Contents

[Tier I Metrics 2](#_Toc8634964)

[Cycle Time 2](#_Toc8634965)

[Deployment Frequency 2](#_Toc8634966)

[Average Features per Release 2](#_Toc8634967)

[Feature vs Issue Investment 2](#_Toc8634968)

[Mean Time to Recovery 3](#_Toc8634969)

[Mean Time to Detection 3](#_Toc8634970)

[Percentage of Failed Deployments 3](#_Toc8634971)

[Defect Leakage 3](#_Toc8634972)

[Number of Defects Opened per Sprint 4](#_Toc8634973)

[Percent Regression Test Suite Automated 4](#_Toc8634974)

[Percent Test Cases Automated 4](#_Toc8634975)

[Test Strategy Documented 4](#_Toc8634976)

[Tier II Metrics 5](#_Toc8634977)

[Next/Last Release 5](#_Toc8634978)

[Application Telemetry Data 5](#_Toc8634979)

[Features in Production 5](#_Toc8634980)

[User Feature Adoption (Feature Debt) 5](#_Toc8634981)

[Frequency of Pull Requests 6](#_Toc8634982)

[Frequency of Production Defects 6](#_Toc8634983)

[Code Coverage 6](#_Toc8634984)

[Test Coverage 6](#_Toc8634985)

[Team Velocity 7](#_Toc8634986)

[Backlog Health 7](#_Toc8634987)

[Number of Test Cases per Requirement/Story 7](#_Toc8634988)

[Number of Requirements/Stories without Test Cases 7](#_Toc8634989)

[Number of Test Executions per Sprint 7](#_Toc8634990)

[Requirement/Story Card Test Coverage in Sprint 8](#_Toc8634991)

[Number of Test Cases in Regression Test Suite 8](#_Toc8634992)

[Percentage of Regression Test Run per Sprint 8](#_Toc8634993)

[Percent Story Card Unit Test Automated 8](#_Toc8634994)

[Test Strategy Reviewed and Approved 8](#_Toc8634995)

[Test Executions outside Sprint 8](#_Toc8634996)

[Number of Test Cases Developed in Sprint 9](#_Toc8634997)

[Number of Defects Opened 9](#_Toc8634998)

[Number of Defects Closed 9](#_Toc8634999)

[Number of Defects Open 9](#_Toc8635000)

[Links 9](#_Toc8635001)

# Tier I Metrics

## Value Delivery Rate

* **Description** – The accumulated value of features and defects delivered into production minus team cost. This measures the value delivered to the organization.
* **How to Measure**
* **Benefits –** Shows how successful the development effort is.
* **Disadvantages –** requires teams to know the value of what they are delivering. Places burden on product manages and POs to quantify value of backlog items and not just represent work.

## Cycle Time

* **Description** – This metric is the amount of time it takes for a feature that is ready to be worked on to the point where it is deployed to an environment where users can start interacting with it. Wait time represents all the time between handoffs, process time being active development, and queue time being the amount of time where it’s waiting to be released.
* **How to Measure**
* **Benefits –** Features get out in front of users quicker and start generating value as soon as they’re ready. Any time a feature spends not being deployed when it’s ready is lost value.

## Deployment Frequency

* **Description** – How often is the team deploying to its different environments. Ideally over time the frequency should increase as teams are doing more frequent, smaller releases.
* **How to Measure**
* **How it can be abused** – Features need to be deployed within deployments. Merely pushing out to environments is useless without new features.
* **Benefits—**A higher deployment frequency helps get your features out quicker so they can start generating value. It also leads to smaller releases, which in turn reduces the risk factor in your deployments.

## Average Features per Release

* **Description**—The average number of features released over time. This should decrease over time, eventually reaching the point where a team is able to release individual features as soon as they’re ready.
* **How to Measure**
* **How it can be abused** – This needs to work with deployment frequency to be of any use. With this, teams should get to the point where they’re deploying a minimal amount of features in a deployment. However, if those deployments aren’t done frequently, then the team has lost the value of smaller amounts of features per release.
* **Benefits –** Similar to deployment frequency, a smaller amount of features in a release will reduce the complexity of the release and lead to a smaller risk factor.

