**Acceptance criteria, DoR, DoD,gerkin**

- **acceptance criteria:**

https://resources.scrumalliance.org/Article/need-know-acceptance-criteria

- **practical assignment: DoR (pbi, sprint):**

https://www.atlassian.com/agile/project-management/definition-of-ready
https://resources.scrumalliance.org/Article/pros-cons-definition-ready
https://resources.scrumalliance.org/Article/definition-vs-ready
https://www.scrum.org/resources/blog/walking-through-definition-ready

- **DoD (release, sprint, pbi)**

https://resources.scrumalliance.org/Article/definition-dod
https://www.scrum.org/resources/what-definition-done
https://resources.scrumalliance.org/Article/use-definition-agile-project

- **GERKIN**

https://cucumber.io/docs/gherkin/reference/

---

**From resources:**

When you're creating, building, or developing a product – whether it's software, a marketing campaign, or something else – how do you know that what you've made fulfills your customer's expectations? Enter, acceptance criteria: the conditions required for the customer, product owner, or stakeholder(s) to accept the work you've done.

These criteria are not inherent to the scrum framework, but many scrum and agile teams use them to organize work and deliver results that delight.

**Acceptance Criteria Defined**

-Acceptance criteria are defined as the conditions that must be satisfied for a product, user story, or increment of work to be accepted. While not a part of the Scrum Guide, AC can be a useful tool teams may choose to use to improve the quality of product backlog items.

Also shortened to the acronym AC, these conditions are pass/fail. Acceptance criteria are either met or not met; they're never only partially fulfilled.

AC are often expressed as a set of statements. These statements should be:

- Clear, so that everyone understands them
- Concise, so that there's no ambiguity
- Testable or verifiable
- Focused on providing customer-delighting results

Acceptance criteria do not focus on “how” a solution is reached or “how” something is made. Instead, they illuminate the “what” of the work you are doing. For example, the criteria may be:

Users can pay with Google Pay or Apple Pay at checkout.
The spirit of acceptance criteria is not to tell you how to do it, for example:

*Install a Wordpress plugin that allows you to create a checkout page.*
Or
*Write HTML that makes it possible to pay with Apple Pay or Google Pay.*

These statements get at how the work will be done, not the conditions for accepting the work. It’s up to the developers on the scrum team to decide the how of fulfilling the acceptance criteria.

The agile practice of formally stated acceptance criteria started in software development, but today they are applied to a wide array of deliverables in addition to software products across diverse industries, from app development to Human Resource departments and beyond.

**Acceptance Criteria vs. User Stories: What’s the Difference?**

For some scrum teams, a user story is the smallest chunk of work and one way to express a product backlog item. Your team may use something other than user stories to define and describe PBIs. Many teams simply leave their PBIs as PBIs.

Acceptance criteria are not the same thing as a user story. Instead, they complement one another.

A user story is a brief description of your customer's needs, written from their perspective. It describes their goal or problem they’re trying to solve. The acceptance criteria are what should be done to solve their problem or achieve their goal.

In this way, the user story describes the “why” of the work, while the acceptance criteria describe the “what.” The “how” is decided by developers as they work through the sprint.

**Who Should Write the Acceptance Criteria?**

What does your customer want, expect, or need from the product you are making? An answer to this question will help us identify the acceptance criteria. After all, our goal is to delight the customer.

Because we all work in unique industries and jobs, acceptance criteria don’t always originate from a traditional customer. Instead, it may be your product owner’s or stakeholders’ criteria or those of another type of client or user.

**A Process You Can Use to Create Acceptance Criteria**

Some of the opportunities for defining acceptance criteria include:

- Discussions with customers or clients
- Discussions with stakeholders
- During product backlog refinement
- During the scrum sprint planning event

- In team brainstorming
- After evaluating customer or end-user feedback

On a scrum team, the product owner is often responsible for writing these conditions of acceptance; however, the development team can and should be involved to offer their expertise and feedback while aligning with the product owner's expectations.

The scrum master on the team can support the process by looking for potential ambiguities with the criteria, helping the team understand the purpose of AC, and encouraging developers to speak up if the criteria are unclear.

In these ways, it truly can be a team effort, even though the product owner's proximity to the customer is often the starting point for generating the criteria.

**When Should You Write Acceptance Criteria?**

In scrum, we continuously talk about product backlog items as part of planning and refinement activities. Initial criteria are often identified during backlog refinement; however, finalizing the acceptance criteria should be done right before development begins.

Just-in-time acceptance criteria ensure you're working with the latest information, including the customer's goals and expectations. A good time to finalize the criteria is during the sprint planning event. The scrum team reviews the statements, discusses any issues or clarification needs, and decides whether the work can be brought into the sprint.

**How to Write Acceptance Criteria**

Templated approaches to writing the criteria can be found across the internet. When it comes to the Scrum Guide, there is no guidance because these criteria are separate from the lightweight framework of scrum. Try different formats – either custom or from a template – and see what works well for your team.

Use a bullet list, checklist, or verification list.

Many teams simply use a bullet list. You can easily generate the list and place it in the user story or wherever else you're organizing the work in your sprint.

Whether you use a bullet list, a table, a numbered list, a short description, or a sticky note, a custom approach can be a great option. Consider including your acceptance criteria format as the topic of a retro so you can inspect and adapt its effectiveness for your team.

**Use the scenario-based template.**

Other agile teams will use a format known as scenario-based, in which you use a formula: Given that; when; then:

- Given (some given context or precondition), when (I take this action), then this will be the result
-

**A Checklist for Writing Acceptance Criteria**

- Clear to everyone involved
- Can be tested or verified
- Either passes or fails (cannot be 50% completed, for example)
- -Focus on the outcome, not how the outcome is achieved
- As specific as possible (fast page load speed vs. 3-second page load speed)

**What Are Some Examples of Acceptance Criteria?**
For a checkout page on a store's website:

- PayPal, Google Pay, Apple Pay, and all major credit cards can be used to complete the transaction
- Shopping cart item(s) are displayed
- Items can be deleted from the shopping cart
- User is prompted to log in if they aren't already
- For an upcoming conference:

- ***Given*** that I'm a registered attendee
- ***When*** I enter the conference venue
- ***Then*** branded signs will lead me to the registration table

For an emailed water bill statement:

- Displays amount due
- Displays due date
- Displays possible late fee
- Allows user to log in and pay easily
- Lets user know how they can contact the office with questions or concerns
- For a health clinic's pre-appointment paperwork:

- ***Given*** that I'm an existing health center patient,
- ***When*** I schedule an appointment online,
- ***Then*** I will receive the fillable pre-appointment paperwork electronically

**Examples of Bad Acceptance Criteria**
When applying agile practices in the real world, you should feel empowered to experiment and see what works. But generally, you can see below why writing AC in certain ways is ineffective:

***Example 1***
*Acceptance criteria:*

- The online shopping cart uses checkboxes to select which items you want to delete

*Explanation:*
This criterion will probably go over fine with your development team and is unlikely to cause any immediate issues; however, it technically tells the development team "how" to make it possible to remove items on the checkout page: by including a checkbox.

***Example 2***
*User story:*

- As an online shopper, I want to be able to remove items from my shopping cart on the checkout page so that I can easily make last-minute adjustments without clicking the back button.

*Acceptance criteria:*

The online shopping cart allows users to remove items when they're on the checkout page, and when you view an item detail page, you can see a "Quick Shop" option
Explanation: The criterion technically goes beyond the scope of the user story. This can lead to too much work being lumped into one user story.

**Key Takeaway**

As it goes with all things agile and scrum, there's always some gray area between "good" and "bad" ways of writing acceptance criteria. The tool is yours to experiment with. Something that one team considers "bad" may work well for another team and its customers.

Use your retrospective to inspect how the acceptance criteria are working with the approach you're using. Adapt as necessary.

**Common Mistakes to Avoid When Writing Acceptance Criteria**

While there are many ways to customize acceptance criteria to your team and your context, there are also common pitfalls that tend to run counter to the purpose of AC:

- Ignoring the user perspective
- Telling the developers "how" to do the work
- Writing acceptance criteria too early
- Waiting until development is underway in the sprint to write the acceptance criteria
- The criteria are broad
- The criteria are vague
- There are a cumbersome number of criteria (a large quantity may indicate you need to break up the work into smaller parts)

Most agilists find that they can get the most benefit from acceptance criteria by writing them clearly while focusing on user experience and customer expectations.

