Scrum Manager's training services.

Scrum Manager's quality control is based on its students' feedback. Your opinion helps us maintain the standard of our materials, courses, centers, and instructors.

If you have taken part in a training activity audited by Scrum Manager®, we kindly ask you—and are grateful—to rate the quality of the training you received. The information collected is anonymous.

You can send us your comments from the "Members' area" at https://scrummanager.com.

Bibliography

Adzic, Gojko

- (2009). *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*, Neuri Limited.
- (2011). *Specification by Example: How Successful Teams Deliver the Right Software*, Manning Publications.
- (2012). *Impact Mapping: Making a Big Impact with Software Products and Projects*, Provoking Thoughts.

Beck, Kent (1999). *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Bland, David J.; Osterwalder, Alexander (2019). *Testing Business Ideas: A Field Guide for Rapid Experimentation*, Wiley.

Clegg, Dai; Barker, Richard (1994). *Case Method Fast-Track: A RAD Approach*, Addison-Wesley.

Cockburn, Alistair (2001). *Agile Software Development*, Addison-Wesley.

Cohn, Mike

- (2004). *User Stories Applied: For Agile Software Development*, Addison-Wesley.
- (2009). *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley.
- (2024). "SPIDR: Five Simple but Powerful Ways to Split User Stories", Mountain Goat Software: https://www.mountaingoatsoftware.com/blog/five-simple-but-powerful-ways-to-split-user-stories

Cunningham, Ward (1992). "The WyCash Portfolio Management System", OOPSLA '92.

Ewel, Jim (2011). "Agile Marketing" [video], YouTube: https://www.youtube.com/watch?v=tllAqvvvM_A

Fowler, Martin (2009). "TechnicalDebt", martinfowler.com: https://martinfowler.com/bliki/TechnicalDebt.html

Garzás, Javier (2014). *Agilidad y Lean: gestionando los proyectos y negocios del s. XXI*, Miriadax.

Humanizing Work (2024). "Guide to Splitting User Stories", humanizingwork.com: https://www.humanizingwork.com/the-humanizing-work-guide-to-splitting-user-stories/

Jeffries, Ron; Anderson, Ann; Hendrickson, Chet (2001). *Extreme Programming Installed*, Addison-Wesley.

Krebsbach, Heather (2016). “Know Thy Customer: Agile’s Essential Guide to User Story Maps”, Atlassian: https://www.atlassian.com/blog/2016/05/guide-to-agile-user-story-maps

Lawrence, Richard (2009). “Patterns for Splitting User Stories”, Agile for All: https://www.agileforall.com/patterns-for-splitting-user-stories/

López, Gertrudis (2016). “Esquema con las 10 estrategias de división de historias de usuario”, MindMeister: https://www.mindmeister.com/app/map/682181724?t=LjoGABkbaF

Mollick, Ethan (2024). *Co-Intelligence: Living and Working with AI*, Portfolio.

O’Reilly, Barry (2020). “How to Implement Hypothesis-Driven Development”, barryoreilly.com: https://barryoreilly.com/explore/blog/how-to-implement-hypothesis-driven-development/

Patton, Jeff (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product*, O’Reilly Media.

Pichler, Roman (2010). *Agile Product Management with Scrum: Creating Products That Customers Love*, Addison-Wesley.

Rehkopf, Max (n.d.). “User Stories with Examples and a Template”, Atlassian Agile Coach: https://www.atlassian.com/agile/project-management/user-stories

Reinertsen, Donald (2009). *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas Publishing.

Rubin, Kenneth S. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley.

Savoia, Alberto (2011). *Pretotype It: Make Sure You Are Building the Right It Before You Build It Right*.

Torres, Teresa (2021). *Continuous Discovery Habits: Discover Products That Create Customer Value and Business Value*, Product Talk.

Verwijs, Christiaan (2015). “10 Useful Strategies for Breaking Down Large User Stories (and a Cheatsheet)”, The Liberators.

Wake, Bill (2003). “INVEST in Good Stories, and SMART Tasks”, XP123: https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Wallet, Thomas (n.d.). “Checklist INVEST para evaluación de historias de usuario”.

AI tools and platforms

Atlassian (2024). Atlassian Intelligence: https://www.atlassian.com/software/artificial-intelligence

Anthropic (2024). Claude: https://www.anthropic.com/claude

ClickUp (2024). ClickUp AI: https://clickup.com/ai

GitHub (2024). GitHub Copilot: https://github.com/features/copilot

Linear (2024). Linear AI: https://linear.app/features/ai

Notion (2024). Notion AI: https://www.notion.so/product/ai

OpenAI (2024). ChatGPT and GPT-4: https://openai.com/chatgpt

Information, resources, and professional community

Agile Club →

Skill Arena →

Community →

BoK/Wiki →

Observatory →

Comic strips →

About Scrum Manager® certification →

FAQs about Scrum Manager® courses and exams →

Social and professional networks

LinkedIn

Youtube

Table of illustrations