## Feature vs Issue Investment

* **Description** – Measures how much effort teams are spending in an iteration on developing new features versus how much time is being spent fixing issues that have been introduced. DevOps practices should help to eliminate the amount of issues being reduced so teams should see the amount of effort being spent on issues drastically decrease.
* **How to Measure**
* **How it can be abused** – If teams are not putting the proper effort points on their features and issues then this number can be easily manufactured. Teams need to stay honest during the estimation process to make sure they’re not artificially inflating this number.
* **Benefits –** More time being spent on delivering features instead of fixing issues leads to more value being delivered and generated.

## Mean Time to Recovery

* **Description** – The average amount of time it takes a team from when an issue is first reported to how long it takes to properly resolve the issue. DevOps practices typically introduce automation around deployments which drastically reduces the amount of overhead required to push out a fix for an issue by removing as many manual steps as possible.
* **How to Measure**
* **How it can be abused** – The “recovery” needs to be a true recovery where the issue has been fixed. Rolling back to a previous version of an application may get it working for users, but the issue hasn’t necessarily been solved.
* **Benefits –** The quicker an issue can be resolved, the quicker the application can get back to functioning correctly.

## Mean Time to Detection

* **Description** – The average amount of time it takes a team to detect issues. As teams start to implement application telemetry, the amount of time it takes to detect an issue should decrease. Additionally, a comprehensive suite of unit tests, integration tests, acceptance tests, etc. should provide quick feedback as to whether or not an issue has been introduced.
* **How to Measure**
* **How it can be abused** – Teams need to respond to issues that have been detected. This statistic quickly loses its value if issues are allowed to linger.
* **Benefits –** We want to be able to detect issues as soon as they happen so we can start introducing fixes or roll back to a previous working state.

## Percentage of Failed Deployments

* **Description** – A measure of the total percentage of failed deployments over time. After implementing DevOps practices, this number should drastically decrease. Teams should additionally try to measure the amount of time between failed deployments as that should additionally decrease.
* **How to Measure**
* **How it can be abused** – Teams not properly counting deployments as failed. Failed deployments goes beyond just an error in the actual deployment process to things like:
  + Was a bug introduced?
  + Were other systems unintentionally affected?
  + Did the release negatively affect the user experience?
* **Benefits –** Less failed deployments increases user confidence in the product as well as cutting down on any potential critical, value losing issues.

## Defect Leakage

* **Description** – This is a measure of how many of the defects that we should be catching in lower environments are making their way out to production. This is important to capture to figure out if a team is catching things in lower environments or if their tests are allowing things to get past them.
* **How to Measure**
* **Benefits –** Catching issues in lower environments reduces the possibility that they’ll make their way out to production and in front of our users.

## Number of Defects Opened per Sprint

* **Description** – This tracks the average number of defects being opened by a team per sprint. A defect is anything that is identified after the story card is “done” or a regression test problem. Over time this number should start to go down as teams put in more automation testing as issues should be caught more frequently in lower environments.
* **How to Measure**
* **How it can be abused** – This can be abused by teams not accurately reporting issues. Teams need to make sure that once a card is marked “done”, that the card does not get reopened if an issue is found even if the card was completed in that same sprint.
* **Benefits –** Less defects being opened during sprints means teams have more time to work on features and deliver value.

## Percent Regression Test Suite Automated

* **Description** – The amount of the team’s regression test suite that has been covered by some form of automated test. The majority of regression tests should be automatable as any test that has a repeatable outcome is one that likely can be automated.
* **How to Measure**
* **How it can be abused** – If a team isn’t committed to doing regression testing then likely the number of regression tests they have is low so it would relatively easy to have a high percentage of coverage here.
* **Benefits –** As more of the regression test suite is automated, more time can be spent on exploratory testing instead of manual regression testing. These tests can then be run as part of a pipeline meaning that the regression test suite can be run more often, potentially catching issues we’re not always testing for on individual cards.

## Percent Test Cases Automated

* **Description** – Similar to the regression test suite, this is the total number of test cases that have been covered by some sort of automated test. Again, any test that has a repeatable outcome should be one that we can write an automated test against.
* **How to Measure**
* **How it can be abused** – As is the case with regression tests, if a team isn’t committed to testing then they likely don’t have a lot of test cases available to be automated. Teams need to make sure they’re also keeping track of how many requirements can have a test case around them for this to hold meaning.
* **Benefits –** More test cases being automated means fewer tests there are to run manually, which in turn saves time.