**Acceptance Criteria vs. Definition of Done**

You may be wondering how these conditions of satisfaction are different from the definition of done (DoD). After all, the work of the product backlog item is done once the criteria are met, right?

While it's true that both DoD and acceptance criteria indicate a done state, they aren't quite the same.

The definition of done is typically expressed as a list of statements that must be met in order to call the work potentially shippable or to otherwise declare that work complete. The big difference is that the same DoD applies to every product backlog item and does not change between items.

The acceptance criteria are different for each product backlog item.

*Here is an example of DoD:*

- Code is completed
- Tested
- No defects
- -Live on production

*Here's another one:*

- Brand compliant

- Peer reviewed
- Stakeholders have been informed

Sometimes a team will call something “done” when referring to the acceptance criteria. If the terminology causes confusion, you may want to consider calling the work “accepted” instead of “done” when referring to the acceptance criteria.

**The Value of Acceptance Criteria to Agile Organizations**

Acceptance criteria benefit agile teams and scrum teams because the criteria:

- Create a clear line of understanding from end user to product owner to developers
- Let the developers know exactly what they need to accomplish during the sprint
- Can be created quickly and concisely, supporting the value of working software over comprehensive documentation (while also supplying documentation if it is needed for compliance in your industry)
- Create clarity and reduces ambiguity so that a self-managing development team can do their work efficiently
- Reduce the odds that your customer will feel their expectations were unfulfilled
- Provide criteria for testing the product

Try using them with your team if you aren’t already. You may find that acceptance criteria improve communication and collaboration, and connect you more closely with what your customer needs.

---

## Definition of Ready (DoR) Explained & Key Components

You’re a project manager, and your team is about to embark on their next sprint. But are the tasks ready for your team to work on?

To determine this, you'll need a Definition of Ready (DoR), which is vital in Agile project management. This ensures your team can effectively tackle the task. It can also help your team with backlog refinement.

On this page, we'll break down the Definition of Ready in Scrum and Agile approaches and explain how you can gauge if a task meets its criteria.

**What is Definition of Ready?**

A Definition of Ready (DoR) allows you to evaluate work before your team starts on it. It defines a task, user story, or story point for your team. If you use a Scrum approach, the DoR means you can take immediate action. Before starting a project, your team needs to know:

- Your target customers: What are their motivations, pain points, and needs?
- The project goals: What's the purpose of the project?
- The required tasks: Are they valuable, both for the business and the user? Are they clear and feasible?
- Technical requirements: Do they have the necessary resources? Do they understand the technical approach or solution? Can you test it?

- Time estimates: What's the timeline to complete the work? Have stakeholders and the team agreed on an end date?
- The definition of done (DoD): What does completion, or DoD, look like? What Scrum metrics do you plan to use to evaluate success?

Only once the team grasps the project's scope can the work move from product backlog to active. Everyone must collectively agree on whether the work is ready. That way, you'll reduce any back-and-forth on the team's workload.

**Key components of DoR**

There are six critical components of a DoR that you'll want to consider. These components assist you in your Agile planning. The nickname for these components is the INVEST method, which stands for:

- ***Independent*** **-** Whatever backlog item you're working on must not depend on any other task. It must be self-contained. Your team will avoid any unnecessary work this way.
- ***Negotiable*** **-** A task shouldn't be rigid. You must be flexible enough to consider other options the team might bring.
- ***Valuable*** **-** There must be a purpose to your work. More importantly, it must add value to the product, the customer, and the business.
- ***Estimable-*** The task must be feasible, achievable, and measurable. Your team needs to know how much time and effort you will require of them. If the sprint requires multiple tasks, the same goes for each.
- ***Small*** **-** The work must be manageable. If a task is complex, you should be able to break it down into smaller ones. Doing so prevents fire drills and working in overdrive to meet unreasonable deadlines. And your team won't burn out.
- ***Testable*** **-** Specify the success and completion criteria based on business and user needs. These allow your team to evaluate whether the task is complete.

**Why is Definition of Ready important?**

A clear DoR will instill confidence and set expectations with you, your team, and your stakeholders.

*Here's why a DoR is essential for your company:*

- ***Enhances communication***: DoR helps your team better communicate whether a task is ready for work, thereby minimizing confusion and delays.
- ***Improves efficiency:*** DoR allows your team to execute tasks efficiently because team members can move forward quickly knowing that they have a full understanding of the technical requirements.
- ***Reduces errors:*** With a firm grasp of a task, the team can mitigate errors during the sprint.
- ***Promotes collaboration:*** DoR can be considered as a working agreement and promotes healthy collaboration across the entire team.
- ***Empowers your team***: DoR gives your team ownership and control over their work. "Remember that the DoR is created for the team, by the team," says Atlassian's Modern Work Coach Mark Cruth. "It's all about what the team needs to feel comfortable and start work."

**How to create an effective DoR**

Now that you understand the DoR, it's time to create one. Let's go through the step-by-step process of creating an effective DoR for your company:

1. ***Define your team's responsibilities***. Ensure each team member knows what they're responsible for.
2. ***Involve critical stakeholders***. You'll want their input and buy-in on the DoR criteria to mitigate any scope creep.
3. ***Specify the DoR structure and format***. What is your checklist for? What defines ready work? How does your team determine what's ready? These are important questions to ask yourself when creating a DoR.
4. ***Keep your backlog groomed***. Nothing beats a well-kept backlog. Your team must examine whether an item fits the product roadmap and is still relevant.
5. ***Identify and define user stories***. Determine a user story's criteria and whether it's feasible.
6. ***Ensure it meets the INVEST method***. A DoR checklist determines if a task is independent, negotiable, valuable, estimable, small, and testable.
7. ***Review your DoR regularly***. Priorities shift, and your DoR needs to reflect these changes. Otherwise, your team may not be working as efficiently. “If you notice the team is regularly not completing all their work in a sprint, or there is a lot of scrambling to understand work within the sprint, it likely means your Definition of Ready needs to be reviewed and updated,” explains Cruth.

Now that you know all the steps, you can create a Definition of Ready checklist within Jira to ensure your team is on the same page on how to complete a task and what the expectations are.

**Fine-tune your team's DOR with Jira**

Ready to start on a DoR? Jira makes it easy for software teams to be agile and ensures your work is meaningful for customers and the business as a whole.

Define your DoR and weave those requirements into Jira. Create custom fields or download an extension to create issue checklists in Jira on each Jira issue. If you have different types of work, create a different DoR by customizing Jira issue types.

Jira also simplifies the process of sprint backlog refinement. With Jira, your software team can:

- Determine what tasks and user stores are ready and actionable.
- Break down large tasks into smaller, manageable sprints.
- Execute sprints efficiently and stay on task.
- Improve velocity with the least amount of friction.

**Definition of ready: Frequently asked questions**

***What is an example of a DoR?***

A DoR lets your team know if a backlog item is ready for a sprint. Here’s a Definition of Ready example for a bug fix:

A bug fix might've lingered on your backlog, but now you can move it up. That's because your team:

- Determined that it’s actionable. The team feels the bug fix is feasible and independent of other tasks.
- Established a shared understanding. The team collectively grasps what the bug fix entails. They know what they need to make it happen.
- Know its value. The team understands the impact of the bug fix on customers and the business.
- Set criteria and a timeline for completion. The team estimates the time to complete the fix based on key benchmarks.
- Believes the bug fix is testable and verifiable. The team can test the fix to see if it works and demonstrate it to stakeholders.

**How does DoR fit into Agile project management?**

A DoR allows your team to be agile. It's perfect for Agile project management because your team will:

- Know what tasks they can work on within a reasonable timeline.

Work effectively because they know all the dependencies and requirements.

- Have all the necessary information to ensure they can complete the scope of work on time.

**What is the difference between DoR and DoD?**

DoR and DoD are both crucial touchstones at either end of a sprint, but there are a few key differences between them:

- DoR: This is the criteria to determine if a task or user story is ready for your team to tackle.
- DoD: This is the benchmark to evaluate when a task or user story is complete.

