TMMi® FOUNDATION

TMMi® in the DevOps world

#SuccessWithTMMi

www.tmmi.org

TMMi is intended to be lifecycle independent. This document explains how the TMMi test improvement model can be used and applied beneficially in a DevOps context. The level 2 TMMi process area and its goals are discussed one by one. In the next version, we will also address the TMMi level 3 process areas. It is stated how these relate to DevOps and what the practices will typically look like. If a practice is not expected to be performed in an DevOps context, because it has no added value, this is also explicitly stated. Future versions of this document will also address the process areas at higher TMMi levels. Note that this document is not intended to be a full DevOps testing syllabus, but rather a document that "only" shows how the TMMi goals should be interpreted in a DevOps context and provide ideas and directions for further study.

Contributors

Katalin Balla (Hungary)
Suresh Chandra Bose (USA)
Kari Kakkonen (Finland)
Mattijs Kemmink (The Netherlands)
Rik Marselis (The Netherlands)
Szilard Szell (Hungary)
Erik van Veenendaal (The Netherlands)

Revisions

This section summarizes the key revisions between releases this document.
This section is provided for information only.

Release	Revision Notes
V1.0	Initial version of the “Testing in the DevOps world” document

Table of Contents

1 TMMi® in the DevOps world

1.1 Introduction

This document explains how combining a DevOps approach for software development with the TMMi test process improvement model is a possible route to achieve your business objectives. It explains how TMMi can be used and applied beneficially in a DevOps context. Proven DevOps-based alternatives are provided to traditional testing approaches that implement TMMi practices. This document covers how TMMi can help DevOps organizations, irrespective of how mature the organization is on their DevOps journey, by providing reminders of critical testing practices that frequently lose visibility as organizations grow and project pressure increases. Each TMMi process area and its specific goals are discussed one by one. It is stated how these relate to DevOps and what the associated practices will typically look like. If a practice is not expected to be performed in an DevOps context, since it has no added value, this is also explicitly stated.

In today's fast-paced DevOps world, where continuous integration, continuous delivery, and automation drive software development, ensuring quality at every stage is paramount. The Test Maturity Model Integration (TMMi) provides a structured framework to assess and improve testing practices. As DevOps teams strive for rapid releases without compromising on quality, integrating TMMi offers a roadmap to enhance testing maturity, align with business goals, and deliver results. This synergy between TMMi and DevOps creates a balanced approach to accelerate development while maintaining high standards of quality and reliability.

This document has a number of target audiences, the main groups being:

- Mature traditional organizations that want to move to DevOps while keeping their process maturity
- Less mature organizations that want to start applying TMMi to increase their test maturity, and move to DevOps in parallel
- Agile organizations that have a structured Agile testing process, and now want to move to DevOps
- DevOps organizations that are successful and are growing. As a result they need some process maturity while at the same time they want to keep the benefits of being Agile.

This document is not intended to be used independently of the original TMMi model. It is intended to provide guidance to those performing test process improvement in a DevOps environment on the interpretation and meaning of the various TMMi goals and practices. This document supplements the original TMMi model and should be used as a complementary document.

1.2 DevOps and DevSecOps

DevOps is a philosophy and set of practices aimed at bridging the gap between software development and IT operations teams. By fostering a culture of collaboration and shared responsibility, DevOps seeks to streamline the development, deployment, and maintenance of software. The core idea is to break down the traditional silos between development and operations, enabling more efficient and effective delivery of software. At its heart, DevOps emphasizes automation and continuous processes. Automation plays a crucial role in minimizing manual tasks, thereby reducing errors and increasing speed. This includes automating the integration of code changes, testing, and deployment processes through Continuous Integration (CI) and Continuous Deployment (CD) practices. Another strong emphasis is the needed change into DevOps culture of autonomy, leadership, collaboration, frequent experimentation, shorter development cycles, increased deployment frequency, and more dependable releases, in close alignment with business objectives and customers.

The DevOps approach leads to numerous benefits, including faster time to market, improved collaboration between teams, higher software quality and reliability, and increased scalability and flexibility. By enabling quicker detection and resolution of issues, DevOps also reduces downtime and enhances system resilience. Ultimately, DevOps is as much about cultural change as it is about technology and processes. It requires a shift in mindset where both development and operations teams work towards common goals, embrace feedback, and continuously strive for improvement. By integrating these elements, DevOps helps organizations deliver software more efficiently, reliably, and securely.

DevSecOps

DevSecOps, short for Development, Security, and Operations, is an approach that integrates security practices into the DevOps process. This methodology aims to ensure that security is a shared responsibility throughout the entire IT lifecycle, from initial development to deployment and beyond. DevSecOps aims to deliver secure software faster and more efficiently, reducing the risk of security breaches.

1.3 Test Maturity Model integration (TMMi)

The TMMi framework has been developed by the TMMi Foundation as a guideline and reference framework for test process improvement, addressing those issues important to test managers, test engineers, developers and software quality professionals. Testing is defined within TMMi in its broadest sense to encompass all software product quality related activities. TMMi uses the concept of maturity levels for process evaluation and improvement. Furthermore, process areas, goals and practices are identified. Applying the TMMi maturity criteria will improve the test process and has shown to have a positive impact on product quality, test engineering productivity, and cycle-time effort. TMMi has been developed to support organizations with evaluating and improving their test processes.

TMMi has a staged architecture for process improvement. It contains stages or levels through which an organization proceeds as its testing process evolves from one that is ad hoc and unmanaged to one that is managed, defined, measured, and optimized. Achieving each stage ensures that all goals of that stage have been achieved and the improvements from the foundation for the next stage. The internal structure of TMMi is rich in testing practices that can be learned and applied in a systematic way to support a high quality testing process that improves in incremental steps. There are five levels in TMMi that prescribe the maturity hierarchy and the evolutionary path to test process improvement. Each level has a set of process areas that an organization must implement to achieve maturity at that level. The process areas for each maturity level of TMMi are shown in Figure 1.

