Ultimate Guide to Agile Metrics

Metrics is one of the most discussed and misunderstood topics in the whole field of Agile Software Development. I have written some material on Agile metrics before, but this is the definitive guide. It will cover every possible metrics you would possibly want to use, and tell you what they mean, when you should use them, how you should use them, and when you shouldn't use them.

I have broken them up into five categories:

- **Metrics from Agile Project Tools** - these track the items your team are working on
- **Lean Kanban metrics** - these might come from Agile project tools or from other places
- **Metrics from Source Control tools** - these track the commits that developers make to a codebase
- **Metrics from Continuous Integration (CI) and Continuous Delivery (CD) Tools** - these track the frequency and quality of builds and automated tests and deployments
- **Business Intelligence Metrics** - these tell you how customers are using your software, how your business is growing, what customers think of your products, and so on.

For each metrics, I will explain what it is, how to collect, when to use it, and things to keep in mind. I will also include some bonus material on ways to visualise or describe these metrics, such as burndown charts or Cumulative Flow Diagrams (these are not metrics in themselves, despite what a lot of people might try and tell you).

Principles of using Agile Metrics

Before we dive into the agile metrics, it is important to understand some fundamental principles on what these metrics mean and how to use them. I am going to suggest some ideas that may surprise you, or go against what is usually expected. Nevertheless this is a crucial first step and you will get a lot more benefit out of these metrics if you first have a proper understanding of the overall context and purpose of metrics.

Principle #1: Metrics should be used by the team

This sounds like an obvious one, but it isn't. In fact, 99% of the time people are using agile metrics, they're not following this rule. Most of the time, someone outside the team, usually a manager, wants to "spy on" or "measure" or "assess" the team, usually for "checking their productivity". This is a poor pattern for a number of reasons:

- These metrics don't measure productivity
- There isn't an easy way to measure productivity
- Productivity isn't the primary measure of success - delivering valuable software is instead (I'll take an unproductive team who delivers valuable software over a productive one that isn't delivering software or is delivering useless software any day of the week)
- Most of these metrics are supposed to be starting points for conversations, and if "managers" are reading these metrics in some kind of report, then they're probably not being involved in those conversations
- Most of these metrics have meaning because of the story and context around them, and the only people who have experienced and will understand that context are the people in the team.

So the team should be the ones collecting the metrics, and the team should be the ones sharing and using the metrics. Try to avoid these metrics leaving the team and going to outsiders as much as possible. The last thing you want is some random middle manager wanting to know why team X has a velocity of 40 and team Y has a velocity of 50. That is not a productive conversation.

Principle #2: Metrics need to be surrounded by conversations

Numbers are important and can help tell a story, but they need to be part of an actual conversation (not an email trail or a spreadsheet, an actual conversation between actual people). Just shoving numbers into a presentation will not tell a story. If you want to tell a story, weave the numbers into a conversation about how you collected them, when, where, and why. And what you plan on doing with those numbers when you get them (or what you have done with them now you have them).

Principle #3: Metrics should be used as part of a specific investigation or experiment

As you will see by reading this guide, there are a lot of metrics that you can capture when doing software development. If you just collect them all, hoping to understand everything at once, you will get overwhelmed and not get anywhere. Each metric tells a certain story and can be used as part of a specific investigation ("what are we having problems with?"), or experiment ("how can we improve that?"). Ideally, this investigation or experiment will come out of a retrospective or similar activity. For example, a team notices that they are having trouble meeting sprint goals. They decide to investigate their cycle time, and find out that it has gone up. They then decide to an experiment to reduce the number of code and design reviews, and measure their cycle time again to see if it has improved. That is a perfect example of using metrics around a conversation, an investigation and an experiment.

Metrics from Agile Project tools

Velocity

This is the first one everyone thinks of when it comes to Agile metrics. It is probably the most overused and overrated of them all as well.

How to calculate velocity

Velocity is quite simple to calculate. At the end of a sprint, add up the total of all the story points that were moved to "Done" (based on your Definition of Done) in that sprint. Some things to keep in mind:

What matters is when the story was finished, not when it was started. A story that started in sprint 11 and was then finished in sprint 12 contributes its full points to the velocity of sprint 12, and none to sprint 11. Ideally your stories should all be finished within a sprint anyway, but its important to be consistent here with this rule.

The points are contributed to the sprint when the story is Done, not when it is released to customers. A stakeholder or the team might decide to park the work and not release it to customers for weeks or even months (though this is not ideal and constitutes waste). The points should still be tracked though, because velocity is a measure of how much work the team can complete in a given period of time, not how often software is released to customers.

What velocity is and is not

Velocity is not a measure of effectiveness, efficiency, competency, or anything else. It is simply a measure of the rate at which a given amount of problem statements are turned into tested software. There are hundreds or thousands of reasons why a team might have a certain velocity in a sprint, and 99% of them have nothing to do with how skilled or experienced the team are. Velocity should never be used to compare the "performance" of a team. It has one purpose only: to provide the team with a benchmark for estimating and planning how much work it can get done over time. If a manager outside the team asks them what the team's velocity is, ask them why they want to know. If the answer is because they want to compare how effective teams are, don't tell them anything (and consider start looking for a new job). If the answer is because they want to know when some work will be completed, tell them when you think they work will be completed (they don't need to know the velocity).