---

## Pros and Cons of a Definition of Ready

A definition of ready (DoR) is not a part of the scrum framework, but some scrum teams, especially those new to scrum, may choose to use a DoR as training wheels to organize the work of their sprints when they are not used to creating good product backlog items. Often expressed as a simple checklist, the DoR defines the conditions of a ready piece of work.

A scrum team uses short, fixed-length cycles known as "sprints" to deliver valuable, usable increments. The scrum team consists of developers (the people doing the work), a scrum master, and a product owner. Developers can work in virtually any domain, not just people who write code. The scrum team is self-managing, deciding what portion of their team's product backlog to work on in a sprint to support the goal for that particular sprint.

**What Is a Definition of Ready?**

Teams using the scrum framework decide what subset of product backlog items they will focus on during a sprint. For some new scrum teams, an optional DoR checklist helps them identify pieces of work that are the most ready for them to work on.

As with any tool, a definition of ready has advantages and disadvantages. Some experienced agilists advise against a DoR because it tends to place a process and tools over individuals and interactions, which is the exact opposite of the first value of the Agile Manifesto: Individuals and interactions over processes and tools.

**An Example of a Definition of Ready**

If you decide to try out a DoR as a team, it should be tailored to creating clarity for you and your teammates. It should answer the question — what are a few conditions we'd like to know are satisfied before we start working on this backlog item?

*Here's a basic example* of DoR: For a product backlog item :

- There's a clear business value
- Acceptance criteria are clear and defined
- Dependencies are identified
- PBIs should be small enough to be completed within a few days
- Should be sized appropriately (per the Scrum Guide, the PBI should have a description, order, and size)

In this example, a scrum team has decided they'd like to know that the ready PBI is aligned with a business goal, has clear acceptance criteria, and has external dependencies identified.

**Disadvantages of a Definition of Ready**

DoRs can get in the way of a team's agility. It places documentation, processes, and sticking to a plan over individuals, interactions, and collaboration, which is the inverse of the values of the Agile Manifesto.

Here are some of the potential pitfalls of a DoR:

- As a scrum team, you want your focus to be on the sprint goal and achieving incremental progress, not on doing the pre-work to get a checklist 100% complete before you can start working on certain product backlog items. DoR can shift the focus to the DoR itself.
- A definition of ready can lead to a sequential, waterfall approach to getting work done. This occurs when the DoR acts like a gate, blocking certain backlog items from the sprint until all of the pre-work in the DoR is completed.
- Many novice developers put the onus on the PO and others outside the scrum team to get the PBIs "ready" as per DoR and (incorrectly) consider that their job is only to build the PBIs once "ready." This is incorrect. The Developers should collaborate with others (as appropriate) to prepare the PBIs.

Uncertainty is inherent in complex work. DoR should not be used as a tool to avoid uncertainty but instead, as a tool to gain shared understanding about what is clear and what is uncertain and how that uncertainty can be resolved before and during the sprint. The disadvantages above arise when a DoR is written like a strict set of rules. Instead, create criteria that act as guidelines to support your team's ability to focus on the sprint goal.

**Potential Benefits of a DoR**

The best DoR offers guidance, especially to new scrum teams, to plan the work of a sprint. The DoR doesn't block certain PBIs from being worked on but instead creates a shared understanding of what needs to be done to gain more clarity about the product backlog item. Suppose a DoR includes "dependencies are identified," and the scrum team hasn't had a chance to identify any dependencies yet. In that case, they know that the unknown variable will be a consideration if they decide to work on the item this sprint.

Here a some of the other benefits of a DoR:

- A DoR may help teams new to scrum consider coordination and aspects of collaboration within and across the scrum team
- A self-managing team working with a big backlog knows the degree of readiness of a product backlog item
- The scrum team can make informed estimations of the effort that will go into an item based on its readiness
- DoRs reduce the risk of not completing all of the work the team has forecasted they can complete in a sprint (by — hopefully — minimizing unexpected, extra work)

Some scrum teams find that a definition of ready gets in the way of focusing on the goal; they find themselves pursuing a stringent checklist over collaborating to provide increments of value. Experiment with a DoR to see if it supports or hinders you and your agile teammates. The scrum team should be able to inspect and adapt its practices based on how useful or not useful DoR is.

**Definition of Ready vs. Definition of Done**

While a definition of ready describes ready work, a definition of done describes work (i.e., product backlog items) that meets the quality conditions to be considered done.

In scrum, the definition of done (DoD) often functions as a checklist that lets you know when a PBI becomes an increment by meeting the quality measures for the product. The DoD creates transparency for the increment; everyone is clear on what has been completed for the PBIs.

Another difference is that the DoD is part of the Scrum Guide, while the DoR is not. That's because a DoD acts as a commitment to the increment, an artifact in scrum. Scrum has three artifacts: the product backlog, sprint backlog, and increment.

The definition of ready is not part of the scrum framework but is a tool some new scrum teams use to know how ready a PBI is for a sprint. It's completely optional and not the only way to get an item as ready as possible.

**What's an Example of a Definition of Done?**

Here is an example of a DoD checklist:

- Design reviewed
- Code completed
- Tested
- No known defects
- Acceptance criteria are met or negotiated

A PBI cannot be an increment if it doesn't meet the team's definition of done. By definition, PBIs that are experiments or hypotheses (which do not deliver value) do not meet the DoD and are not part of an increment. It's usually moved back to the product backlog for reordering and reconsideration if it's not done.

**Acceptance Criteria vs. Definition of Ready**

PBIs can often be written as user stories (note: user stories are not part of scrum). User stories are associated with one or more acceptance criteria. The AC are the conditions that must be met to complete the user story satisfactorily. The definition of ready means the PBI is ready to be worked on. Definitions of ready often include "clear acceptance criteria" as part of the definition.

*If the DoR for a new steering wheel design is that it:*

- Has acceptance criteria
- Has a clear business value
- Has been estimated by the team

Then the acceptance criteria may be:

- A new design is created
- The design will work with the next year's models
- Feedback about the previous design has been incorporated

Neither the acceptance criteria nor the DoR tells a scrum team "how" to do the work. The AC lets the team know "what" is expected, and the DoR informs them about item readiness.

**What Makes a Good Definition of Ready?**

A good definition of ready is a guideline that shows a scrum team how ready a PBI is to be worked on. It should be:

- Clear and concise
- -Understood by the developers
- Transparent and visible
- Suggests readiness but isn't used a strict rule
- The scrum team should inspect and adapt its usefulness

While there's no right or wrong number of DoR conditions, an exhaustive list of pre-work will likely complicate the team's ability to start work items. Exhaustive pre-work also leads to performing work in stages, where one piece of work isn't started until a previous stage is fully completed. This will affect the agile team's ability to work fast, flexibly, and effectively.

**How to Use a Definition of Ready**

If your team decides to use a definition of ready, collaborate with all your team members to determine what should be included. You could do this as part of creating a team working agreement, or you may create a DoR as needed on certain backlog items during refinement — for example, an item that you know will depend on another team to complete it.

Use the definition of ready to refine product backlog items, checking in with your team to see how you can get the PBI to be a "more ready" state during refinement.

You may not completely fulfill the DoR before it becomes clear you need to pull in the work; unlike a definition of done, the DoR doesn't need to be 100% fulfilled. You should revisit

and rewrite definitions of ready if they are causing your team to fall into a sequential, waterfall-like approach, or you may do away with a DoR altogether.

And with all things, consider making your DoR the subject of a sprint retrospective. Doing so will allow you and your teammates to discuss the value of the DoR, how it's working, what isn't working well, and whether any changes can be made to the DoR to support your collaboration as a scrum team and the value you provide your customers.

---

**Definition of Done vs. Definition of Ready**:

While the definition of done (DoD) is part of scrum, a definition of ready (DoR) is an external and optional tool.

Although these two terms seem similar, they are quite different. On the one hand, the DoD defines when a product backlog item becomes an increment. On the other hand, the DoR can help a scrum team identify when a product backlog item is ready to work on.

While the scrum team must understand and effectively use a great DoD, you may experiment with a DoR and see if it’s right for your scrum team.