A main underlying principle of the TMMi is that it is a generic model applicable to various life cycle models and environments. Most goals and practices as defined by the TMMi have shown to be applicable with both sequential and iterative life-cycle models, including Agile. However, at the lowest level of the model, many of the sub-practices and examples provided are (very) different depending on the life cycle model being applied. Note that within TMMi, only the goals are mandatory, the practices are not. TMMi is freely available on the web site of the TMMi Foundation and also available in published book format. The model has been translated in many languages including Spanish, French and Chinese.

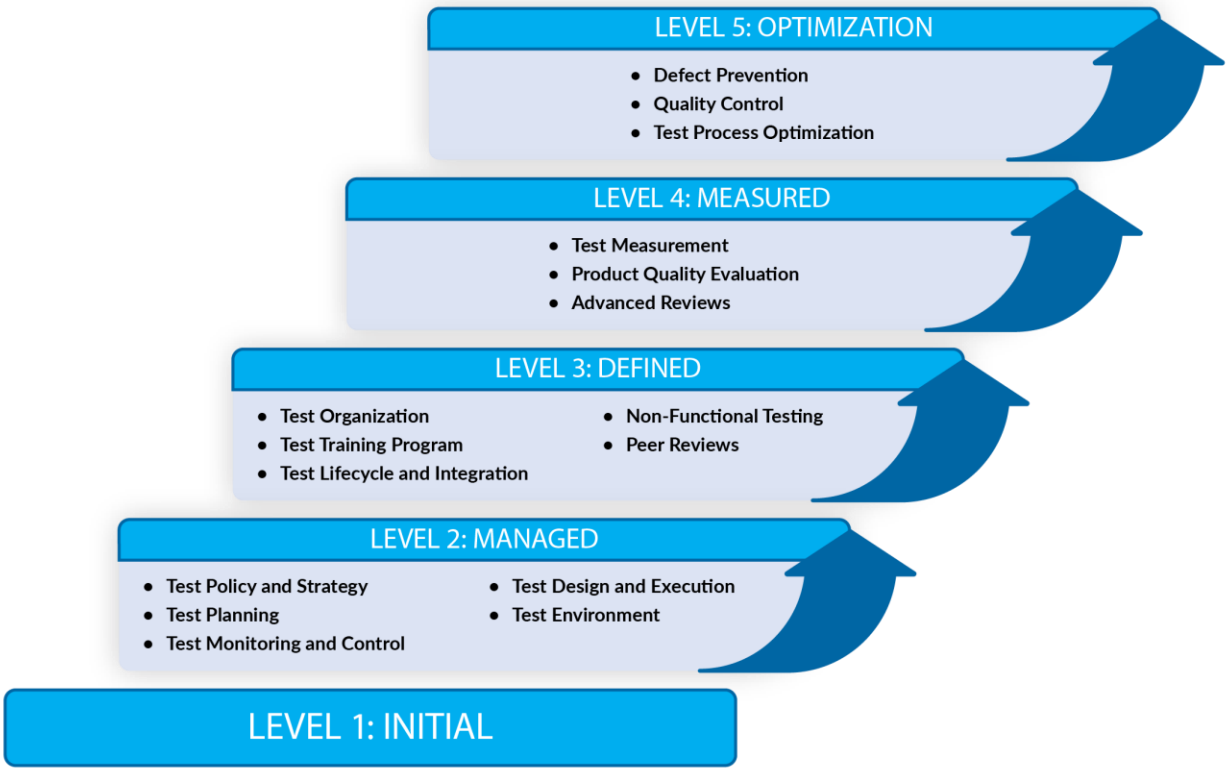

Figure 1: TMMi maturity levels and process areas

1.4 Testing in a DevOps context

1.4.1 Challenges of testing in a DevOps context

Testing in a DevOps context presents unique challenges due to the nature of continuous integration and continuous delivery (CI/CD) pipelines, the collaboration between development and operations teams, and the rapid pace of development cycles. Some of the key testing challenges are:

- *Continuous Testing*: Tests must be executed quickly and frequently to keep up with CI/CD pipelines. There is a high reliance on automated testing, which requires robust and reliable test scripts and frameworks.
- *Environment Management*: Ensuring proper test environment management and provisioning of realistic test environments for continuous testing can be complicated. There is often a lack of parity between production and test environments, which can lead to undetected issues. Another challenge is managing and maintaining consistent configurations across multiple environments. Environments should be deployed and configured automatically, usually on demand, and deleted after testing.
- *Test Data Management:* While test data should also be production like, consistent, and deployed on demand, compliance to privacy guidelines is also a must.
- *Integration and Regression Testing*: Integrating multiple components and services that have complex interdependencies and as such making it challenging to test them. Also ensuring that new changes do not break existing functionality, which requires comprehensive regression test suites.
- *Tooling and Infrastructure*: Integrating various testing tools with CI/CD pipelines and other DevOps tools and ensuring the testing infrastructure can scale to handle the increased load and complexity.

- *Cultural and Organizational Challenges*: Fostering collaboration between development, testing and operations, whom traditionally may have operated in silos with often contradicting incentives. Ensuring team members have the necessary skills to handle automated testing, continuous integration, and deployment practices.
- *Performance Testing*: Simulating real-world loads and conditions in a controlled test environment, but also integrating performance testing into the CI/CD pipeline to continuously monitor and optimize performance.
- *Security Testing*: With fast-moving pipelines, security testing often gets sidelined. Integrating security testing (DevSecOps) without slowing down development and ensuring vulnerabilities are caught early requires specialized tools and strategies, is a challenge to many teams. Security testing preferably needs to be integrated early and throughout the development lifecycle using automated tools to perform static and dynamic security testing.
- *Feedback and Reporting*: Providing quick, clear, and actionable feedback to the team based on test results. Generating comprehensive reports that provide insights into the quality of the build, yet keeping the process Agile.
- *Shift left practices*: Improve quality by moving tasks to the left as early in the lifecycle as possible. Shift Left testing means testing earlier in the process, e.g., by means of (code) reviews, static analyses and thorough unit testing.
- *Shift Right practices*: As the deploying changes to production as frequently as possible is key, a test strategy shall balance between testing activities in the CI part and observing the system usage in production in CD
- *Monitoring and Feedback Loops:* In DevOps, testing doesn't end with deployment; continuous monitoring and gathering feedback from production environments are crucial. Setting up effective monitoring tools and interpreting data from production to catch defects and optimize performance is often overlooked or underdeveloped.