The other important thing to remember about velocity is that it has nothing to do with quality (which is very important) or value (which is even more important, and not the same thing, though people sometimes get them mixed up). Quality of software is the extent to which it matches the expectations of the people who designed it - to what extent it matches the functional and non-functional tests or benchmarks or acceptance criteria with which it is described. Value is the extent to which it enriches the lives of the people who use it. Velocity is based on neither: it is simply how much "stuff" has been moved from one side of a board to the other. It might not be of good quality (though if you have half decent tests and acceptance criteria, it should be pretty decent), and it might be of poor or no or negative value (most software is, sadly).

This is another reason that people should not use velocity to judge how valuable or effective a team is. Firstly, a team can fudge velocity by continually bumping up their estimates (and nobody can stop them doing that, since the team decides their own estimates). Secondly, the numbers they come up with are an estimate of the difficulty and complexity of the user

stories, not their value. I would rather be on a team delivering 10 points of quality valuable software each sprint, than a team delivering 50 points of garbage software that nobody wants or cares about.

How velocity is used

I generally use a three sprint rolling average for velocity. I take the average of the last three completed sprints and use that as an estimate of the team's velocity for future sprints. This can allow the team to predict how much they can deliver over the next few sprints. Keep in mind that due to changes in the team, environment, product context and so on, velocity can constantly vary. So you cannot use your current (probably rolling) average more than a few sprints ahead. The velocity metric can also be an input into a release burnup or burndown chart to visualise progress towards a release milestone.

In some cases velocity is not actually needed anyway to do this type of planning (see Story Throughput below). If the only reason you are putting story points on stories is so you can use velocity to do forward planning, consider dropping the estimates and moving to story throughput instead.

Velocity variance and standard deviation

Ideally, teams should display a fairly consistent velocity, ideally with a gradually increasing slope. Major gaps and spikes in velocity are not a good sign. You can check for these by looking at the variance of your velocity. The proper statistical way to do this is by calculating the average (for this I would use a longer rolling average, maybe six sprints). Then, take the difference of each sprint's velocity from the average and square the difference (so every number is positive). Add these squared differences together, and divide by the number of differences. This gives you your variance. If you want to use the standard deviation (a common figure used in statistics), then simply take the square root of the variance. This cancels out the magnitude increase you got by squaring the differences.

An example:

A teams' average velocity is 30 points. Over the last four sprints, the velocity has been 25, 35, 40 and 20. Subtract each number from the average and square the difference. So 30 - 25 is 5, squared is 25. 30 - 35 is -5, squared is 25. 30 - 40 is -10, squared is 100. 30 minus 20 is 10, squared is 100. If you add these numbers up, you get 25+ 25 + 100 + 100 = a variance of 250. This is a high number so you would often take the standard deviation which is about 16. That is a pretty high number for an average of 30, so the team might want to think about how it can get a more consistent velocity. If you want to know more about variance and standard deviations, there are many excellent resources online for learning descriptive statistics.

Velocity predictability

Another interesting metric is the extent to which velocity matches that planned by the team. For example, if the team in sprint planning puts 24 points of work into the sprint, but then delivers 28 (or 20) points of work, that is an interesting data point, especially if it becomes a pattern. Ideally the team should be working at a predictable sustainable pace. Just make sure that this data point is not used as a means of attacking the team for poor estimation ability. There are many factors that can affect the estimation (and the velocity) of a team, and many of them are external.

How to measure predictability

To measure this, simply take the difference between the points of work planned and points of work completed. You can also use it as a proportion of points, to cater for teams with an unusually high or low velocity in general. To do this, convert the number to a percentage by dividing it into velocity and multiplying it by 100.

Example: a team plans 30 points of work in a sprint and delivers 27. They were off by 3 points, or as a percentage, 10% (3 / 30 * 100). The next sprint they planned 32 points of work and delivered 34. They were off by 2 points, or as a percentage, 6.25%. Just make sure that this number stays within the team: it doesn't need to get reported to management.s

Recidivism

This is an interesting metric: the ratio of user stories that bounce back to development after leaving. This is usually due to failing some sort of QA test (although it could be for other reasons, maybe requirements changed or something similar). You can calculate it by taking the total number of completed user stories in a sprint that entered development for a second time, and then dividing it into the total number of completed stories. If a story went into development more than once, just count it once (though that is not a good sign). You want to get recidivism to as low a number as possible, ideally 0%. If it is

higher than 10% or 20% that is cause for significant concern and probably indicates a quality problem (it could be code quality or requirements quality or test quality or even data or environment quality, so don't go straight to the developer and start asking questions - try to figure out the root cause).

First time pass rate

This is similar to recidivism. It is the percentage of test cases that pass the first time they are run. Simply take the number of test cases that failed at least once, and divide it by the total number of test cases. This can be by sprint or by release, each can be considered a separate metric. This measurement is usually done for progression test cases, I.e. new feature work, though you may want to measure it also for regression tests I.e. testing old features to make sure the haven't broken. If you do, keep the progression and regression metrics separate, since the tell a different story. First time pass rate should ideally be close to 100% - even for teams without extensive automation, there should be some reasonable level of quality built in. If it is below 90%, there is a quality problem (though again, try to find out the part of the process that is a cause of problems, rather than go attacking people). If you implement reasonable test automation then getting close to 100% should be quite easy.