**DoD vs. DoR**

The DoD is a shared understanding among scrum team members of what it means for a product backlog item (PBI) to be considered complete. This definition is agreed upon by the entire team, and any criteria that are deemed important to meet are included in it.

Some examples of what might be included in a definition of done are:

- Work has been fully reviewed by another team member
- Work has been tested, and no errors were found
- Documentation has been updated

The definition of done applies to all work in the backlog. Contrast this with acceptance criteria, which are unique to each PBI, and define the criteria that must be met for that specific item.

While optional, the idea behind the definition of ready is that it outlines the criteria for a product backlog item to even be considered by the team for bringing into their sprint. By having these criteria in place, the team can more effectively commit to work with full knowledge of its readiness.

Although the Scrum Guide does not mention a definition of ready, some teams find it to be a valuable tool for managing their work. Some examples of what might be included in a DoR are:

- The PBI is written in a clear, concise way
- Related acceptance criteria have been defined and are testable
- Any dependencies have been identified and addressed
- All necessary stakeholders have approved the PBI

By having a definition of ready in place, the team can plan their work, avoid surprises, and focus on delivering high-quality work. But a DoR can lead to delays as teams chase down the

completion of a checklist before starting their work. This interferes with the team's ability to behave flexibly and incrementally.

**Differences in Content: DoD vs. DoR**

One of the biggest differences between the two is that the definition of done refers to the PBI itself, while the definition of ready often refers to externalities.

*For example, a definition of done might be a checklist like:*

- PBI meets applicable tests
- PBI has no known defects
- PBI meets acceptance criteria (possibly with negotiation)
- PBI documentation complete

The definition of done is related to testing requirements, acceptance criteria, and any other details necessary for a PBI to become an increment. These are intrinsic to the PBI.

*In contrast, a definition of ready might be a checklist with the following:*

- No blockers have been found that cannot be addressed by the team during the sprint
- The team has the information and skills necessary to complete the item
- The team has what they need to start working on the item
- The definition of ready is related to many things extrinsic to the PBI, including dependencies, team skill and information, and decisions made by the product owner.

**The Right Time to Use Each Definition**

The definition of done is the relevant commitment for the increment and is an essential part of scrum. The team uses their DoD as they determine whether something they've been working on is done. The DoD may also be used during sprint planning because it helps the team understand all of the work involved with a PBI. Finally, they may also share the DoD in the sprint review so that stakeholders can inspect how the outcomes or PBIs align with the definition.

The definition of ready is not an integral part of scrum, and teams may utilize it at different points during a sprint, but most often during refinement and sprint planning.

While refining the PBIs, the team uses the DoR to see if there is any work that must be completed in the PBIs before taking them to the next sprint planning.

During sprint planning, teams add PBIs to the sprint backlog. This is the first event of the sprint, and it's when teams utilize their definition of ready to decide if they should add PBIs to the sprint backlog. Adding PBIs to the sprint goals when they are not ready can create unnecessary impediments to creating valuable increments.

**Updating and Revising the Definitions**

The definition of done and the definition of ready should be changeable and might sometimes need to be changed. A good course of action is to have a conversation in a retrospective to identify the possible reasons that the DoD or DoR is not serving the team. Inspecting the root cause may reveal a way to better meet the definition or a need to modify it.

You may need to update either definition for the following reasons:

- The definitions are exhaustive and impeding the team
- A new learning or discovery calls for an update
- Criteria in the definition(s) is no longer valid or relevant
- The definition(s) aren't serving their intended purpose

Scrum teams are learning fast, and their DoD and DoR should be flexible enough to reflect those learnings.

**Team Maturity May Influence the Definitions**

On a team new to agile, you may notice a shift in the DoR and DoD as the team matures and gains experience. Commonly, this shows up as new agile teams having extensive ready lists and short done lists.

Oftentimes, new teams will attempt to have a long list checked off before they get started on a PBI. Perhaps they're not yet comfortable with transitioning away from a phased project to an iteration or sprint, and a long ready list feels like a good way to prepare for the iteration. As the team gains experience together, their readiness definition may shrink as they learn to start a PBI without exhaustive pre-work.

At the same time, they may not yet know how to define high-quality standards in their definition of done. That list may be short because the team isn't quite sure what conditions indicate a done product or service.

The new team may at first lack all of the skills and knowledge to complete everything on a high-quality DoD. And that's okay! Agile teams solve this issue over time by learning new skills or adding new members who possess the missing skills.

Once they have this fully cross-functional team built, their DoD may get longer and more well-rounded because they will be able to perform all of the activities that should be in the definition.

**Two Seemingly Related (But Quite Different) Definitions**

While the DoD sets the standard for what work must be completed, the DoR ensures that the team is working on the PBIs with the right level of preparation. By having both in place, teams can improve their efficiency and effectiveness by working collaboratively towards their goals.

---

**Walking Through a Definition of Ready:**

**A glance back at “Done”**

A few weeks ago we looked at the Definition of Done, which describes the conditions which must be satisfied before a team’s deliverables can be considered fit for release. That’s the acid test of what “Done” ought to mean. Can a team’s output actually be deployed into production and used immediately, or is any work still outstanding? We saw that a team’s Definition of Done will often fall short of this essential standard, and “technical debt” will be incurred as a result. This debt reflects the fact that certain things still need doing, no matter how small or trivial they might be held to be. Additional work will need to be carried out before the increment under development is truly usable. Perhaps there might be further tweaks to be done, or optimizations, or tests, or integration work with the wider product. Any such technical debt will need to be tracked, managed, and “paid off” by completing the outstanding work so the increment is finally brought up to snuff.

In other words, an increment is not truly complete if it is not of immediate release quality, since more work must be done before the value invested in it can be leveraged. Hence a Definition of Done must be articulated clearly, no matter how shoddy it might be. Only then can shortcomings in the standard for release be acknowledged and remedied. A Definition of Done is instrumental to achieving transparency, and of course any "deficit for release" should be made equally plain. A team which runs with a deficit for release cannot be said to be operating in an agile manner, since no increment will be released and inspected and adapted, and this must be recognized. Any technical debt incurred as a result of that deficit may then be tracked, so the nature and extent of it can be understood. This will give stakeholders an improved picture of how much work genuinely remains to be completed, and of the gaps which lie between the current operating model and robust agile practice.

**"Done" can be at multiple levels**

The Scrum Guide tells us that there can be multiple levels of "Done". The Definition of Done sensu-stricto must of course pertain to the increment, since that is the artifact which is ultimately subject to release. However, this does not imply that "Done" must be an atomic and indivisible measure. There's nothing to stop the evidencing of "Done" from being built up gradually as work is performed, and there can be distinct advantages in doing so.

In a car factory for example, quality is likely to be inspected on an ongoing basis as work is completed. The testing of components as they are brought together is not likely to be deferred until the car finally rolls off the assembly line. The risk of finding problems so late in the day would clearly be high, and the opportunity to provide meaningful remedy at such a late stage would be limited. Extensive re-work might be necessary by that point. Testing at multiple discrete points on the assembly line, each time significant value is added, reduces the magnitude of such risk. Defects or other problems can be detected and resolved as close as possible to the time and place of the work being carried out. The opportunity for complex rework to accumulate, and for waste to be compounded, is thereby reduced.

A software development team can also use multiple elevations of "Done" in order to inspect and adapt work on an ongoing basis, and thereby assure quality in the timeliest possible manner. For example, if a user story is being implemented then it is reasonable to expect that all of its acceptance criteria ought to be satisfied. Sometimes this particular level is referred to as "Story Done". Before that work can be committed to a repository though, it might have to be peer reviewed, documented, or expected to satisfy a particular type or degree of test coverage. Such criteria would represent an additional level of "Done" to be satisfied before work-in-progress can be committed and integrated. The wider "Definition of Done", which relates to the completed increment, would comprise these and all levels of "Done". However, it can be seen that the Definition of Done is not asserted atomically just prior to release, but rather at each point where effort has been invested in development and value is being added. Any problems can be swiftly uncovered, corrected, and action taken to prevent their re-occurrence. Rework and waste is minimized. In effect the DoD consists of multiple discrete checks and testable assertions, each of which is applied as closely as possible to the time and place of the relevant activities being carried out.