1.4.2 TMMi and DevOps

The mistaken belief is that the TMMi and DevOps approaches are at odds. DevOps approaches and TMMi can not only co-exist, but when successfully integrated will bring substantial benefits. There is also a challenge of looking at testing differently, being fully integrated within development and having a CI/CD pipeline and what that means in the context of a “test” improvement program. Note that the “i” of TMMi refers to the fact that testing should be an integrated part of software development, and not be treated as something that is totally separate. Using the TMMi model provides reminders of critical testing practices, e.g., product risk analysis, that are often “forgotten” in an DevOps context. This document will show with examples that TMMi and DevOps methods can effectively work together.

When implementing TMMi one must take into account that the intent of the TMMi model is not to “impose” a set of practices on an organization, nor is it to be applied as a standard to which one must “prove compliance”. Used appropriately, TMMi can help you locate the specific testing areas where change can provide value given the business objectives. This is true regardless of the lifecycle model that is being applied. It is important to always remember that TMMi practices are an expected component, but can also be achieved by what is referred to as “alternative” practice with respect to a defined TMMi practice. Always think, what is the intent of the practice, what is the rationale and how does it add value to the business? Often in a DevOps environment the intent is already achieved but through an alternative practice. Typically, “any” solution is compliant as long as it is driven by business needs! When using TMMi don’t be too prescriptive, this was not how TMMi is intended in the first place. Always interpret the TMMi goals and practices to the context of your

situation. In general, by first establishing the process needs within your specific business context, decisions can be taken on how to focus and drive process improvement priorities.

Most of the conflicts that arise between the TMMi and DevOps are based in either a historical TMMi view of what a "good practice" should look like when implemented, or a misunderstanding of the rationale for DevOps practices based on how they should support the DevOps way of working and values. TMMi experts, including (lead) assessors, will need to re-consider and potentially re-think messages that might be inadvertently shared related to what a "good TMMi compliant" practice should look like when implemented. When DevOps approaches are implemented appropriately together with TMMi processes, this will result in the effective implementation of testing practices, not their deletion. Note that in addition to the specific (testing) practices within the TMMi, there are also generic practices. The purpose of the generic practices is to support the institutionalization of a process area, which effectively means ensuring the organization has an infrastructure in place to support the process area when new people come in or other changes happen within the organization.

Moving from traditional based software development to DevOps can also bring out the initiative to prune and lean the processes as they are defined today. In this way TMMi based organizations will benefit from the Agile and lean way of thinking which both are present within DevOps. There has been a tendency for people to read things into the TMMi model that are not there and thus create unnecessary non-value-added processes and work products. By going back to the roots and improvement goals and using the TMMi model as it is intended, one will support the alignment of real value-added processes with real process needs and objectives. Pruning and leaning processes with an Agile/Lean mindset will result in processes that reflect what people really do and will ensure that only data that is used is being collected. This mindset will also bring a focus on keeping things as simple as possible, which is typically not easy but will bring a benefit for those practicing a TMMi implementation. Improvements within DevOps, guided by the DORA performance metrices of Change lead time, Deployment frequency, Change fail percentage and Failed deployment recovery time [DORA16], [DORA24], will typically take place through small, empowered teams that can take rapid action, which is another way in which TMMi can benefit from applying DevOps. Whilst TMMi is comprehensive, organizations cannot apply each end every TMMi element. To be successful, organizations must identify their own key testing practices and improvements requiring priority and focus.

1.4.3 Relationship ISTQB DevOps syllabus

TMMi and the ISTQB have entered an alliance to further promote the Software Testing profession together. Also during the development of this document, the TMMi technical committee and the ISTQB "Quality in DevOps" syllabus working party have worked together. TMMi is a test improvement model, as such it provides a pre-defined improvement approach with priorities based on the structure of the TMMi model. In this document the focus is on providing an interpretation of the TMMi improvements goals for those working in a DevOps context. It does not provide a detailed full description of good DevOps engineering practices. The ISTQB "Quality in DevOps" syllabus is a content-based document, as such it aims to provide a detailed description of good DevOps engineering practices, e.g., quality engineering (including testing) practices. The two documents are therefore highly complementary, in essence: TMMi states what should be improved (which processes), the ISTQB DevOps syllabus describes how things should be done (engineering best practices).

2 TMMi Level 2 Managed

2.1 Process Area 2.1 Test Policy and Strategy

The purpose of the Test Policy and Strategy process area is to develop and establish a test policy, and an organization-wide or program-wide test strategy in which the test activities, e.g., test types and test levels, are unambiguously defined. To measure test performance, the value of test activities and expose areas for improvement, test performance indicators are introduced.

2.1.1 SG1 Establish a Test Policy

Any organization that embarks on a test improvement project should start by defining a test policy. The test policy defines the organization's overall test objectives, goals and strategic views regarding testing and test professionals. It is important for the test policy to be aligned with the overall business and quality policy of the organization. Test improvements should be driven by clear business goals, which in turn should be documented in the test (improvement) policy. A test policy is necessary to attain a common view on testing and its objectives between all stakeholders within an organization. This common view is required to align test (process improvement) activities throughout the organization. Note that test objectives should never be objectives by themselves, they are derived from the higher level goal to establish working software and product quality.

The above is also true in an organization that has embraced the DevOps culture and practices DevOps software development. Indeed, within many organizations there is much discussion on the changing role of testing, independence of testing, test automation and professional testers in DevOps software development. These items and others are typically issues that should be addressed in a discussion with management and other stakeholders and documented in a test policy. Any organization, including those that practice DevOps, that wants to start a test improvement project needs to identify and define the business drivers and needs for such an initiative. Why else start an improvement project? By spending time to capture the true business needs, one can provide a context to make decision where to focus the (test) improvement priorities, e.g., on which process area. Note that a test policy is typically a one-page lean document, web page or wall chart at organizational level, and not a document at project level.