If you are not using test cases, then you can just base it on user stories. In which case, take the number of completed user stories that failed one or more tests in a QA, ST or SIT state and divide them by the number of completed stories in that period. Again, you can do this separately by sprint or by release. User stories should only be for progress testing - do not create user stories for regression tests. That is an anti-pattern (user stories represent incremental work to build a product, not retesting of that product).

Defect count by sprint

This is a simple metric that lets you see how many defects you are raising each sprint. You calculate by adding the number of defects that were created at some point during the sprint. If a defect somehow got created twice (that shouldn't happen), just count it once. If it got closed in a different sprint (that shouldn't happen either), don't worry about that. It is just a count of defects created. This is a number that should obviously be going down, ideally at or close to zero.

Defect count by story count

This is probably a more useful metric than the previous one. It is the ratio of the previous number (defects created during the sprint) to the total number of user stories in the sprint. The problem with using a flat number like defect count per sprint is that it doesn't take into account how many stories there are in a sprint. This can vary a lot depending on the team size and experience, project context, amount of current defects and technical debt, etc. Two defects raised in a sprint with two stories is very different to two defects raised in a sprint with 20 stories (using this metric it is 50% versus 10% defect rate per story). So to find this, simply divide the number of defects created in a sprint (previous metric) by the number of stories in the sprint. I would do this at the end of the sprint by counting total number of stories that were in development or a later state at that point in time. Don't forget to include closed defects and closed user stories!

An example: say you are at the end of a sprint and you notice you have two open defects and three closed defects. Two of the closed defects were raised a couple of sprints ago and only just got closed now. You have two user stories in the backlog, six in development, two in QA and two that are Done. You start by calculating your defects raised this sprint, which is three (two open, and three closed but one was raised an earlier sprint so it doesn't count). You then add up your stories that are at least in development - which is 10 (4+2+2). So your defect count by story count is 3 in 10 or 0.3 repeater.

Story completion ratio

This is the number of stories completed in a sprint versus the number of stories that were committed. So if a team puts ten stories into a sprint, and completes seven of them, they have a completion ratio of 70%. This is a metric you want to be pretty high - stories are supposed to small and able to completed within one iteration. Make sure to include stories carried over from previous sprints when counting the number of stories committed. Those are effectively part of the goal or commitment of the sprint. And make sure to also include stories that were abandoned during the sprint and put back into the backlog. Ignoring those can make this metric look better than it otherwise would be.

Story point completion ratio

This is similar to the last metric, but you calculate it by using the story point estimates on the stories, not the count of stories. Although I'm not a huge fan of story point estimation, this is a pretty useful metric, and tells a better story than the

story count. Say a team commits to 10 stories, two of which are one point and eight of which are 13 points. If they fail to complete the pair of one point stories, they have failed to deliver 20% of the number of stories in the sprint, but have failed to deliver only 2% of the total story points in the sprint.

How to use this metric

This metric provides a guide to how well the team is forecasting its capacity, I.e. how well it is doing sprint planning. Ideally, the team should be making a reasonable and stable commitment each sprint, and completing most or all of that work. If there is a pattern of large gaps, that can point to several possible problems. The team could be underestimating the size of the stories. They could be put under pressure to take on more work than they think they can deliver (this is a MAJOR red flag; the team should always feel like the can freely choose their sprint goals and commitments). Or they could be making a reasonable forecast, but running into impediments during the sprint that impair their ability to deliver on the work.

How not to use this metric

As always, make sure to attack the problem rather than the person. Use techniques such as Five Whys to perform root cause analysis and discover corrective actions or improvement items. And Make sure that this metric is not shared with management unless the team all agrees and there is a good reason to do so.

Time blocked per work item

These last two metrics describe the impact and duration of impediments on your work items. To calculate time blocked per work item, you first need to be tracking blockers (sometimes known as impediments) in your project tracking tool. Most of the agile tracking tools today allow you to mark stories or tasks as “blocked”, and will have some form of auditing or reporting on the amount of time that an entity was blocked. If you are tracking blockers, then at the end of a sprint, simply add up all the time that any story or task in that sprint spent in a blocked state. Do that for any entity in the sprint (I.e. anything that spent any time in any workflow state in the sprint, including sprint backlog, but not things in the product backlog). Then divide that amount of time by the total number of entities that you counted as candidates for this metric. So if you had 20 stories in the sprint, and there were a total of twelve hours that some of them (in some combination) spent blocked, then your stories are spending 36 minutes blocked on average each (four hours is 240 minutes divided by 20 stories).

How to use this metric

This number tells you how badly your workflow items are on average affected by impediments. You want this number to be low and trending lower. Anything over one hour is quite troubling.

How not to use this metric

Don’t use this metric to criticise the team or find scapegoats. This is a metric the team should use to assess how badly their ability to deliver work is impaired by impediments. Keep in mind that many of these impediments will be external to the team - so don’t let a manager use this metric to attack the team or the people in it. These “blockers” are usually the result of inefficient artifacts in the value stream, such as signoffs, approvals, handovers and so on.