**Towards a "Definition of Ready**"

Now, since a level of "Done" may be applied to each station in a workflow, it is reasonable to surmise that this includes the transitioning of work into the Sprint Backlog itself.

In other words, before work can be planned into a Sprint, the relevant items on the Product Backlog must be "Done" in terms of being sufficiently well described and understood. The Development Team must grasp enough of its scope to be able to plan it into a Sprint, and to frame some kind of commitment regarding its implementation so a Sprint Goal can be met.

In practice, this standard is often referred to as a "Definition of Ready". During Product Backlog refinement, detail, order, and estimates will be added or improved until the work on the backlog meets this condition. In effect Product Backlog refinement helps to de-risk Sprint Planning. By observing a Definition of Ready, the chances are reduced of a Sprint starting where Development Team members immediately shake their heads at Product Backlog items they do not sufficiently understand.

In fact, many teams struggle to implement a Definition of Ready. This is partly because refinement is hard. It takes discipline to reserve 10% or so of a team's time during a Sprint - as the Scrum Guide recommends - so that the Product Backlog is adequately prepared. Some teams will cut this agile hygiene short, or otherwise try to wing it. The result is likely to be a shoddy Sprint Planning session where the body of work selected is not sufficiently well understood, and the team's ability to frame a suitable commitment is glossed over or faked. Yet even quite mature agile teams can fail to master "Ready".

**How much is too much?**

Experienced developers are usually aware that a user story is meant to represent an ongoing and evolving conversation with stakeholders, and not a fixed specification. How then, they sometimes wonder, can a level of "Done" be applied to the refinement of such a backlog item? How much envisioning and architecting, and analysis and design, to say nothing of exploratory spike investigation, may reasonably be performed? How much is too much, and when does all of this activity start to encroach onto actual development work? Might a "Definition of Ready" potentially be an anti-pattern? Isn't there a danger of the Sprint boundary becoming rather mushy, if backlog items are worked on both before and during a "Sprint"? How do we stop the Sprint construct from fading into irrelevance? If we are to stop things from going to the dogs, how can team members tell when "enough" refinement has been done, and any further evolution must constitute development effort?

To understand the answer, we need to bear in mind that Product Backlog refinement ought to de-risk Sprint Planning. Enough refinement will therefore have been done when a team can plan it into in their Sprint Backlog as part of their achievable forecast of work. The acid test of that condition can be brutally simple. If work has been refined to the point that a team can estimate it, and it is thought small enough to be planned into a Sprint without being broken down further, then that might well be enough. Whether getting to that point requires analysis or design, or both, or even some coding in the form of an exploratory spike, is completely irrelevant. Enough must be done to make the scope of the item comprehensible in terms of its probable size. Any more refinement than that is waste, while any less will not be enough.

This condition - having Product Backlog Items which are sized and sufficiently granular - can represent a belt-and-braces Definition of Ready. It may be adequate for a team to pick up and run with, and it is certainly a condition which they should keep in the back of their minds. After all, if this criterion doesn't hold, should they really frame a commitment which involves that work?

A team can up their game by asserting further conditions which must be met before work is considered ready for planning. It is reasonable, for example, to expect acceptance

criteria to be articulated for a backlog item such as a user story. Without such criteria developers may not really understand the scope of the work or how it will be tested and validated. In fact, it can be hard for developers to truly estimate work at all unless the acceptance criteria are defined. That's where the meat often is. They may also reasonably expect clear value to be associated with the item, and for this to be expressed in a succinct way which makes it quite evident. The standard user story format of "As a <type of user>, I want <some goal> so that <some reason>" is helpful and a team may reasonably expect something along these lines. They may further expect each item to be actionable in its own right and free of dependencies. Additionally, they may expect enough flex in the item to allow for experimentation in finding the best way to implement it.

**Example Definition of Ready**

These considerations are often summarized as the "INVEST criteria", and they provide us with a useful Definition of Ready which can be applied to Product Backlog Items. By actively participating in Product Backlog refinement, a good Development Team will collaborate with the Product Owner in making sure that a standard such as this is observed.

- **I (Independent).** The PBI should be self-contained and it should be possible to bring it into progress without a dependency upon another PBI or an external resource.
- **N (Negotiable).** A good PBI should leave room for discussion regarding its optimal implementation.
- **V (Valuable).** The value a PBI delivers to stakeholders should be clear.
- **E (Estimable).** A PBI must have a size relative to other PBIs.
- **S (Small).** PBIs should be small enough to estimate with reasonable accuracy and to plan into a time-box such as a Sprint.
- **T (Testable).** Each PBI should have clear acceptance criteria which allow its satisfaction to be tested.

---

## What is the Definition of Done (DoD)?

Definition of done (DoD) is crucial to a highly functioning scrum team. The following are characteristics that you should look for in your team's definition of done. Verifying that your team's DoD meets these criteria will ensure that you are delivering features that are truly done, not only in terms of functionality but in terms of quality as well.

**Definition of Done is a Checklist of Valuable Activities Required to Produce Software**

Definition of done is a simple list of activities (writing code, coding comments, unit testing, integration testing, release notes, design documents, etc.) that add verifiable/demonstrable value to the product. Focusing on value-added steps allows the team to focus on what must be completed in order to build software while eliminating wasteful activities that only complicate software development efforts.

**Definition of Done is the Primary Reporting Mechanism for Team Members**

Reporting in its simplest form is the ability to say, “This feature is done.” After all, a feature or product backlog item is either done or it is not done. DoD is a simple artifact that adds clarity to the “feature is done” statement. Using the definition of done as a reference for this conversation a team member can effectively update other team members and the product owner.

**Definition of Done is Informed by Reality**

Scrum asks that teams deliver “increments of value” at the end of every sprint. In reality, many teams are still working towards an increment of value. Such teams may have a different DoD at various levels:

- Definition of done for a feature (story or product backlog item)
- Definition of done for a sprint (collection of features developed within a sprint)-
- Definition of done for a releasable increment

There are various factors that influence whether a given activity belongs in the DoD for a feature, a sprint or a release. Ultimately, the decision rests on the answer to the following three questions:

- Can we do this activity for each feature? If not, then
- Can we do this activity for each sprint? If not, then
- We have to do this activity for our release!

Certified Scrum Trainer Chris Sterling recommends that for activities that cannot be included for a sprint/feature, teams should, “Discuss all of the obstacles which stop them from delivering this each iteration/sprint.”

*Common root causes for impediments include:*

- Team does not have the skillset to incorporate activities into the definition of done for a sprint or for a feature.
- Team does not have the right set of tools. (Example: continuous integration environment, automated build, servers etc.)
- Team members are executing their sprint in mini-waterfalls.

* Aha! Here’s an opportunity to be more cross-functional and share responsibilities across functional silos.

**Definition of Done is Not Static**

The DoD changes over time. Organizational support and the team’s ability to remove impediments may enable the inclusion of additional activities into the DoD for features or sprints.

**Definition of Done is an Auditable Checklist**

Features/stories are broken down into tasks both during sprint planning and also within a sprint. The DoD is used to validate whether all major tasks are accounted for (hours remaining). Also, after a feature or sprint is done, DoD is used as a checklist to verify whether all necessary value-added activities were completed.

It is important to note that the generic nature of the definition of done has some limitations. Not all value-added activities will be applicable to each feature since the definition of done is intended to be a comprehensive checklist. The team has to consciously decide the applicability of value-added activities on a feature-by-feature basis. For example, following user

experience guidelines for a feature that provides integration point (eg. web service) to another system is not applicable to that particular feature; however, for other features within the system that do interface with a human being, those user experience guidelines must be followed.

**Summary**

The definition of done is orthogonal to user acceptance criteria (functional acceptance) for a feature. It is a comprehensive checklist of necessary, value-added activities that assert the quality of a feature and not the functionality of that feature. Definition of done is informed by reality where it captures activities that can be realistically committed by the team to be completed at each level (feature, sprint, release).