The TMMi specific goal Establish a Test Policy, including its specific practices, is fully applicable to organizations applying DevOps software development. However, in DevOps, Development, Testing and Operations are usually covered under the same process or at least they are tightly aligned. Thus a Test Policy is also expected to be an integral part of an overall DevOps policy, focusing more on quality engineering than just testing. Testing in DevOps is organized balancing Shift Left and Shift Right approaches, utilizing collaborative practices and Test First thinking. In DevOps, testing activities shall support all the three ways (as defined in the DevOps Handbook); the flow, the feedback, and continuous experimentation and learning.

As stated the test goals shall focus on a clear balance between Shift Left and Shift Right; proactive, design and test first thinking to build the needed quality and testing activities before deployment (within and around the CI/CD pipeline) to capture defects, and reactive measures to reduce service downtime in production in case a fault slips through, for example automatic roll-back mechanism to reduce down time, blue-green

deployment[1], or canary release to reduce impact. While the goal of testing activities is to reduce the probability of deploying a fault (faulty software) to production, deployment strategies are applied to reduce the impact of any given failure, resulting in better overall quality. In order to achieve impact reduction, testing activities of the deployment pipeline shall be streamlined to improve flow and lead times, while overall risk levels are still kept low.

In addition to testing throughout software development, Shift Right (testing in production) is also a common practice in DevOps, and this will affect the definition of the test goals as defined in the policy as well, focusing on quick reactions to issues in production, based on telemetry information, to reduce down-time, improving availability, thus improving quality. Site Reliability Engineering (SRE) practices cover monitoring health of systems, services and applications, and creating closed loop automation mechanisms to react to health issues/deviations [Beyer]. This is also called self-healing.

Due to the number of changes expected in DevOps, the test goals are focusing heavily on regression testing both on functional, as well as non-functional aspects. Some non-functional aspects could be measured in production (Shift Right) after deployment, while others needs to be tested before going to production, e.g., security.

Examples of test goals:

- Testing is focused on making sure changes deliver the expected value (acceptance test), and that they do not break any existing functionality (regression test)
- Testing shall support the development goals of the organization, like reduction of change failure rate, or increased deployment frequency.

Also independence and the high level process definition per test policy are typically affected in the context of DevOps. Regarding independence, testing is a role in the development organization, many times referred as DevTest to represent dual role. Thus, independence is usually low. Organizations usually rely on a so-called platform team with Software Development Engineers in Testing (SDET) whose main role is to develop and maintain testability aspects of the product and test automation solutions, to be used by DevTest roles. Additionally, some dedicated test teams might be appointed for more complex areas, like non-functional testing or complex end-to-end scenarios. In this case independence is higher, but still co-operation is tight. Regarding the high level test process definition, testing is now an integral part of the development process, or the CI/CD strategy, as such so will be the test process.

It is important to ensure that test policy, usually a lightweight/lean document, is distributed to the whole organization / value stream / Agile release train / tribe. As testing is a crucial part of the Agile and DevOps teams work, and not a separate organization, the policy should be known by all those involved and affected.

2.1.2 SG2 Establish a Test Strategy

The test strategy follows the test policy and serves as a starting point for the testing activities within projects. A test strategy is typically defined either organization-wide or program-wide. A typical test strategy is based on a high-level product risk assessment and will include a description of the test types and test levels that are to be performed. It is not good enough to just state for example that within each project integration testing

[1] A blue/green deployment is a deployment strategy in which you create two separate, but identical environments. One environment (blue) is running the current application version and one environment (green) is running the new application version.

and acceptance testing will be carried out. We need to define what is meant by unit and acceptance testing; how these are typically carried out and what their main objectives are. Experience shows that when a test strategy is defined and followed, less overlap between the various testing activities is likely to occur, leading to a more efficient test process. Also, since the test objectives and approach of the various test types and levels are aligned, fewer holes are likely to remain, leading to a more effective test process and thus higher a level of product quality.

A test strategy is a vital document within a DevOps environment. It defines on a high-level the testing to be done by the teams, mainly within the CI/CD strategy; what test levels and test types are executed and on a high-level their approach. For each test level and test type, the objectives, responsibilities, main continuous testing tasks, acceptance criteria/quality gate are defined. The test strategy is usually following the Agile testing quadrants (see figure 2), however is built around the CI/CD pipeline to represent the flow of activities within the pipeline. The pipeline shall be understood as not only the activities running by a CI/CD tool, but everything the teams do when they work on a backlog item. Test activities like reviewing and exploratory testing are happening outside of the CI/CD tools, however as an integral part of the flow/pipeline/value stream. Note, the pipeline can be used not only as the technology/tool, but as the visual representation of the activities, process, and people [see figure 3].