Percent of items blocked

This is similar to the previous metric, but instead measures what proportion of user stories in a sprint are blocked by impediments. It doesn’t tell you for how long, which is a limitation, but it does tell you how many (as a proportion of total) stories are blocked, which can be useful. If you are using time blocked per work item, then your numbers can be skewed by an outlier - one story that is blocked for a long time can blow up the average (which would otherwise be low). You can exclude the outlier, or use this metric instead. Calculating it is simple - you just count the number of stories that became blocked during a sprint (assuming they made it as far as the sprint backlog, don’t bother counting or even “blocking” anything in the product backlog). Then divide it into the total number of stories that were in some state of progress during that sprint. That gives you your percent of items blocked.

How to use this metric

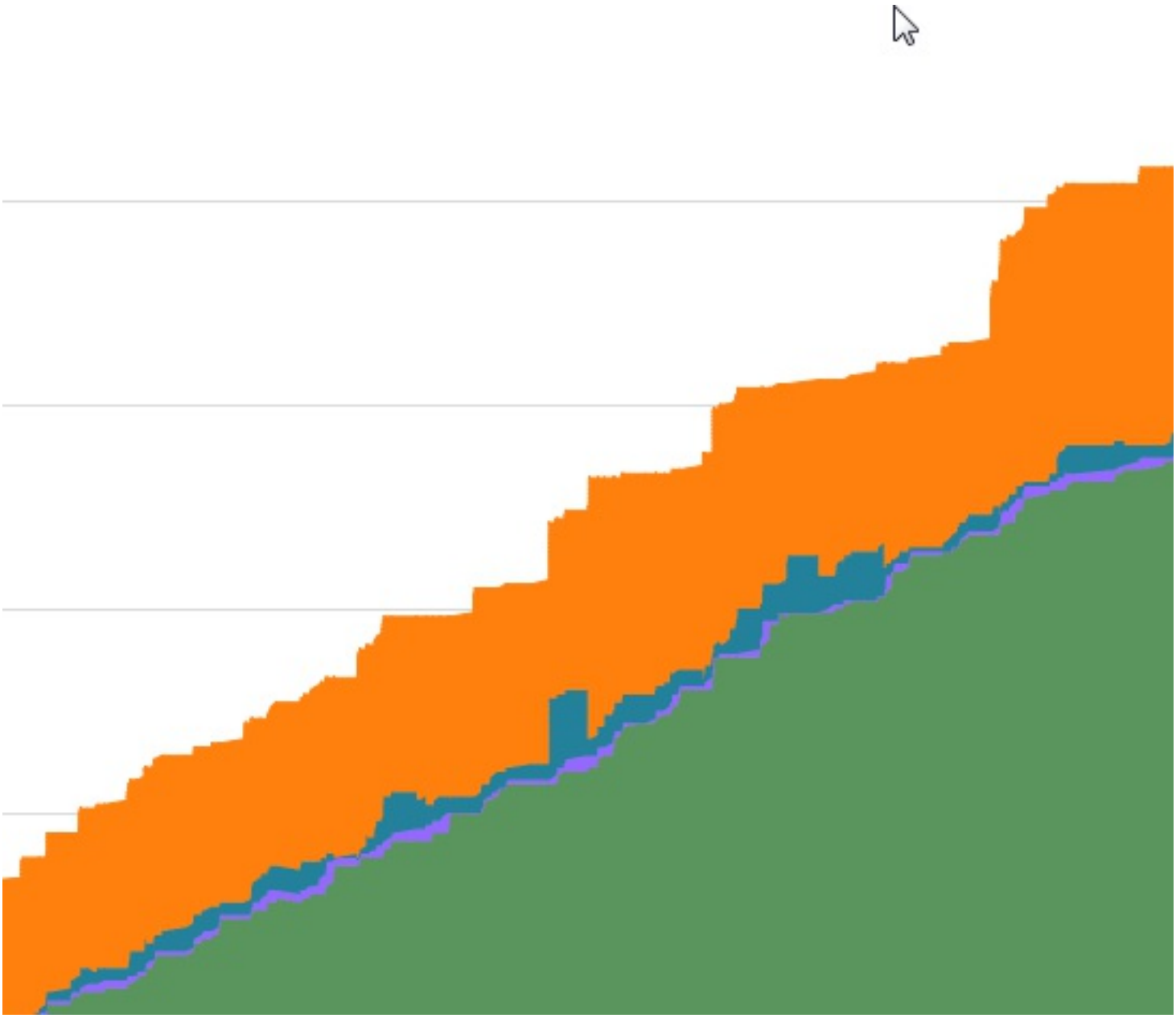
You can use this metric to determine the general frequency of impediments on a team. You want this number to be low, ideally 10% or lower (hopefully close to zero). Like the previous one, make sure it is used to investigate possible root causes and ideas for continuous improvement, not to beat people up - impediments are often from forces outside the team’s

control (otherwise they would just remove them right away).

Cumulative Flow Diagram

What is this metric?

Cumulative Flow Diagrams (CFDs) are not really a metric per se, but a visualisation of a set of underlying metrics. More specifically, a CFD is a stacked line graph showing the total point values of work in various states. It is hard to verbally describe but much easier to explain with a visual example. Below is a typical Cumulative Flow Diagram for a team. The horizontal axis is time, and the vertical axis is stories in a particular state, by story points. The different colours represent different states that those stories are in.



In this diagram, the green sections represents stories that are Done, purple is stories in Review, Blue is stories in progress, and orange is stories in the backlog.