---

## What is a Definition of Done?

**The Definition of Done Commitment**

If you are just getting started using Scrum and/or learning about it, you are going to hear a lot about Done and the Definition of Done. Think of Done as all of the ingredients it takes for an Increment of product to be complete. The Definition of Done is the commitment by the Developers for the Increment, much like the Sprint Goal is the commitment by the Developers for the Sprint Backlog and the Product Goal is the commitment by the Product Owner for the Product Backlog. The Definition of Done includes all of the characteristics and standards an Increment needs to meet in order to be released.

The Scrum Guide says the Definition of Done is a formal description of the state of the Increment when it meets the quality measures required for the product. Once the Definition of Done is met, the Increment is Done and can be delivered.

The Definition of Done creates transparency by providing everyone a shared understanding of what work was completed and what standards were met as part of the Increment. If a Product Backlog Item does not meet the Definition of Done, it cannot be released yet. Think of the Definition of Done as the standards set for the products delivered.

Sometimes the Definition of Done for an Increment includes the standards of the organization. In that case, all Scrum Teams must follow these standards as a minimum. They can elaborate on it with any other standards or characteristics that need to be met for the product. If there are not specific organizational standards, the Scrum Team must create a Definition of Done appropriate for the product.

**How to create a Definition of Done**

The Definition of Done creates transparency by providing everyone a shared understanding of the required standards of work that was required as part of the Increment. This can be as simple as the team collaborating and writing everything down together. Some Scrum Teams may also incorporate brainstorming activities from resources such as Liberating Structures.

**To give context, below are some examples of items you may find in a Definition of Done:**

*Some examples of the items in a Definition of Done for a written Marketing Case Study:*

- Meets featured client branding guidelines
- Written in AP style
- Reviewed by the client featured and feedback received
- Feedback implemented
- Final Draft Approved by Client

*Some examples of the items in a Definition of Done for a health-focused software application*

- All testing completed
- No known defects
- Code review completed and passed
- Meets HIPAA Compliance standards

Meets general security requirements

Once all of the items in the Definition of Done are checked off and complete, this Increment is considered Done. Of course, Scrum is used for complex work and many complex characteristics may be added to a Definition of Done to make it more stringent.

---

## Why Use a Definition of Done in an Agile Project?

If you want to learn to play the piano, it's going to be a tough endeavor if it takes 30 minutes before your piano produces a sound after you press a key.

When you demonstrate your software just before the deadline, you know for sure that the project won't finish early. If you demonstrate it every week and implementation is done based on the product owner's priorities, there is a big chance the product owner will approve the application even before all requested features are implemented.

**Feedback will help you improve, learn, and reach your goal more effectively.**

It's important that you get feedback early and often, and iterative development can facilitate this. What you actually get feedback on is defined in the Definition of Done. The Definition of Done defines all steps necessary to deliver a finished increment with the best quality possible at the end of a sprint. The more you do in your sprint, the more you get feedback on, and the more you can improve and learn.

This introduces the first reason for using a Definition of Done:

**Definition of feedback and improvement**

When the Definition of Done is complete, it will define all steps to deliver a finished increment, and therefore it creates feedback regarding the product and also regarding the process within the sprint.

With steps such as the sprint demo, performance testing, acceptance testing, etc., you generate feedback on the product. When the product owner is trying out the application during

the demo, he will give his feedback. The acceptance tests generate continuous feedback on the acceptance criteria, especially when all criteria are implemented with SpecFlow or any other framework of specification by example.

With steps such as peer review and deployment, you get more feedback on the process: Are the deployment processes correct? Are we coding like we want to? And so on.

The more steps defined in the Definition of Done, the more feedback you will get.

**The second reason for using the Definition of Done is that it improves release planning**

Typically when finishing a sprint, different items are left undone. Some bugs are still in the code, integration testing is not yet done, performance testing in a production-like environment is not done, the manual is not up to date, etc. All this work has to be done at some time. The problem with this undone work is that it piles up every sprint; every feature that is added will let this undone work grow.

What happens in an agile project release planning session is that, based on the number of user stories, points and velocity for a release are planned. For example, when the team has a velocity of 6 and 23 user story points need to be implemented, the release date can take place after 4 sprints.

The problem is that after 4 sprints, this undone work is still there. Many teams solve this by introducing so-called "hardening sprints" or "release sprints." These sprints are used, for example, to create the deployment packages, solve some last bugs, do some final testing, and so on — everything to make the software ready for production.

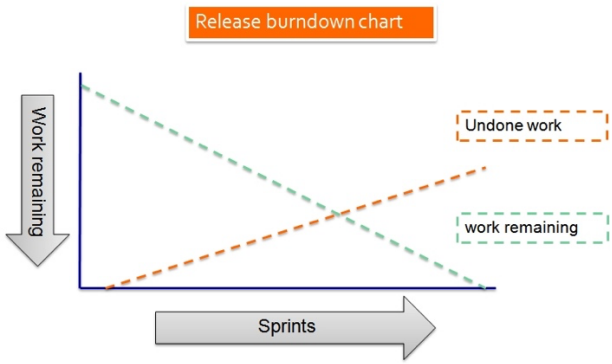
The problem with these release sprints is that they are a bad agile practice. You are trying to timebox work that is unknown (last-minute tests can reveal all kinds of bugs), unplanned, not estimated but still it is really necessary — and it all has to be done in a fixed amount of time and before the release date.

In addition, your release date doesn't match with your release planning. Instead of planning the release based on sprints defined in the release planning, it is now based on "planned" sprints plus one or more extra release sprints.

When the team defines a complete Definition of Done and applies it, all the undone work is done within the sprints and no release sprints are necessary.

**Applying a complete or ideal Definition of Done also gives sense to burndown charts**

A burndown chart shows the amount of work still to be done in the progress of time represented by the green line.




This burndown chart is well visible in most teams but will give a false indication of when the software is production-ready. When the Definition of Done is not well applied, undone work

will pile up in every sprint, represented by the orange line, and this line is usually not visible in regular burndown charts.

The line showing the ideal burndown line plus the undone work line represent the real burndown chart, but this is usually not shown, and the product owner is caught by surprise to realize, after four sprints, that there is still work to be done even though the burndown chart showed differently.

When no release sprints are used, the delta of the black line and the green line shows the risk that is delayed. If the team doesn't pick up this work during a sprint, it will reveal itself in production. For example, when no performance testing is done during the sprints, there is a chance that later an issue regarding performance can occur in production.

**Almost done is not done at all**

A typical conversation that most developers will recognize goes like this:

***Product owner (PO):*** Is the software done?

***Developer (Dev):*** Yes, almost.

***PO***: Can we go to production?

***Dev***: No, not yet.

***PO:*** Why not?

***Dev***: Well, some bugs have to be solved, some integration tests still have to be run, release packages have to be updated, etc.

***PO***: When can we go to production ?

***Dev***: I don't know...

To avoid these kinds of discussions, there should be a common understanding of what is meant by "done" software. A Definition of Done will create more transparency about what the team is doing in every sprint, and what is delivered. When, for example, the Definition of Done doesn't say anything about performance testing in a production-like environment because the organization is not fit to accomplish this in every sprint, then the product owner is aware of this.

**The Definition of Done minimizes the delay of risk**

When the Definition of Done is complete, with all the steps necessary to deliver an increment with the best quality, you are minimizing the delay of risk. All steps in the Definition of Done are subjected to feedback and therefore risky items are inspected, adapted, and improved as much as possible in an early stage, and as many times as there are sprints. In other words, risks are covered several times in early stages of the project.

The smaller the Definition of Done, the more undone work is likely to pile up after every sprint. This undone work is not subjected to feedback but will reveal itself somewhere, sometime, in production.

A complete Definition of Done will minimize this undone work and therefore minimize the delay of risk.

**The Definition of Done represents the agility, quality, and maturity of the team**

A team is able to complete a (new) feature in one sprint and release it immediately to production with all steps defined in the Definition of Done necessary to guarantee the best quality.

The agility of the team shows in the fact that it can release a feature to production in every sprint. The quality of the team is represented by the number of steps in the Definition of Done that are applied when releasing this feature to production.

**How to put the Definition of Done into practice**

Start off by defining two versions of the Definition of Done: one current, and one ideal. The possible reasons to need two versions are competence and maturity.

