ASERVO Software

KNOW-HOW

SLAs and KPIs for CI/CD

Finding and defining good metrics in the DevOps environment

ASERVO Software

Finding and defining good metrics in the DevOps environment

Continuous Integration (CI) and Continuous Delivery (CD) are basic components of agile software development. They ensure that good software is developed quickly. Automated, flexible processes are essential for development and operation (DevOps). But how can the success factors of agile projects be defined? Do SLAs and / or KPIs measure success? Do SLAs and / or KPIs measure success? Which indicators make sense and when?

Usability of KPIs and SLAs on CI/CD //

If a company operates CI/CD as a central service, SLAs are usually specified for this service - as for other company services. These generally include specifications for **availability, performance** and/or **reaction times** to incidents. SLAs originated from the IT environment and have been used for quite some time, especially in the area of hosting. There, the framework conditions for the provision of a service are sufficiently well understood and definable.

This does not apply in the same way to KPIs. Formulating them for an application service is generally not that easy. What's more, in an agile or DevOps environment, KPIs also require **constant feedback**. This continuous feedback continuously modifies the initially selected KPIs. Over time and with the increasing maturity of the CI/CD service, established metrics take a back seat and other - usually "superior", more complex - metrics increase in importance. This is normal and no reason to panic.

KPIs are then often used to control employee evaluations and/or bonuses. For a long time these specifications came from the upper hierarchical levels. Increasingly, DevOps teams now develop KPIs together, which are then subject to constant feedback so that they can change during operation.

The most important mantra for DevOps is 'Measure', i.e. a large number of metrics are recorded and evaluated with the help of suitable tools.

Definitions

CI (Continuous Integration) refers to the translation and testing of software after each commit / push. The end result is usually a binary artefact that is stored in a repository for further use.

CD (Continuous Delivery / Deployment), as a superset of CI, tests the interaction of the generated artefacts with the goal of reaching production maturity. Continuous Delivery provides the corresponding binary artefacts for deployment and automatically sets them to productive - after successful testing.

SLAs (Service Level Agreements) have existed in one form or another for a very long time in the field of hosting They describe guaranteed characteristics of a service, on which the customer and the contractor have agreed prior to the performance of the service. They have similarities with contract terms or guaranteed technical properties.

KPIs (Key Performance Indicators) are - generally speaking - data relating to the achievement of goals. They are intended to provide information on how good or bad the measured values are in comparison with given targets or average values of comparable companies.

DevOps (Development + Operations) is a procedure from the agile environment that combines developers, testers and infrastructure operators in one team ("You build it, you run it, you fix it").

For CI/CD as a service, this means recording a host of **metrics** from a variety of systems:

- upstream systems such as SCM, LDAP, mail, HTTP proxy, ticketing
- Infrastructures such as build servers, agents, test machines
- Performance data of the application in production environments

Once all these measured values have been recorded, the work begins...

Defining indicators correctly

//

SLA definitions must be mapped to the measured data: does "available" mean whether the system in question is available at all, or that it responds to a defined request within a defined maximum time, for example? This corresponds to the formulation of a "Definition of Done" (DoD) from the agile environment.

It is also important to find an equivalent for KPIs in the collected data. An **"Indicator"** is not an absolute measured value. An indicator is a prompt to take a closer look. If there are deviations (mostly on the time axis), you must always look at the reason and not simply accept the value.

DoD is the abbreviation for a list of criteria that a product must meet to be considered finished.

Why KPIs are not quite so simple

//

In larger companies there is a tendency to derive assessments or variable salary components (bonuses) from KPIs directly. This is often insufficient, however. Many of the values from the overview below sound plausible at first, depending on your point of view, but on closer inspection and taking human nature into account they reveal some weaknesses.

Examples:

Lines of code per developer, per day - actually came from a highly paid consulting firm, and was fortunately rejected because it was obviously nonsense.

Cost distribution after use - if I want to establish a service, I should not receive payment for utilisation, but rather penalise non-use, and thus bill everyone for the service costs. Those who don't use the service will have problems justifying this.

Build duration - the build duration is influenced by too many different factors, such as the number and thoroughness of tests, parallelisation within the build, availability of resources, etc.

Number of errors of a component in an iteration - not a good indicator because it depends too much on individuals and environmental conditions. May, however, be good for improving the process, e.g. commits / pushes only once all tests have been run locally.

Number of tests - the number of tests can increase easily without actually increasing the quality.

Test coverage - only suitable as a sole criterion under certain conditions. What is more important is that the value continuously improves. It is also important, however, to have a common definition of what is to be tested and how.