At any point in time, you can draw a line down and see the proportion of work in various states. The fact that you can see all the states and the changes over time makes this a very powerful and sophisticated tool. Much more so than the more famous and popular graphs like burnup or burndown charts (which are quite poor representations of work in a system).

This is quite a good Cumulative Flow Diagram, because the backlog is never too big (you want it to be creeping just ahead of stories in progress), stories are continually being completed, and most importantly, there is a small amount of work in Review and in Progress (I.e. WIP is being kept under control).

Lean / Kanban metrics

This section will describe some metrics that belong to the Lean and Kanban systems of working.

Story Lead time

Lead Time is a concept often used in Lean and Kanban. It is the total elapsed time from the point when a user story enters a system (for example it is written up in a backlog or created in an Agile project tracking tool), until the time it is completed, I.e. meets the Definition of Done. This includes the time spent sitting in a backlog. So the Story Lead Time tells you how long it takes for a request or enhancement to move all the way through a system. You may want to change the criteria of when the story is "Done" to mean actually released to customers, instead of being ready to be released. When calculating velocity, you are using your normal definition of done (which is usually Ready for Production, not actually released to customers) because you are trying to figure out how quickly your team can burn through work and pick up the next story. Lead time is a measurement of the total time it takes for a requirement from when it is first created to when it is realised, I.e. starts earning value. Lead time is useful for figuring out the overall speed of your value chain. You should be trying to reduce lead time (it is in many ways a much more important metric than velocity).

Make sure to use the total elapsed time, even if it includes long periods of waiting. If a story goes in the backlog and sits there for six months, and then is built and delivered in one month, the lead time is seven months! If you're thinking "but that will force us to quickly move stories through the backlog and get them into production", you're starting to understand the importance of lead time.

Story Cycle time

Story Cycle Time is similar to Lead Time, but with an important difference., It is the time it takes for a story to go from being "In progress" (you probably have that status or a similar one on your VMB or in your digital project tracking tool) to Done. It is therefore a subset of Lead Time and is thus always shorter than Lead Time. Lead Time minus Cycle Time is Wait Time: the elapsed time that a story or requirement spends sitting in a backlog, waiting to be actioned. Much like Lead Time, you should be trying to reduce your cycle time. Ideally your average cycle time should be around one half of a sprint or less. If your average cycle time is longer than one sprint, you have a big problem, because you are not finishing stories within one sprint. Much as with Lead Time, make sure to use total elapsed time from when the user story first goes into development, even if it becomes blocked or stuck or spends a long time not really being worked on. If it bounces back and forth between states (e.g. between In Development and In QA for example), make sure to include all that time too. The clock is never restarted; it starts ticking when the story first goes into development and it stops ticking when it is finished. No exceptions.

Feature Lead Time

Feature Lead Time is like Story Lead Time, but for features instead of stories. Since user stories are often bundled together and released as a feature, this is actually a useful metric. It describes how long it takes a body of valuable work to go from an idea to customers' hands. As with user stories, the clock starts ticking on a feature as soon as it enters any sort of backlog. If you want to measure "concept to cash", I.e. how long it takes to get into customers' hands, then the clock stops ticking when it is released to customers. If you want to measure time to get a feature delivered and ready to be released to customers, then the clock stops ticking when it is ready for release (this might be actually deployed but dormant or killswitched for example).

Feature Cycle Time

Like the previous metric, Feature Cycle Time is like Story Cycle Time, but for features instead of stories. It describes how long takes on average for a feature to be built. Like all these four metrics, you want this one to be trending lower. Breaking stories and features into smaller parts will encourage this (and is a good practice to get into). Make sure to include total elapsed time, including waiting and handovers. The measurement is similar to Feature Lead Time but the clock starts when the feature goes into development. You should define this to be when the first user story or task that is part of the feature goes into development.

Story Throughput

This is an excellent and underrated metrics. It is extremely easy to measure, difficult to game, tells a number of important stories, and is an important part of moving away from estimation. It is also very simple: it is the number of stories

completed in each sprint. Similar to velocity, only count stories that meet the Definition of Done (whatever that is for your team), and count them for the sprint the were finished in, not the sprint they were started in.

How to use Throughput

Story throughput doesn't tell you the size of the stories you are completing, but it tells you how many of them you are completing. Here is a very important point: if you stories are all roughly the same size, then you can basically do away with velocity and use story throughput. Here is another important point: if your stories follow a normal distribution (I.e. most of them are around a certain average size, with a roughly equal number above and below that average), then you can also use story throughput to plan your burnup / burndown charts. So you can use throughput even if your stories are NOT all the same size. They just have to an average and a normal distribution, which you will almost certainly have. You can use a long-term average or a shorter one (e.g. three sprint rolling average) to use as a baseline for your story throughput.

Takt Time

Takt time is an unusual metric that is more suited to Lean Manufacturing than it is to Lean Software Development. But some people discuss it and it is an important part of Lean so it is worthwhile at least understanding what it is. Takt time is the average time between customer orders. So if you get 48 customer orders per day, your takt time is 30 minutes (48 is two per hour and there are 24 hours in a day). Takt time is a measure of customer demand so it is the primary metric used to drive the output of a Lean Manufacturing system (which are pull rather than push based systems). Customer orders do not really drive "output" in a software context since software can be copied at essentially zero cost (or delivered over the web as a service for close to zero cost), hence why it is not as useful in a software context.