Competence is a real reason, because not every team is capable of doing everything in one sprint in order to deliver a production-ready product. This is especially true at the beginning of a project. To deliver a finished increment in one sprint, you need to automate many steps in the Definition of Done. For example, automate build processes, automate tests, automate deployment, maybe automate some documentation, etc. This can be complex and time-consuming to set up.

Maturity is another reason why the Definition of Done is perhaps not yet ideal. Some teams are just not ready enough to want to do all the steps in one sprint. They feel it's better to do the regression tests only at the end of all the sprints, or to update the manual just before going to production, because they feel it isn't necessary or takes too much time to do this in every sprint. Those teams don't have an agile mindset yet.

The ideal Definition of Done defines all steps necessary to deliver a finished increment from development to deployment in production. No further work is needed.

The current Definition of Done defines the steps the team is currently capable of doing in one sprint.

It's best to make both visible on the wall so that what the team is delivering in the sprint is transparent to the product owner, and to create a common understanding of what is done. What's important to understand is that the product owner is also responsible if the team is not using an ideal Definition of Done. He can decide that performance testing is not needed every sprint because it has never been an issue on the much faster production servers, and because the team hasn't yet automated performance testing, so it takes too much time to do it every sprint. With this decision, the product owner consciously delays the (potential) risk of having a performance issue in production.

If the product owner wants to have more steps in the current Definition of Done – for example, automated acceptance tests – he should make it a priority that a framework is created that facilitates the automation of these tests. This can be done by giving the work item containing this framework a higher priority in the product backlog.

So, putting two versions on the wall will create transparency for the product owner. It represents the current capability of the team and shows what could be improved. The team can try to regularly expand the current Definition of Done with steps from the ideal Definition of Done. Expanding the Definition of Done will actually mean growing in quality and maturity.

**Conclusion** A good Definition of Done will help with:-

- Getting feedback and improving your product and process
- Better release planning
- Giving burndown charts sense
- Minimizing the delay of risk
- Improving team quality and agility
- Creating transparency for stakeholders

## Gherkin Reference

Gherkin uses a set of special keywords to give structure and meaning to executable specifications. Each keyword is translated to many spoken languages; in this reference we'll use English.

Most lines in a Gherkin document start with one of the keywords.

Comments are only permitted at the start of a new line, anywhere in the feature file.

They begin with zero or more spaces, followed by a hash sign (#) and some text.

Block comments are currently not supported by Gherkin.

Either spaces or tabs may be used for indentation. The recommended indentation level is two spaces. Here is an example:

```
Feature: Guess the word
  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

The trailing portion (after the keyword) of each step is matched to a code block, called a step definition.

Please note that some keywords are followed by a colon (:) and some are not. If you add a colon after a keyword that should not be followed by one, your test(s) will be ignored.

### Keywords

Each line that isn't a blank line has to start with a Gherkin keyword, followed by any text you like. The only exceptions are the free-form descriptions placed underneath Example/Scenario, Background, Scenario Outline and Rule lines.

The primary keywords are:

- **Feature**
- **Rule** (as of Gherkin 6)
- **Example** (or Scenario)
- **Given**, **When**, **Then**, **And**, **But** for steps (or *)
- **Background**
- **Scenario Outline** (or **Scenario Template**)
- **Examples** (or **Scenarios**)

There are a few secondary keywords as well:

- **"""** (Doc Strings)
- **|** (Data Tables)
- **@** (Tags)
- **#** (Comments)

Localisation: Gherkin is localised for many spoken languages; each has their own localised equivalent of these keywords.

**Feature**

The purpose of the **Feature** keyword is to provide a high-level description of a software feature, and to group related scenarios

The first primary keyword in a Gherkin document must always be **Feature**, followed by a "**:"** and a short text that describes the feature.

You can add free-form text underneath **Feature** to *add* more description.

These description lines are ignored by Cucumber at runtime, but are available for reporting (they are included by reporting tools like the official HTML formatter).

***Feature: Guess the word***

The word guess game is a turn-based game for two players.

The Maker makes a word for the Breaker to guess. The game is over when the Breaker guesses the Maker's word.

*Example: Maker starts a game*

The name and the optional description have no special meaning to Cucumber. Their purpose is to provide a place for you to document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

The free format description for **Feature** ends when you start a line with the keyword **Background**, **Rule**, **Example** or **Scenario Outline** (or their alias keywords).

You can place tags above Feature to group related features, independent of your file and directory structure.

You can only have a single **Feature** in a **.feature** file.

**Descriptions**

Free-form descriptions (as described above for **Feature**) can also be placed underneath **Example/Scenario**, **Background**, **Scenario Outline** and **Rule**.

You can write anything you like, as long as no line starts with a keyword.

Descriptions can be in the form of Markdown - formatters including the official HTML formatter support this.

**Rule**

The (optional) **Rule** keyword has been part of Gherkin since v6.

*Cucumber Support for Rule:* The **Rule** keyword is still pretty new. It has been ported in a lot of Cucumber implementation already. Yet if you encounter issues, check the documentation of your Cucumber implementation to make sure it supports it.

The purpose of the **Rule** keyword is to represent one business rule that should be implemented. It provides additional information for a feature. A Rule is used to group together several scenarios that belong to this business rule. A Rule should contain one or more scenarios that illustrate the particular rule.

**For example:**
# -- FILE: features/gherkin.rule_example.feature
**Feature**: Highlander
**Rule**: There can be only One
**Example**: Only One -- More than one alive
    Given there are 3 ninjas
    And there are more than one ninja alive
    When 2 ninjas meet, they will fight
    Then one ninja dies (but not me)
    And there is one ninja less alive
**Example**: Only One -- One alive
    Given there is only 1 ninja alive
    Then he (or she) will live forever ;-)
**Rule**: There can be Two (in some cases)
**Example**: Two -- Dead and Reborn as Phoenix

This is a concrete example that illustrates a business rule. It consists of a list of *steps*.

The keyword **Scenario** is a synonym of the keyword **Example**.

You can have as many steps as you like, but we recommend 3-5 steps per example. Having too many steps will cause the example to lose its expressive power as a specification and documentation.

In addition to being a specification and documentation, an example is also a test. As a whole, your examples are an executable specification of the system.

*Examples follow this same pattern:*

- Describe an initial context (Given steps)
- -Describe an event (When steps)
- Describe an expected outcome (Then steps)

**Steps**

Each step starts with **Given**, **When**, **Then**, **And**, or **But**.

Cucumber executes each step in a scenario one at a time, in the sequence you've written them in. When Cucumber tries to execute a step, it looks for a matching step definition to execute.

Keywords are not taken into account when looking for a step definition. This means you cannot have a Given, When, Then, And or But step with the same text as another step. Cucumber considers the following steps duplicates:
*Given there is money in my account*
*Then there is money in my account*

This might seem like a limitation, but it forces you to come up with a less ambiguous, more clear domain language:
*Given my account has a balance of £430*
*Then my account should have a balance of £430*

**Given**

**Given** steps are used to describe the initial context of the system - the scene of the scenario. It is typically something that happened in the past.

When Cucumber executes a **Given** step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of **Given** steps is to put the system in a known state before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in Given's. If you were creating use cases, Given's would be your preconditions.

It's okay to have several **Given** steps (use **And** or **But** for number 2 and upwards to make it more readable).

***Examples:***
*Mickey and Minnie have started a game*
*I am logged in*
*Joe has a balance of £42*

**When**

**When** steps are used to describe an event, or an action. This can be a person interacting with the system, or it can be an event triggered by another system.

*Examples*:
*Guess a word*
*Invite a friend*
*Withdraw money*
*Imagine it's 1922*

Most software does something people could do manually (just not as efficiently).

Try hard to come up with examples that don't make any assumptions about technology or user interface. Imagine it's 1922, when there were no computers.

Implementation details should be hidden in the step definitions.

**Then**

**Then** steps are used to describe an expected outcome, or result.

The step definition of a Then step should use an assertion to compare the actual outcome (what the system actually does) to the expected outcome (what the step says the system is supposed to do). An outcome should be on an observable output. That is, something that comes out of the system (report, user interface, message), and not a behaviour deeply buried inside the system (like a record in a database).

**Examples:**

- See that the guessed word was wrong
- Receive an invitation
- Card should be swallowed

While it might be tempting to implement Then steps to look in the database - resist that temptation!

You should only verify an outcome that is observable for the user (or external system), and changes to a database are usually not.

**And, But**

If you have successive **Given's** or **Then's**, you could write:

***Example:*** Multiple Givens
*Given one thing*
*Given another thing*
*Given yet another thing*
*When I open my eyes*
*Then I should see something*
*Then I shouldn't see something else*

Or, you could make the example more fluidly structured by replacing the successive Given's or Then's with And's and But's:

***Example***: Multiple Givens
*Given one thing*
*And another thing*
*And yet another thing*
*When I open my eyes*
*Then I should see something*
*But I shouldn't see something else*

***"*"***

Gherkin also supports using an asterisk (*) in place of any of the normal step keywords. This can be helpful when you have some steps that are effectively a list of things, so you can express it more like bullet points where otherwise the natural language of And etc might not read so elegantly.

***For example:***
**Scenario**: All done
*Given I am out shopping*
*And I have eggs*
*And I have milk*
*And I have butter*

*When I check my list*
*Then I don't need anything*

***Could be expressed as:***
*Scenario: All done*
*Given I am out shopping*
** I have eggs*
** I have milk*
** I have butter*
*When I check my list*
*Then I don't need anything*

**Background**

Occasionally you'll find yourself repeating the same **Given** steps in all of the scenarios in a **Feature**.

Since it is repeated in every scenario, this is an indication that those steps are not essential to describe the scenarios; they are incidental details. You can literally move such Given steps to the background, by grouping them under a Background section.

A **Background** allows you to add some context to the scenarios that follow it. It can contain one or more Given steps, which are run before each scenario, but after any Before hooks.

A **Background** is placed before the first Scenario/Example, at the same level of indentation.

***For example:***
*Feature: Multiple site support*
*Only blog owners can post to a blog, except administrators,*
*who can post to all blogs.*

*Background:*
*Given a global administrator named "Greg"*
*And a blog named "Greg's anti-tax rants"*
*And a customer named "Dr. Bill"*
*And a blog named "Expensive Therapy" owned by "Dr. Bill"*

*Scenario: Dr. Bill posts to his own blog*
*Given I am logged in as Dr. Bill*
*When I try to post to "Expensive Therapy"*
*Then I should see "Your article was published."*

*Scenario: Dr. Bill tries to post to somebody else's blog, and fails*
*Given I am logged in as Dr. Bill*
*When I try to post to "Greg's anti-tax rants"*
*Then I should see "Hey! That's not your blog!"*

*Scenario: Greg posts to a client's blog*
*Given I am logged in as Greg*
*When I try to post to "Expensive Therapy"*
*Then I should see "Your article was published."*

***Background is also supported at the Rule level, for example:***

*Feature: Overdue tasks*
*Let users know when tasks are overdue, even when using other features of the app*

*Rule: Users are notified about overdue tasks on first use of the day*
*Background:*
*Given I have overdue tasks*

*Example: First use of the day*
*Given I last used the app yesterday*
*When I use the app*
*Then I am notified about overdue tasks*

*Example: Already used today*
*Given I last used the app earlier today*
*When I use the app*
*Then I am not notified about overdue tasks*

You can only have one set of **Background** steps per **Feature** or **Rule**. If you need different **Background** steps for different scenarios, consider breaking up your set of scenarios into more **Rules** or more **Features**.

For a less explicit alternative to Background, check out conditional hooks.

**Tips for using Background**

- Don't use Background to set up complicated states, unless that state is actually something the client needs to know.
  *For example, if the user and site names don't matter to the client, use a higher-level step such as Given I am logged in as a site owner.*
- Keep your Background section short.
  *The client needs to actually remember this stuff when reading the scenarios. If the Background is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.*
- Make your Background section vivid.
  *Use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like "User A", "User B", "Site 1", and so on.*
- Keep your scenarios short, and don't have too many.
  *If the Background section has scrolled off the screen, the reader no longer has a full overview of what's happening. Think about using higher-level steps, or splitting the *.feature file.*

**Scenario Outline**

The Scenario Outline keyword can be used to run the same Scenario multiple times, with different combinations of values.

The keyword Scenario Template is a synonym of the keyword Scenario Outline.

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

*Scenario: eat 5 out of 12*
*Given there are 12 cucumbers*
*When I eat 5 cucumbers*
*Then I should have 7 cucumbers*

*Scenario: eat 5 out of 20*
*Given there are 20 cucumbers*
*When I eat 5 cucumbers*
*Then I should have 15 cucumbers*
*We can collapse these two similar scenarios into a Scenario Outline.*

Scenario outlines allow us to more concisely express these scenarios through the use of a template with < >-delimited parameters:

*Scenario Outline: eating*
*Given there are <start> cucumbers*
*When I eat <eat> cucumbers*
*Then I should have <left> cucumbers*

*Examples:*
*| start | eat | left |*
*| 12 | 5 | 7 |*
*| 20 | 5 | 15 |*

**Examples**

A **Scenario Outline** must contain one or more **Examples** (or **Scenarios**) section(s). Its steps are interpreted as a template which is never directly run. Instead, the Scenario Outline is run once for each row in the Examples section beneath it (not counting the first header row).

The steps can use <> delimited parameters that reference headers in the examples table. Cucumber will replace these parameters with values from the table before it tries to match the step against a step definition.

You can also use parameters in multiline step arguments.

**Step Arguments**

In some cases you might want to pass more data to a step than fits on a single line. For this purpose Gherkin has Doc Strings and Data Tables.

**Doc Strings**

**Doc Strings** are handy for passing a larger piece of text to a step definition.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:

```
Given a blog post named "Random" with Markdown body
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
```

In your step definition, there's no need to find this text and match it in your pattern. It will automatically be passed as the last argument in the step definition.

Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be dedented according to the opening """. Indentation beyond the column of the opening """ will therefore be preserved.

Doc strings also support using three backticks as the delimiter:

```
Given a blog post named "Random" with Markdown body
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
```

This might be familiar for those used to writing with Markdown.

Tool support for backticks

Whilst all current versions of Cucumber support backticks as the delimiter, many tools like text editors don't (yet).

It's possible to annotate the DocString with the type of content it contains. You specify the content type after the triple quote, as follows:

```
Given a blog post named "Random" with Markdown body
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
```

Tool support for content types

Whilst all current versions of Cucumber support content types as the delimiter, many tools like text editors don't (yet).

**Data Tables**

Data Tables are handy for passing a list of values to a step definition:

*Given the following users exist:*

```
| name   | email            | twitter       |
| Aslak  | aslak@cucumber.io  | @aslak_hellesoy |
| Julien | julien@cucumber.io | @jbpros       |
| Matt   | matt@cucumber.io   | @mattwynne    |
```

Just like Doc Strings, Data Tables will be passed to the step definition as the last argument.

**Table Cell Escaping**

If you want to use a newline character in a table cell, you can write this as \n. If you need a | as part of the cell, you can escape it as \|. And finally, if you need a \, you can escape that with \\.

**Data Table API**

Cucumber provides a rich API for manipulating tables from within step definitions. See the Data Table API reference reference for more details.

**Spoken Languages**

The language you choose for Gherkin should be the same language your users and domain experts use when they talk about the domain. Translating between two languages should be avoided.

This is why Gherkin has been translated to over 70 languages .

Here is a Gherkin scenario written in Norwegian:

```
# language: no
Funksjonalitet: Gjett et ord

  Eksempel: Ordmaker starter et spill
    Når Ordmaker starter et spill
    Så må Ordmaker vente på at Gjetter blir med

  Eksempel: Gjetter blir med
    Gitt at Ordmaker har startet et spill med ordet "bløtt"
    Når Gjetter blir med på Ordmakers spill
    Så må Gjetter gjette et ord på 5 bokstaver
```

*A # language: header on the first line of a feature file tells Cucumber what spoken language to use - for example # language: fr for French. If you omit this header, Cucumber will default to English (en).*

Some Cucumber implementations also let you set the default language in the configuration, so you don’t need to place the # language header in every file.