## Test Strategy Documented

* **Description—**Teams need to ensure that they’ve taken the time to capture and document their test strategy in a way that’s easy to access and reproduce.
* **How to Measure –** Yes or No
* **How it can be abused –** Teams must commit to a comprehensive testing strategy. Simply having one and then neglecting it doesn’t benefit the team at all.
* **Benefits –** The team will easily determine their overall test plan.

# Tier II Metrics

## Next/Last Release

* **Description**—Teams should always track their most recent release and when the next one is scheduled. The time between releases should decrease over time.
* **How to Measure**
  + Release Date
  + Feature(s)
  + Available to user (feature flag)
  + Deployment time
  + Deployment size

## Application Telemetry Data

* **Description** – Key statistics about a team’s application generally tracking data about the performance of pages, features, etc. These are typically necessary to be able to inform other DevOps measurements.
* **How to Measure**
  + Code libraries are typically available that are able to track these like Application Insights and Google Analytics among others.
  + Change in user volume per time period – helps teams know when they should scale their applications to meet user demand
  + System uptime/availability – helps teams to know when the system is going down so they can account for it
  + Page/feature usage – helps teams know what features and pages their users deem important
* **How it can be abused** – Teams need to make sure they also understand the context behind these metrics. It’s easy to accidentally jump to the wrong conclusions. Making sure that a team is taking into account the entire picture the telemetry is telling them instead of hyper-focusing on specific statistics is important.

## Features in Production

* **Description** – A measurement of how many features have been deployed out to the Production environment. Teams should see more features being pushed to production more frequently as time goes on.
* **How to Measure** - Number of Features in Production
* **How it can be abused** – Raw number of features in Production can sometimes be misleading. Teams need to take into account how many of those features are being used and are working as intended. More features also means that there is more code in production to support.

## User Feature Adoption (Feature Debt)

* **Description** – A measurement of how much a feature that has been deployed is actually being used versus how much it was intended to be used. This is important to find out if teams are focusing on what their users care about. Additionally, teams should be tracking their overall feature usage.
* **How to Measure**
* **How it can be abused** – Outsiders may make assumptions about the value of a feature when it’s not usable or it’s hidden within the application.

## Frequency of Pull Requests

* **Description** – The amount of pull requests a team is doing over a time period. As teams become more mature, the frequency of pull requests should increase. Teams should work in short-lived feature branches and create pull requests as soon as the branch is ready to be merged into the main branch.
* **How to Measure**
* **How it can be abused** – Teams need to make sure they’re taking the pull request process seriously. It’s very easy for teams to fall into a trap where they’re just immediately approving pull requests rather than actually taking the time to make sure the code is good to be merged into a main branch.

## Frequency of Production Defects

* **Description** – The average amount of time it takes a team from when an issue is first reported to how long it takes to properly resolve the issue. DevOps practices typically introduce automation around deployments which drastically reduces the amount of overhead required to push out a fix for an issue by removing as many manual steps as possible.
* **How to Measure**
* **How it can be abused** – The “recovery” needs to be a true recovery where the issue has been fixed. Rolling back to a previous version of an application may get it working for users, but the issue hasn’t necessarily been solved.

## Code Coverage

* **Description** – A measurement of the amount of code that is covered by unit or integration tests. The main way teams can increase this metric over time is to use Test First Development, particularly by focusing on the Test Driven Development practice.
* **How to Measure**
  + Typically there are tools available that will run tests and then calculate coverage like Visual Studio, SonarQube, etc.
* **How it can be abused** – Tests involved in code coverage need to be relevant. Typically this just calculates code that is covered by tests without necessarily factoring in what the tests are doing. A test could not even assert anything and call the code and tools will typically count that code as covered.

## Test Coverage

* **Description** – The measurement of how much of the entire application is covered by tests including unit tests, acceptance tests, performance tests, etc. This is more than just code coverage and takes into account the totality of tests around an application
* **How to Measure**
  + Teams typically come to an agreement over a set of tests they deem constitutes their test coverage.
* **How it can be abused** – Test coverage needs to be comprehensive. Merely doing unit tests or integration tests doesn’t cover the actual running application.

## Team Velocity

* **Description** – The average amount of effort points a team accomplishes per iteration. Combining Agile and DevOps practices should lead to teams accomplishing more points per sprints before eventually leveling out as a team matures.
* **How to Measure**
* **How it can be abused** – Teams need to make sure that effort points stay consistent between cards over time. Teams can inflate the effort points on stories which can make it seem like a team is doing more effort over time.