Created to Finished ratio

This is an interesting one: the ratio of created stories to finished ones. Every sprint, divide the number of stories you created (regardless of where you put them or what state they are in), and divide it by the number of stories that were completed (I.e. that met the relevant Definition of Done). In the early stages of a product, this will probably be a number greater than one. That is because people are coming up with new ideas and writing stories, and developers might be slow to move stories through while they are building scaffolding, agreeing on designs, and so on (though of course they should be looking to release some form of working software each sprint). Over time though, the ratio should start coming down. If you want to actually work through your backlog, this number will obviously need to go below one. There is a useful motto here: "stop starting, start finishing".

How to use this metric

Use this metric if you are worried that too many items are going into the backlog and not enough getting completed. Remember, backlog is a form of inventory and inventory is a form of waste. You want to have enough things in there to keep the team busy, and a high level roadmap, but not much else.

Metrics from Source Control tools

Source control systems are a fundamental part of any software development activity. Everyone uses them, even people working alone on software, because it provides version history and rollbacks. There are roughly two types of source control systems: Centralized Version Control systems (CVS, such as SVN) and Distributed Version Control Systems (DVCS, such as Github). Almost everybody is moving to DVCS, since they are offer some important advantages. I will assume for the purposes of this guide that you are using or can get access to the data in a DVCS. If you are using a CVS, there will be a much smaller set of useful data you will be able to get access to, and they have some other inherent problems, so I would move to a DVCS as soon as you can.

Number and proportion of merged pull requests

One simple metric is simply the number of merged pull requests, though it really makes more sense when seen as a proportion. More specifically, the proportion of merged pull requests to total pull requests. If this number is quite high, that's probably a good thing, though you might want to make sure that there are some meaningful code reviews and discussions still going on. If it's a bit lower, that's ok. PRs don't have to be always merged. Sometimes developers come up with a different or better way of doing things, ideally one that doesn't require code changes at all (source code is a liability, not an asset). If the number is very low, that could be a warning sign that there is a problem between two or more developers. As always, make sure this metric is used as part of a series of conversations, not as a random "report" that is

sent to management without the context of the people and situations involved with the metric.

Duration of open pull requests

This is the average amount of time a pull request stays in an open state. This is one of those interesting "Goldilocks" metrics where you want it to be not too high and not too low, but somewhere in the middle: just right. If pull requests are open for a long time, this can indicate there are a lot of long raging arguments going on over someone's commit. While discussions are a good thing, you don't want them to go on forever: decisions have to be made at some point. If each PR stays open for a week, it will be very hard for the team to deliver a meaningful amount of work. On the other hand, you don't want it to be too low. If every PR is open and then closed five minutes later, that suggests that nobody is really looking at or reviewing it. That could be because the team are not collaborating effectively, or maybe because someone is harassing the team to just "hurry up and get on with it!". Either on is a bad sign and the team will need to investigate the root cause of this.

How to use this metric

This is one to just check in on every now and then. Your DVCS should calculate and show this automatically; if it doesn't, get one that does.

Comments per pull requests

This is similar to the previous one, but instead of measuring the average amount of time a pull request is open, it is the average number of comments on a pull request. Again, you want this to be not too high (indicates that there are some personality issues or conflicts in the team, if someone gets 12 comments every time they raise a PR), but not too low (if every PR gets one or zero comments, nobody is really looking at them). Use this in a similar way and context than you would the duration of open pull requests. Your DVCS should calculate and show this for you automatically.

Code additions and deletions

Your DVCS should feature a graph that shows changes in code (additions and deletions) over time, usually measured in CLOC (Count Lines of Code or Changed Lines of Code). This is a good one to look at regularly. You should not just be looking for the frequency and amounts, but looking for patterns too. Are there lots of spikes on the weekends? People are working overtime, which is usually a bad sign. Are there lots of additions but no deletions? That suggests that nobody is refactoring (since refactoring usually results in a net decrease in the number of lines of code in a system, which is a good thing). Remember, source code is a liability, not an asset. The best system has zero lines of code, not millions.

Metrics from CI and CD tools

These metrics are pulled in from your Continuous Integration and Continuous Delivery tools. Nowadays these generally are part of a holistic DevOps toolchain and automated pipeline.

Test coverage

This is a popular and controversial metric that a lot of people get stuck on. It is the proportion of the codebase that is covered by automated tests. More specifically, it is the proportion of methods that have one or more automated tests (unit or integration) defined for that method. While thinking about automated test coverage is a good idea, you need to be careful with this metric. There are some reasons to be careful about relying on this metric too much:

- It doesn't distinguish between good or bad tests
- It doesn't prevent developers from putting in completely useless tests (just asserting True or something on a test)
- It doesn't (usually) cover end to end tests, since those aren't testing a method but the entry points and user interface components of a system.

A high test coverage score is not necessarily a good thing. If you focus too much on it, and hassle developers into getting the number up, you may encourage poor behaviours such as slapping useless tests on things just to improve the metrics. This is bad for two reasons. Firstly, it is giving you an unrealistic view of your real test coverage. And secondly, it is a wasteful practice and depressing for developers to have to do.