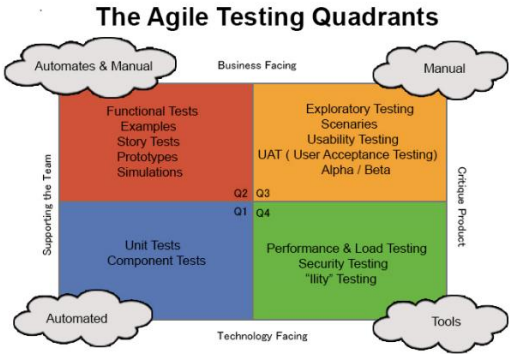

Figure 2: The Agile Testing Quadrants

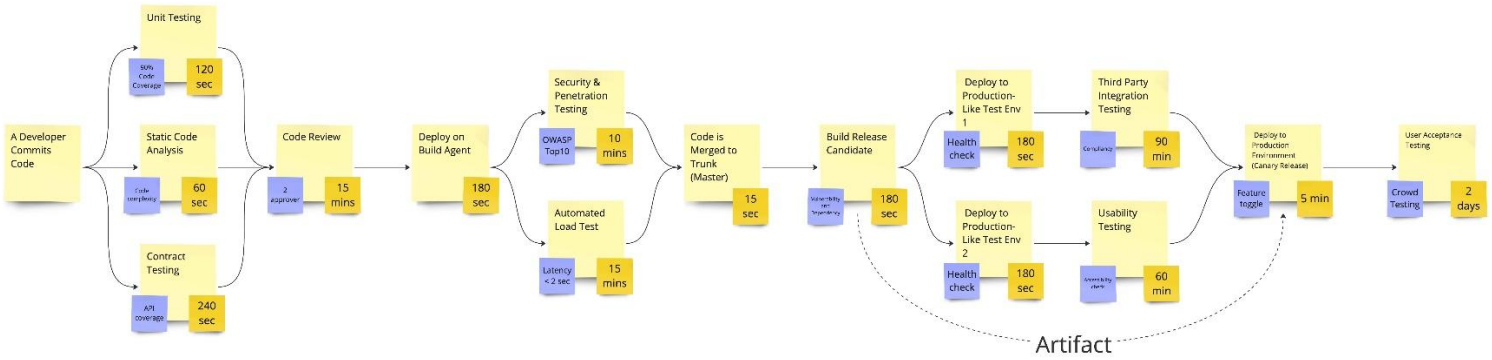

Figure 3: A test strategy represented as a high level pipeline

A test strategy is a vital document also when applying a DevOps way of working. DevOps is relatively new to many organizations, as such a common organization-wide test strategy will guide all those involved and explain where and how testing is performed in the pipeline. It will also prevent each project/pipeline from re-investing the wheel.

The test strategy should of course be fully aligned with the development strategy, as testing is an integrated part of the development pipeline. Some considerations (organized by topic) to be taken into account when developing a DevOps test strategy:

Test levels

- Test Levels are mainly represented as test activities/jobs in the CI/CD pipeline.
- The test strategy is often built around acceptance testing of the developed user stories, driven by Behavior Driven Development (BDD) practices. This requires continuous testing at each stage of the pipeline.

- The test strategy typically defines a test level like contract testing to mitigate the risks of the usage of open APIs, and to support Integration testing.

Low level testing

- Low level development and testing practices like pull request, merging, and code review practices shall be taken into consideration.
- Static testing and other low level test activities are usually mandated by the CI/CD tool and often cannot even be avoided (automatic check and tests as part of the pull-request for example)
- Based on the shift left paradigm emphasis is also put on unit testing, achieving an agreed level of code coverage, and integration testing.

Test Automation

- Continuous testing is seamlessly integrated into the software delivery pipeline and DevOps toolchain
- A test automation strategy is a crucial part of the DevOps test strategy.

Entry and Exit Criteria

- Entry Criteria are represented as Definition-of-Ready (DoR) for the DevOps teams. DoR shall mandate high quality test basis that serves as input for testing, for example Acceptance Criteria defined collaboratively by the team, usually in the form of a Given-When-Then statement. It is important that risks and test goals are well defined and agreed before achieving Ready state, so estimation covers also the testing needs
- Exit Criteria are split into the Definition-of-Done (DoD) for DevOps teams and the Quality Gate built in of every test job in the CI/CD pipeline. The DoD can be defined differently for different backlog levels (epic, feature, story) representing the fulfillment of testing, the confidence level reached, at different levels/stages. On top of successful acceptance testing, DoD usually contains expected testing activities to fulfil Non-functional Requirements, that serves as a constraint to all backlog items. In DevOps teams, 'DONE' state is only achieved when the backlog item is already deployed into production, and works based on monitoring of the system/service.

Test Types: Security Testing

- Identify which non-functional characteristics, e.g., security, performance, reliability or usability, are critical for success and their test approach.
- With the importance of security, Static Application Security Testing (SAST) Dynamic Security Analysis Tooling (DAST), Interactive Application Security Testing (IAST), Software Composition Analysis (SCA), penetration testing and API security testing could be mentioned as (part of) the security testing approach.

Test Environments

- Test environments are shown for the various testing stages in the CI/CD pipeline (for example, development, staging, pre-production) each representing a different focus. They are usually also aligned to the various identified test levels (e.g., unit, integration testing, system testing and acceptance testing). As DevOps is focusing on heavy automation of deployment and configuration practices for production environments, applying practices like configuration-as-code, it is obvious to use the same practices when designing and building test environments. With the use of cloud-based environments creating a test environment on demand, and deleting it after use is a common practice, resulting in a standardized, controlled, stable test environment. When utilizing deployment strategies for example Blue-Green deployment, the test environment of the latest version of our product could become the production environment after successful fulfilment of the quality gates and DoD. In this case the previous production environment is reused as the next test environment (switching purpose between the Blue and Green environments).

The lean test strategy document is often a good solution for DevOps organization and a way out from detailed (project) test plans. The TMMi specific goal Establish a Test Strategy, including its specific practices, is fully applicable to organizations applying DevOps software development. Like with the test policy, it is important to ensure that test strategy is distributed to the whole organization / value stream / Agile release train / tribe.

2.1.3 SG3 Establish Test Performance Indicators

The business objectives for test improvement, as defined in the test policy, need to be translated into a set of key test performance indicators. The test policy and the accompanying performance indicators provide a clear direction, and a means to communicate expected and achieved levels of test performance. The performance indicators must indicate the value of testing and test process improvement to the stakeholders. Since investments in process improvement need long term management support, it is crucial to quantitatively measure the benefits of an improvement program to keep them motivated. Beware that this TMMi specific goal is about defining a limited number (e.g., 2 or 3) test performance indicators. It is not about setting up and implementing a full measurement program but rather about defining a core set of indicators that tell you how the value of testing is changing over time and within different delivery environments.

As within Agile, also with DevOps the focus will be more team-based and systems thinking. This may result in a corresponding broadening of the indicators to development and deployment process performance indicators rather than being confined solely to the specifics of testing itself. The main DevOps performance indicators typically follow the DevOps Research and Assessment (DORA) Metrics. Note that instead of (test) coverage, change failure rate is measured during deployment to production (percentage of failed deployments).

Examples of DORA metrics:

- Software delivery performance metrics: deployment frequency, lead time for changes, time to restore service and change failure rate
- Operational Performance: reliability metrics, e.g., MTTR and MTTF

Performance indicators should be an integral part of the CI/CD activities, and the data should be available in almost real-time, and for everyone in the organization. On top of the DORA metrics, Flow metrics are also used to support Value Stream Management/optimization. Testing activities can be measured in line with the above metrics making sure their contribution is understandable.