Ticket handling time - typically causes tickets to be closed mercilessly, without actually fixing the problem in question. A combination of measured values that take into account the steps within the workflow, including loops as well as other factors, is better.

Errors found in production - here an analysis as to why errors are not found until the system has gone live would be better

Disabled tests / number of tests per release - if abnormalities are found, this is a good time to have a look at the causes: Is the code currently being refactored, are new third-party libraries being used, which means some of the existing tests cannot be used without being adapted? A comparison with the previous release would be worthwhile here.

Architectural index / Maintainability index (e.g. from SonarQube) - a very good indicator of code quality, but not for other aspects of the application.

Number of known vulnerabilities per release, per application, broken down / weighted by severity. Realistically, you should only measure the improvement and not the absolute value.

Infrastructure utilisation - depending on available resources, it makes general sense to measure utilisation. However, the interpretation depends on many details, e.g. do I have to evaluate a static infrastructure with bare metal or VMs differently in this respect than a Kubernetes cluster.

Visualisation of KPIs - selected examples
//

The following figures show examples using a combination of Prometheus and Grafana. Utilisation of the ELK stack (Elasticsearch, Logstash, Kibana) is common in this context.

Prometheus is an open source toolkit for system monitoring, alerting and trending on the basis of time series. https://prometheus.io/

Grafana is an open source monitoring program, which can be used to analyse, prepare and graphically display data from different data sources. https://grafana.com/

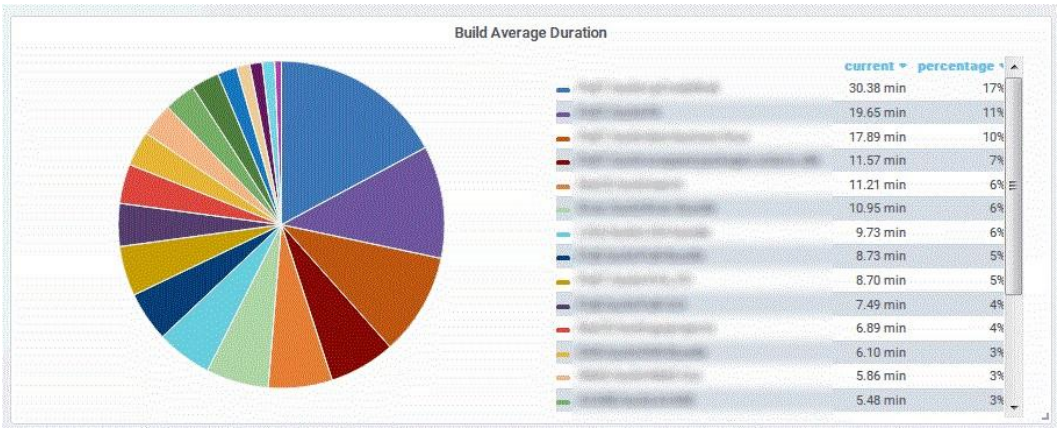

Fig. 1: Average build duration

The graphic in figure 1 is well suited for an initial overview, but there are several things to question before KPIs can be derived from it:

- Are the builds homogeneous, i.e. are all builds structurally the same or is there a colourful mix of micro services, J2EE and C#?
- How was the delta at earlier points in time? What are the expectations on the part of the developers?

Fig. 2: Build results

The graph in figure 2 is also well suited for an initial overview, but the results are not meaningful without knowledge of the context:

- Is the procedure test-driven? Depending on the requirements, expectations can change as to which part of the builds should be successful.
- What are the causes for failed builds? Infrastructure or program problems?

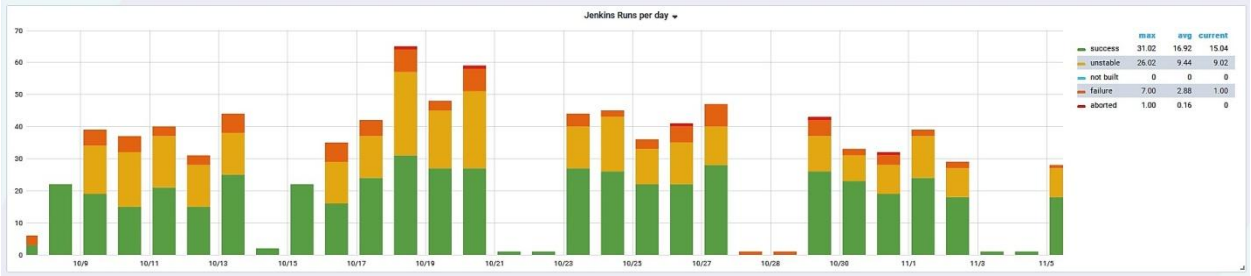

Fig. 3: Builds per day

The builds per day as shown in figure 3 provide a good entry point for the daily controls of the service provider.