That is not to say that you shouldn't care about test automation. It is important to keep in mind that your test coverage is a complex and nuanced thing that cannot be expressed by one simple number. It should be part of a series of regular conversations backed up by multiple data points and regular code reviews.

A brief note: this metric is actually possible to report without CI or CD tools, since it can be determined by analysis of source control (there are many plugins and tools that can calculate it). I however placed it in this section because it is conceptually closely related to these other CI and CD metrics.

Good versus failed builds

This is the proportion of builds that fail out of all builds. This should be a low number, ideally very low. Developers should be performing builds on their machine (which should have a regularly rebased codebase on it) before checking anything in. If it is above 5%, that's a bad sign.

Escaped defects

This is the number of defects that are found only after they have gone to production. This number is similar to Incidents Raised (which is in the Service Management section below), but not quite. It is possible for a defect to go into production but not to be ever raised as an incident (it might not affect any customers, but it is still a defect). It is also possible have an incident raised without a defect (e.g. a temporary failure of infrastructure). A defect is a problem in the codebase. Escaped defects should be zero or close to zero.

Unsuccessful Deployments

This is simply the number of deployments that fail. You might want to count this per week, month, or year, depending on how often you do do a release. Deployments can fail for a number of reasons, usually to do with mistakes in the deployment tool configuration. You might want to count this only for production, or you might want to include other environments too, such as a staging or test environment. This should be zero or close to zero, especially for production. You should be able to get this number from your DevOps tool without much difficulty.

Mean time between releases

This is simply the average amount of time between the time you do a release. It might sound very simple, but there are some things that you keep some things in mind:

- You should only count deployments to a production environment. Staging doesn't count and test environments definitely don't count. You should of course be deploying to at test environment very frequently (multiple times per day), so if you're not doing that, you have a deeper problem.
- You should only count successful deployments. Failed deployments don't count (see other metric above).
- The software doesn't have to be released to customers; it might be a blue - green deployment, a pilot group or dialup, feature flagged off, and so on: but it still counts as a successful deployment. The decision to release to customers is a different decision (and mainly business rather than a technical decision).

How to measure this metric

This is a simple metric to measure: take the number of deployments within a given interval of time (maybe three months) and divide it by the amount of time periods (e.g. number of days in three months). That is your mean time between releases. Mean is just another word for average here. You can choose any time period you like, but some tips:

- A larger range of time is usually a better sample size, Three months is probably a good starting point, but you can change it if it makes sense to do so.
- Use a rolling range of time: so if you are doing three months, don't take the average every three months, take the average every month of the last three months.
- Don't arbitrarily change the time sample size to make stats look good or bad, whatever you do. Be honest and transparent.

CLOC per release

CLOC is Changed Lines of Code. This is a measurement of the average number of lines of changed code per release. A changed line means a line removed, a line added, or a line modified: those all count as one changed line. So to calculate this metric, you divide the total number of changed lines of code over a given series of releases by the number of releases over that period. It might seem counter-intuitive, but you want this metric to be small and getting smaller. That's because smaller releases have less change, less complexity and less risk. Many small releases are vastly preferable to a few big ones.

Metrics from Business Intelligence tools

Some people might be surprised to see these metrics here. They might think that these are more abstract metrics that belong to "the business", and that a software team should just be focused on traditional things like velocity and defect count. I think that is rubbish! These are actually the most important metrics of all. These are your "outcomes" rather than your "outputs". And the whole team should be aware and thinking about these metrics. The best successes I have ever had with software teams is when the whole team feels a sense of pride and ownership in the things they are building. That they feel connected to the customers and the value they are delivering and the outcomes for the business they are driving.

I have divided the content of this section according to what are known as the "Pirate Metrics", or AARRR (because pirates say "Aarrr!", get it?). That stands for Acquisition, Activation, Revenue, Retention and Referral.

Acquisition

Unique visitors

This is a common metric used for tracking how people come to your website. Make sure the visits are unique (usually done by de-duplicating by IP address). This metric is at the top of the pirate metric "funnel" (I.e. each other metric is a smaller number than this one, usually expressed as a percentage / proportion), and the numbers can often be large. But make sure you do not put too much emphasis on unique visitors. It is referred to by some people in the startup community as a "vanity metric": it is a large number and is often thrown around to impress potential investors or media. But unless your site is behind a paywall, and few are, then this metric is not very meaningful for a business. If it is very small, then you should probably put some marketing or PR efforts into boosting it, but once it starts growing, your efforts are usually better applied to other metrics in the funnel.

Activation

Conversion rate

Conversion rate is a term used a lot in various different contexts. It can often be used to refer to different types of conversions at different points in the funnel: converting website visitors into website subscribers or leads (which is the context I'm referring to here), converting leads into trial customers, converting trial customers into paying customers, and so on. But the concept is basically the same at each stage: what proportion of customers at a higher point in the funnel go on to the next point in the funnel, expressed as a percentage. So if you have 10,000 unique website visitors a month, and 2,000 of them subscribe to your newsletter or download your whitepaper or whatever, then your conversion rate is 20%.

Retention

Monthly Active Users (MAU):

Monthly Active Users is a very common metric, especially for social media applications or SAAS (Software As A Service) providers. It is just the measure of how many users have interacted with your website or app or used your SaaS product in a month. Some say that MAUs are the "lifeblood" of an app or SaaS business. I put this metric under Retention because it indicates not how many people are coming to your product, but how many people are sticking with it. Note that these MAUs might be paying users, or non-paying users, depending on your service's pricing model. Some have a free trial, others have a free tier and a paid tier, some have only a paid tier, and so on.