Examples of flow metrics for Value Stream Management related to testing:

- Test execution lead time, Test activity success rate or failure rate within CI/CD per code change
- Flow metrics, how the Test Execution lead-time contributes to overall lead time

The challenge would be to define the appropriate blend of indicators relating to the DevOps approach and systems thinking while giving a good indication of the performance achievements of a TMMi-based test improvement programme. The TMMi specific goal Establish Test Performance Indicators, including its specific practices, is fully applicable although the performance indicators selected and applied may well have a larger scope and be more broader than being related to testing only. Of course, the latter will make the analysis and interpretation of performance indicators more challenging. In fact, the performance indicators being used may not be called test performance indicator. This is still ok in the context of TMMi as long as it has testing related elements and is being used to evaluate the progress being made doing test improvement.

2.2 Process Area 2.2 Test Planning

The purpose of Test Planning is to define a test approach based on the identified risks and the defined test strategy, and to establish and maintain well-founded plans for performing and managing the testing activities.

Beware that the key to successful test planning is in upfront thinking ("the activity"), not in defining the associated test plan ("the document").

For DevOps lifecycles, two kinds of planning typically occur, release planning and iteration planning. The Test Planning process area at TMMi level 2 focuses on the testing related activities at both release and iteration planning. Release planning looks ahead to the release of a product at the start of a project. Release planning requires a defined product backlog and may involve refining larger user stories into a collection of smaller stories. However, keep in mind that DevOps and Agile are usually working with fixed time lines and flexible content. It is important to make sure only those backlog items are released that have been tested in line with the test approach and DoD. Release planning provides the basis for a test approach and test plan spanning all iterations. Release plans are high-level, focusing on the overall strategy or approach rather than detailed tasks. After release planning is done, iteration planning for the first iteration starts. Iteration planning looks ahead to the end of a single iteration and is concerned with the iteration backlog.

2.2.1 SG1 Perform Risk Assessment

Exhaustive testing is impossible, and choices need always to be made and priorities need always to be set. This TMMi goal is therefore also applicable for DevOps projects. For DevOps projects, a high-level product risk assessment shall be performed based on a product vision document or set of high-level product goals at release planning. For each iteration a more detailed product risk session shall be performed based on the user stories or other requirements for that iteration as part of the iteration planning session. Sometimes this is already done at refinement sessions, where user stories discussed and are also analyzed for risks. The product risk assessment process in a DevOps project will have a much more lightweight format compared to those applied in a traditional project following a sequential lifecycle model. Examples of a lightweight product risk assessment techniques to be used are risk poker [Veenendaal12] and ROAM (Resolved, Owned, Accepted, Mitigated) [Baah]. At release planning or program increment planning (PI planning), business representatives who know the features in the release provide a high-level overview of the functionality to be developed, with the non-functional requirements as constraints, and the whole team, including the tester(s), will assist in the risk identification and assessment.

During iteration planning the DevOps team identifies and analyzes product risks based on the user stories to be implemented in the upcoming iteration. Preferably all members of the Agile team and possibly some other stakeholders participate in the product risk session. The result is a prioritized list of product risk items identifying the critical areas for testing. This in turn will help in determining the appropriate amount of test effort to allocate in order to cover each risk with enough tests, and sequencing these tests in a way that optimizes the effectiveness and efficiency of the testing work to be done. The estimated tasks can be prioritized based on the level of product risk associated with them. Tasks associated with higher risks should start earlier and involve more testing effort. Tasks associated with lower risks should start later and involve less testing effort.

With the importance of security in many organizations, threat modeling can, in addition to the above, be used to identify security requirements, pinpoint security threats and potential vulnerabilities, quantify threat and vulnerability criticality, and prioritize remediation methods to mitigate. Security threats are prioritized, using guidelines for quantifying the likelihood and impact of each threat to arrive at severity.

2.2.2 SG1 Establish a Test Approach

A test approach is defined to mitigate the identified and prioritized product risks. For a specific iteration, the items and features to be tested are identified during iteration planning. This activity is also based on the result of the product risk session. The prioritized list of items to be tested typically relates to the user stories to be tested in this iteration. The features typically relate, among others, to the various software quality characteristics to be tested. New product risks may become apparent during the iteration requiring additional testing. Issues like new product risks requiring additional testing are typically discussed at daily standup meetings.

With DevOps, the Test approach is defined based on Continuous Testing through the CI/CD pipeline. Continuous Testing means testing early and often using Automation with actionable feedback appropriate for each stage of the delivery pipeline. Continuous Integration (CI) and Continuous Deployment (CD) tools are typically invoked automatically upon code commit and orchestrate build, integration, testing, security, compliance, and deployment activities. The testing activities within the CI/CD pipeline usually build upon the testing levels identified using the Agile Testing Quadrants, and continuously optimized by the teams.

The test approach that is defined at iteration level or at backlog item level to mitigate the risks can cover for example additional reviewing of user stories and acceptance criteria, testing effort proportional to the level of risk, the selection of appropriate test technique(s) based on the level and type of risk. It can also suggest testing activities that are carried out only after the deployment, for example crowd testing in production to collect feedback on user experience, or testing performance with real user traffic using dark launch. In the context of DevOps, Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD) are popular techniques to drive the identification and design of test cases. The test approach at release level will be at a much higher level and shall be based on defined test strategy at program or organizational level. Often a test approach is held or displayed on the team/project wiki.

An important risk that is always apparent in iterative development, is the regression risk. The test approach needs to define how the regression risk is managed. Typically, this will be done by creating a specific regression test set, which is preferably automated. In this context the test automation pyramid is helpful [Cohn]. It shows how to maximize the value of regression test automation, starting with unit tests at the lowest level of the pyramid and moving on to service level testing. User interface testing sits at the very top. Unit tests are fast and reliable. The service layer allows for testing business logic at the API or service level, where you're not encumbered by the user interface (UI). The higher the level, the slower and more brittle testing becomes. Since automation is an important part of DevOps, test automation is of main focus within a DevOps test approach and not just to support regression testing. Another way to deal with the regression risk is to rely on observability of the system, continuously monitor its state and usage, and build automatic mechanisms to avoid risk, for example roll-back to previous, stable version.