- A sudden accumulation of failed builds should give rise to further investigation.
- If the relationship between successful, unstable and failed builds remains more or less consistent, the service will essentially run smoothly.

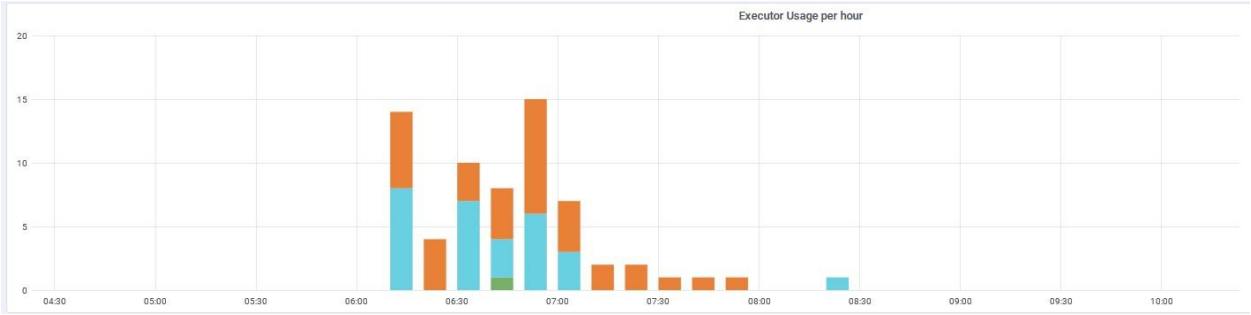

Fig. 4: Executor usage per hour

The executor usage per hour in figure 4 provides an important assessment, but the context must also be taken into account here:

- Do I have a limited number of executors of a certain type? I should measure this separately.
- Do I have a limit regarding the maximum number of executors e.g. due to the infrastructure? I should measure this separately, too.
- Generally, CI will result in a typical split between scheduled and push/commit-controlled builds. Here you should keep the number of overlaps as low as possible. Daily builds usually accumulate before lunch and before the end of the working day, so scheduled builds should take place in the early hours of the morning or late in the evening.

Fig. 5: Queued builds

Queued builds, as shown in figure 5, are a sign that there are not enough executors available.

- In such a case, it is a nightly build that builds many components. At night, this shouldn't bother anyone, but during the day valuable resources would be blocked.
- Queue peaks can also occur when all masters want to access the agent pool at the same time.
- Another reason may be that there are not enough of a certain type of agent available.

Taking this into account, what is the actual goal of CI/CD? If you keep an eye on this question, the answer is usually to produce software in good quality and at high speed. Good quality includes, for example, maintainability, performance, exclusion of known critical security gaps, adherence to governance, risk and compliance standards.

For every one of these terms, all participants - whether operators, users, service sponsors or others - have to agree on a common view in advance and, in case of doubt, adapt this view during the course of the project.

In order to be able to develop software under this premise, the interaction of several tools is required:

- SCM (e.g. Git, Subversion, Mercurial)

- Ticketing (e.g. Jira)
- Build (e.g. Jenkins)
- Code analysis and test evaluation (e.g. SonarQube)
- Vulnerability analysis (e.g. Nexus Lifecycle, Nessus)
- Unit tests, integration tests, user acceptance tests, performance tests, regression tests
- Application performance monitoring

As part of a central CI/CD service, all of these systems provide measured values that can be used for KPIs and SLA monitoring.

Which of these measured values are actually relevant depends on many specific details. Usually it makes sense to start with a handful of simple values and then refine them further once you have seen the first evaluations with real data. It is also important to determine **what** you want to measure.

How do I find the right KPIs?

There is **no single set of KPIs** that fits any setup. In fact, specific KPIs have to be determined based on the customer and the tools and technology used. It is best to start with a few simple KPIs and modify them as experience increases to fit the **purpose** in question.