Revenue

There are different pricing and therefore revenue models, so it is difficult to choose which metrics to put here. There will probably be some specific ones for your particular business and pricing model. I have tried to include some popular and general ones however that you are likely to use or see used in the marketplace.

Conversion rate

We earlier saw conversion rate as an activation metric. There it was talking about the percentage of people at the Acquisition stage of your funnel (e.g. visitors to your website, appstore page, etc.) who then move to the Activation stage: they sign on for a trial, sign up for a newsletter, etc. Here, conversion rate is referring to the percentage of people who have already reached the activation stage of the funnel, who then become paying customers. This method of payment will of course depend on your particular pricing model (one-off, recurring monthly, recurring yearly, maybe a per call cost for an API or something). It is expressed as a percentage (I.e. it is revenue customers divided by activation customers multiplied by 100).

Monthly recurring revenue

This metric (often abbreviated to MRR) is another classic of the SaaS business. It is simply the number of your monthly users (MAU) multiplied by the price per user. If you have different pricing tiers, then you would have to break those down into separate MAUs and mulitply by their respective pricing tiers (or weight the revenue per user by the proportion of your user base at that pricing tier, So if 30% of your users are paying $20 a month and 70% are paying $50 a month, then your weighted revenue per user is 0.3 * 20 + 0.7 * 50 = 41. Multiply that by your number of users and you'll get your MRR.

Throughput

Throughput is a concept from Eli Goldratt's theory of Throughput Accounting, closely related to his Theory of Constraints. If you want to understand the details, I would recommend you read his novel The Goal. To put it simply, and without getting too far into accounting territory, throughput is basically net sales. So you take the revenue per unit, and subtract from it the cost per unit, and then multiply it by the number of units sold. It makes more sense in a manufacturing or retail context where there is a clear cost per unit (cost of raw materials for manufacturing, or cost of goods sold for retail). The important point is not to amortise other operational expenses onto the cost per unit (which most Cost Accounting methodologies such as Activity Based Costing will usually try to do). Selling software does not really have the concept of cost per unit sold (software costs basically nothing to mass produce or distribute, especially with the demise of physical media). So net sales is simple: it is just total revenue, over a particular period. This might sound overly simple or not useful, but there is a profound shift in the Throughput way of thinking that is difficult to explain without a longer explanation and comparison to traditional accounting methods.

Cost of Delay

This metric (often abbreviated to COD) is considered by some (such as Donald Reinertsen) to be the most important metric of all, especially in the context of product development (which much of Agile software development is concerned with). It is simply the cost of not releasing your product, over a given period of time. So for example, if you could release your software today and earn $1000 over the next month, then by not releasing it now and releasing it instead in a month's time, you are missing out on that $1000 of revenue. So your cost of delay for that month period is $1000. How to estimate that revenue lost could be complex and would be highly dependent on your particular product and circumstances, so I cannot provide guidance here. According to Reinertsen in his book "The Principles of Product Development Flow", many decisions around priority of work or resources can and should be converted to Cost of Delay to provide a simple answer. Always remember that it should be described as a cost over a period of time: a week, a month, etc. For that reason, some people call it Cost of Delay Divided by Duration, or CD3. This metric is sometimes used as an input into a prioritisation technique called WSJF (Weighted Shortest Job First).

Referral

Net Promoter Score (NPS)

NPS stands for Net Promoter Score. It is an important business metric that indicates how likely your customers are to refer you to other people. It is a number that ranges from -100 (no customers are likely to refer you to others) to 100 (all of your customers are likely to refer you to others). The calculation is a little complicated. First, you have to conduct research and ask your customers "on a scale of 0 to 10, how likely are you to recommend our products or services to a friend or colleague?". You then divide the responses up into Promoters (who give a score of 9 or 10), Passives (a score of 7 or 8) and Detractors (a score of 6 or below). Then, calculate the percentage of responses that are Promoters and Detractors. Finally, subtract the percentage of Detractors from the percentage of Promoters. So if 50% of your respondents are promoters and 30% are detractors, your NPS is 20 (50 minus 30). NPS is considered very important for the overall growth and success of a business. It is difficult to prove strict causal links, because NPS is based on survey responses that may not be indicative of actual behaviour (and many referrals are word of mouth and thus impossible to track or prove). But studies have shown

that NPS is a very strong predictor of a successful business. Ignore it at your peril.

Referral success rate

Your product or service might have an actual referral system, where you offer customers a bonus or discount in exchange for promoting your company to others. This can be done by a variety of methods, including a discount for a successful referral of a friend, a free month if people share an ad on their facebook feed, and so on. Referral rate is the percentage of referrals that end up in a sale. You could also separately track the proportion of customers who decide to participate in a referral scheme (though that is a vanity metric, since if no prospects take up the offer and turn into customers, it doesn't really matter how many people are sharing the content).

Summary

Hope you enjoyed this long journey into the complex world of Agile metrics. I'm confident I covered every important metric that you would ever want to think about. Please remember that these should be chosen and used wisely. And remember — **not everything that can be counted counts, and not everything that counts can be counted.**