Entry criteria (also referred to as Definition-of-Ready), normally a part of a defined test approach (specific practice 2.3) are likely to be handled differently in DevOps development. Within DevOps software development, testing is an integral part of the team process and a continuous activity. The formerly used entry criteria checklist is split between the Definition-of-Ready, for example having Acceptance Criteria defined as test scenarios using Given-When-Then statements, and checks within the CI/CD machine, for example the checks within a pull request, running a smoke test on a freshly deployed test environment before starting a larger test execution activity, or checking that the needed test data was also deployed. Consequently, a specific checklist or gateway to determine if testing can, or cannot, be started should be implemented in the CI/CD workflow, and fail the build on non-conformance, just like any other defect. This also applies for a component going within a DevOps team from one test stage (e.g., unit testing) to another (e.g., acceptance testing).

Test exit criteria (specific practice 2.4) are part of the so-called Definition-of-Done (DoD) that heavily relies on quality gates accomplished within the CI/CD pipeline. It is important that the DoD has specific test-related

criteria, e.g., for test coverage and product quality. The iteration should result in the implementation of the agreed set of user stories and meet the (test) exit criteria as defined in the DoD. In DevOps "DONE" usually means that the code changes related to the backlog item(s) are deployed into production, passing all quality gates within the CI/CD pipeline. In case Release is happening On-Demand, and detached from Deployment using technology like Feature Toggles, there is also a DoD at release level that could span multiple iterations, however the aim of DevOps is to release often, as small chunks as possible. The DoD at release level will again typically have coverage and product quality related criteria, usually focusing on validating that the expected value has been delivered to the users, based on testing activities happening in production, like crowd testing, or canary testing.

Specific practice 2.5 Define suspension and resumption criteria is likely to be non-relevant to DevOps lifecycles. As testing is an integral part of the DevOps software development process, it will of course not be treated as separate and independent activity from other iteration activities. When there are blocking issues that could be considered as potential or actual threats to the progress of testing, these are discussed in the daily standup meeting. In this forum the team will decide what actions, if any, need to be taken to solve the issues. Thus, formal suspension and resumption criteria will not be required and defined, issues around this are dealt with as part of the normal DevOps routine. The DevOps routine thus serves as an alternative practice for this specific practice.

2.2.3 SG3 Establish Test Estimates

For DevOps teams, detailed estimates for testing will be made during iteration planning. High-level (test) estimates are made during release planning and possibly also at backlog refinement sessions. All estimates are of course done as a team estimate which includes all effort required to deliver each story. It is important to ensure that testing activities are indeed considered during the estimation sessions. This can be done by having testing activities identified as separate tasks and subsequently estimating each individual test task, or by estimating each user story whereby the testing to be done is made explicit and thus considered. As in DevOps automation of almost everything is expected, automation of testing related activities shall also be estimated, including test automation, test environment deployment automation, test data deployment automation, as well as improvements of the pipeline in line with the identified test approach for the Iteration. During estimation, a tester, being part of the DevOps team, shall participate in the estimation sessions. Planning Poker, T- shirt sizing, swimlane sizing (bucket sizing) and dot voting are all typical estimation techniques being used at DevOps software development.

The work to be done for the next iteration is typically defined by the user stories. The user stories need to be small in size for them to be estimable. Estimable is one of the user story criteria defined by INVEST and is applicable for user stories that are to be part of an iteration. During iteration planning Agile teams will typically identify and define test-related tasks. Test tasks will be captured on a task board together with the other development tasks. At the retrospective, teams should look back on their estimation accuracy (if identified as a problem), and improve it based on learning from the size of the individual tasks done in the iteration.

At release planning, user stories or epics will typically be more high level defined and not yet detailed into specific tasks. This will of course make the estimations harder and less accurate. As stated, DevOps projects will not establish a work-breakdown structure as a basis for estimations but may at release level benefit from information on what and how to develop the Backlog Item that is enough to fulfill the needed estimation accuracy, that is enough to make the backlog item prioritization possible. As in DevOps the aim is to optimize the flow of value, the estimation needs to support the decision of what to build next, and not to create a full-blown project time-line.

Although a DevOps team will estimate in a relatively informal way, the rationale for the estimations should be clear (i.e. what factors are being considered). Discussions based on rationale promotes a higher level of

accuracy of the estimations. Typically, in DevOps projects the estimation is focused on size (using story points) or effort (using ideal man days as an estimation unit). Costs are normally not addressed as part of estimation sessions in DevOps projects. Note that infrastructure costs are typically high in DevOps context, their cost estimates are typically established in specific estimation sessions, not in team estimation sessions. Close monitoring, and control of infrastructure cost, for example public cloud usage, is needed, and can be embedded into the CI/CD pipeline, with automatic reporting. (for example every GitHub Action job executed is presented in vCPU minutes used).

This TMMi specific goal and its specific practices are therefore fully applicable with the exception of specific practice 3.2 Define test lifecycle. One of the basics of iterative and DevOps software development is to work in small chunks. Therefore, the tasks that have been identified are typically detailed enough to serve as a basis for (test) estimation. There is thus need in DevOps context to also define a lifecycle for testing activities to serve as an additional basis for estimation.

2.2.4 SG4 Develop a Test Plan

Re-stating the comment that test planning is about upfront thinking ("the activity") and not about defining the associated test plan ("the document"). In DevOps most of the test planning activities, as defined by this TMMi specific goal, will be performed during release and iteration planning, as well as in backlog item refinement sessions. However, the result of these activities will typically not be documented in a test plan, especially for iteration planning where they could be reflected on the task board. As testing is performed as an integral part of the release and iteration planning, the resulting "schedules" will also include testing activities. Rather than a detailed schedule such as developed with sequential lifecycles, the schedule within a DevOps project is much more like an ordering of user stories (backlog items) and tasks that reflects the release and iteration priorities, e.g., based on desired delivery of business value. Keep in mind that test execution is mainly happening within the CI/CD pipeline, triggered by code commit. Additional, manual testing, or activities done in Production can be planned on the Iteration board. Mind-maps are often used here as a supporting technique. The task board will reflect the iteration priorities. Thus, no explicit (test) schedule is established; it is expected that clear release and iteration priorities are defined for the user stories, respectively the tasks to be performed, including the testing tasks.