Business KPIs

From a business perspective, there are two central KPIs:

1. **"Idea to Production"**, also **"Time to Market"** - the time between the formulation of an idea as a ticket and the "go-live" of the feature. Several factors are taken into account here:
    - How precisely the idea was recorded in the ticket (description, acceptance criteria)
    - "Size" of the ticket (small modification / addition vs. change of architecture)
    - Prioritisation / workload of developers
    - Speed of the CD pipeline
    -
2. **"Hotfix deployment"**, also **"MTTR (Mean Time To Repair)/ MTTF (Mean Time To Fix)**" - the time between the (analysis of a problem and the) creation of a hotfix and the "go-live". Several factors are also taken into account here:

    - Quality and scope of the previous analysis
    - Speed of the CD pipeline vs. completeness of the tests
    -

Experience shows that it makes sense to start thinking about the hotfix deployment at an early stage (which tests do I not need? Special pipeline or special parameters of the "normal" pipeline?), so that you don't panic and make mistakes in the event of an emergency.

Other measured values that may be relevant for the operation of a central service:

Change success rate
The number of successful builds / deployments relative to the total number of builds / deployments. If the change success rate is too low, you should analyse where the error lies in the pipeline. If sources cannot be compiled, the is usually to blame. If the pipeline fails during integrative testing, better mocks may be needed to intercept such errors earlier on. If the pipeline fails at quality gates, the associated data may not be available in the developer's IDE, or he may not know what to do with the existing information.

Deployments per month, per pipeline / application
Enables the comparability of different applications and technologies, provided that the framework conditions are reasonably similar.

"Lead time for change"
How long does it take for a commit to reach PROD (minimum, maximum, average)? Is related to 1. and 2.

"Batch size"
How many story points per deployment (minimum, maximum, average)? This is based on the individual cases and the Scrum velocity.

Application KPIs
//

Code quality
Evaluation of test coverage, maintainability index, architectural index, etc., usually as a delta over time or against set standards. Derived indices, such as the maintainability or architectural index, are less susceptible to manipulation than simple metrics like test coverage. In any case, the measurement procedure must first be coordinated - if, for example, getter/setter are to be tested, what about the generated code?

Critical security bugs
Total number and/or number in new / changed code at a certain level. Corporate security may also be able to highlight certain individual errors here.

Performance deviations
Should always be treated with caution, but should definitely be observed. If there are unexpected deviations from previous measured values, the cause should always be determined.

System KPIs

//

Availability

What percentage of the service is available at the previously agreed times (24 x 7 vs. 9 x 5)? Are there any pre-defined maintenance periods on the infrastructure or service side?

Erroneous vs. successful calls

When does the service deliver errors? This can happen for example with session timeouts or deep links. With some applications, deep links don't work well, and then you have to find other ways to provide the desired functionality.

Queue wait time (minimum, maximum, average)

How long does a job have to wait on average / maximum until it is dealt with? If waiting times occur, what is the cause? Are there generally too few agents, are there too few agents of a certain type, do all nightly builds start at the same time?

Builds/Deployments per day day/week

Actual, meaningful values depend on the application and the type of deployment. Here too, depending on the goal, the delta over time is the most interesting aspect; as a rule, the goal is to create more deployments per time unit.

Utilisation rate of the build agents

Setups with static machines are fundamentally different from dynamic infrastructures such as Docker / OpenShift / Kubernetes / AWS / Azure / etc. For static machines, I aim for a load that is as evenly distributed as possible. Working with a dynamically provided infrastructure is more about limiting or capping costs.

Process KPIs

//

Process-related KPIs are indicators of how well processes are really utilised:

WTFs per day/week

How often does the team experience WTF ("What the fuck") moments? How often do things come up that nobody expected before?

Impediments per sprint

How many impediments come to light per sprint?

Impediment removal time

How long does it take to remove an impediment?

Non-availability of the product owner

A common problem when introducing agile working: project managers become product owners, but little changes apart from this. This automatically leads to the fact that they cannot do justice to the task of a product owner - neither from the point of view of the company nor from the point of view of the team(s).

Summary
//

SLAs should be agreed between the operator and the user prior to each commissioning of a service to ensure there are no **misunderstandings** later one due to completely different expectations.

KPIs are indicators that generally require a closer look when changed. They are only poorly suited 1:1 for measuring quality, usually the difference at a previous point in time is the better approach. They should be re-evaluated and revised regularly.

There are different types of KPIs, depending on the **point of view**; and each of these points of view has its justification. There is often the danger of getting too involved with the purely technical measured variables. However, experience has shown that it is the application and process KPIs that provide the most insight, even though their determination involves more effort. The technical KPIs, on the other hand, are more of a help when it comes to the diagnosis and removal of weaknesses.

ASERVO Software GmbH
Konrad-Zuse-Platz 8
81829 München

Tel.: +49 (0) 89 7 16 71 82-40
Fax: +49 (0) 89 7 16 71 82-55
Email: rmayr@aservo.com

www.aservo.com