## Backlog Health

* **Description** – How much refined work the team has ready in the backlog.
* **How to Measure-** Total refined work in the backlog/average velocity of the team for the last five sprints
* **How it can be abused** – Too much refined work is waste and this metric can be swayed by large swings in average velocity.

## Number of Test Cases per Requirement/Story

* **Description** – The average amount of test cases that are being associated with stories. Over time as teams starting applying Agile this number should increase, hopefully covering the majority of test cases.
* **How to Measure**
* **How it can be abused** – In most of the projects we don't have requirement documenters and we need to use ITPI to go with high-level functionality requested. Story cards are created from a development point of view. Few story cards are just for configuration, few are for code/configuration migration in lower environment and few are for environment setup and deployment. Hence for preparing this metrics we might have to consider only development story cards where testing is applicable. Also when we say applicable testing, the testing we are considering is unit testing, integration testing or system testing.

## Number of Requirements/Stories without Test Cases

* **Description** – On the other side of the previous metric, teams should also be tracking how many requirements or stories they have that haven’t been covered by some sort of test case.
* **How to Measure –** Count

## Number of Test Executions per Sprint

* **Description** – This is a measurement of how many test executions are being run by teams throughout the course of their sprints. As teams start to automate more of their test cases and build out a Continuous Integration/Continuous Delivery process that runs those test cases, the number of executions should start to go up over time.
* **How to Measure**
* **How it can be abused** – Teams need to be honest about what constitutes a test execution. Merely running one or two tests is not an execution. Here we’re looking for teams to be running entire automation suites.

## Requirement/Story Card Test Coverage in Sprint

* **Description** – This measures how many of the stories that have been pulled into the sprint have been appropriately covered by test cases. As teams start to implement Agile and DevOps processes they should see the coverage number get close to 100%.
* **How to Measure**

## Number of Test Cases in Regression Test Suite

* **Description** – This is a count of the total number of test cases a team has in their regression test suite. This number should grow organically over time as teams add more features and then tests to cover those features.
* **How to Measure** – Count

## Percentage of Regression Test Run per Sprint

* **Description** – This is a measurement of how much of the regression test suite is being run on average during sprints. This number should get pretty close to 100% as teams implement automated tests to cover their regression tests and then have those tests get run automatically as part of their CI/CD pipelines.
* **How to Measure**

## Percent Story Card Unit Test Automated

* **Description** – This is the number of story cards that have been covered with unit tests over the overall number of story cards. Not every single card is going to be able to be unit tested, but if teams are implementing Test Driven Development then they should be able to cover every other card that can be.
* **How to Measure**

## Test Strategy Reviewed and Approved

* **Description –** Building off of the previous metric, teams need to make sure they’ve had their test strategy reviewed and approved to avoid the potential pitfalls of the previous metric.
* **How to Measure –** Yes or No (note the date)

## Test Executions outside Sprint

* **Description -** If test cycles occur outside the Sprint plan they need to be monitored/measured separately. Examples are regression tests, integration tests, performance tests, acceptance tests, etc.
* **How to Measure –** Count

## Number of Test Cases Developed in Sprint

* **Description –** A measurement tracking the number of test cases the team created during the sprint. Over time this number should rise on a per sprint basis.
* **How to Measure –** Count

## Number of Defects Opened

* **Description –** Tracks the number of defects that the team has opened over time. Hopefully as teams buy into automated testing and CI/CD practices, they’ll introduce less defects in their sprints. This should be categorized by severity and priority with a trend over time presented in graph form.
* **How to Measure –** Count

## Number of Defects Closed

* **Description –** This tracks the number of defects that the team has closed. As the number of defects being opened decreases, teams should have more time to focus on closing out the open defects they have. This should also be categorized by severity and priority with a trend over time presented in graph form.
* **How to Measure –** Count

## Number of Defects Open

* **Description –** A measurement of the amount of defects the team still has open. This one should as well decrease over time. This should also be categorized by severity and priority with a trend over time presented in graph form.
* **How to Measure –** Count

# Links

<https://stackify.com/15-metrics-for-devops-success/>

<https://www.datical.com/database-devops/9-metrics-devops-teams-tracking/>

<https://devopedia.org/devops-metrics>

<https://techbeacon.com/security/5-application-security-metrics-should-matter-your-team>