On a project level, test staffing is part of building the team; the need for test or multi-skilled resources is identified upfront. As projects change or grow, one can easily forget the rationale for the initial selection of an individual to a given team/project, which often includes specific skill needs or experience specifically related to the team/project. This provides a good rationale for writing down the skill needs of people, providing back-up information related to why they were selected for the team/project. Once the DevOps teams are defined test staffing is more or less fixed. During iteration planning the identification of the (additional) resources and skills, e.g., for non-functional testing, needed to perform the testing in an iteration can if necessary be discussed to ensure that the team has sufficient testing resources, knowledge and skills to perform the required tests.

An initial project risk session should be part of release and iteration planning. Identification (and management) of further project risks during the iteration is part of the daily standup meetings, and at the regular scrum-of-scrums, and are typically documented by means of an impediment log. It is important that testing issues are also noted in the impediment log. The impediments should be discussed at the daily standup meeting, until they are resolved, , or at the scrum-of-scrum meetings if the team has no level or authority to resolve them.

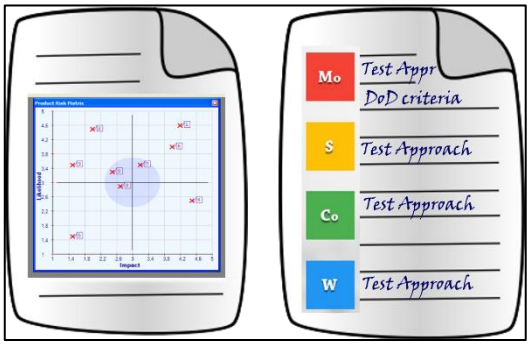

Figure 4: A One Page Test Plan

Although typically no specific and detailed test plan is developed and documented, specific practice 4.5 Establish the test plan is still relevant within a DevOps context. The result of the discussions that take place in the context of test planning are however likely to be captured in a lightweight form, possibly a mind-map or just a one-page document (see figure 4), a team level Agile Testing quadrants, and the testing activities within the CI/CD pipeline..

2.2.5 SG5 Obtain commitment to the Test Plan

Within DevOps the process to develop and establish a test approach and test plan is a team-based exercise, possibly lead by a test professional (being one of team members). Product quality is a team responsibility. As such, provided the team follows the correct process, and the agreed testing activities are embedded in the CI/CD pipeline, commitment to the (test) approach and (test) plan is already an implicit result of release and iteration planning as it is a team-effort. This of course is a huge difference with the way of working in a traditional environment where typically the tester prepares the test plan and then thereafter needs to obtain explicit commitment. The infrastructure resources needed for the CI/CD pipeline to implement the Test Plan in the form of deployed test environments and executed automatic tests needs to be taken into consideration and provided by the stakeholders.

In a DevOps context, the team (including product owner) must understand and agree on the prioritized list of product risks and the test mitigation actions to be performed. The understanding and commitment can for example be achieved by means of a short presentation followed by a discussion during release or iteration planning explaining the product risks, test approach and its rationale to the team.

During estimation session (see SG 3 Establish Test Estimates) the workload is estimated. The (test) resources for a team are fixed by the setup of the DevOps team. User stories that are estimated and selected to be developed take the available resources as a starting point (constraint). Thus, again reconciling work and resource levels is not a meaningful activity, however continuous focus on technical resources, like cloud compute/storage/network availability is key, and need to be provided to the teams. The specific practice 3.2 Reconcile work and resource levels is therefore a practice that is typically not relevant in a DevOps context. Daily stand-up meetings, and Scrum-of-Scrum meetings, will be used to address any resource issues during the iteration and to (re-)allocate appropriate resources immediately or remove a deliverable from an iteration to be reconciled in future iteration or release planning.

2.3 Process Area 2.3 Test Monitoring and Control

The purpose of Test Monitoring and Control is to provide an understanding of test progress and product quality so that appropriate corrective actions can be taken when test progress deviates significantly from plan and product quality deviates significantly from expectations.

There are some things that are fundamentally different to monitoring and control in DevOps context compared to traditional projects. Although monitoring and control are essential elements within DevOps it does not imply sticking to a rigid plan is the goal. From a test monitoring and control perspective this means we are not plan driven, but constantly reviewing our progress and results from testing, and adapting our plan and approach as appropriate, e.g., as new product risks become apparent.

The test plan, that is usually spread between fine-tuned acceptance criteria, backlog items (including their risk level) and tasks, is a living entity and continuously needs to be reviewed and updated as new information becomes available or feedback given. The target for testing activities are defined by the set of acceptance criteria for the given backlog item, the team's test approach being updated for the given backlog item, and at backlog level specific Definition-of-Done criteria are defined. DevOps projects also need to keep the 'bigger picture' in mind and monitor and control at CI/CD level as well as at iteration level.

It is important to note that as testing is a process that is fully integrated into the overall process of the DevOps team, test monitoring and control is also an integral part of the overall monitoring and control mechanisms of the DevOps team. As a result, testers do not report to a test manager as with traditional projects, but to the team.

2.3.1 SG1 Monitor Test Progress against Plan

DevOps teams utilize various methods and tools to monitor and record test progress, e.g., progression of test tasks and stories on the Agile task board, and burndown charts. These can then be communicated to the rest of the team using media such as wiki dashboards and dashboard-style emails, as well as verbally during stand-up meetings. Teams may use burndown charts both to track progress across the entire release and within each iteration. A burndown chart will typically show progress against expected velocity and backlog of features to be implemented (so, it shows the performance of the team). In addition, continuous monitoring with on-line dashboards will be used in DevOps to monitor the infrastructure, application and network in an automated way throughout the CI/CD pipeline. Test environment resources are continuously monitored against those agreed during iteration planning and for CI/CD pipeline usage. Another common practice is to track test environment issues via the task board, e.g., by using stickers marked 'blocked test environment' on the story cards or by creating a separate column on the board where all stories blocked by environments wait until unblocked. It is all about creating visibility of the impact of a test environment that is blocking progress, and turning the team's attention to fix this issue before working on any other task.

To provide an instant, detailed visual representation of the team's and CI/CD current status, including the status of testing, teams typically use task boards. The story cards, development tasks, test tasks, and other tasks created during iteration planning are captured on the task board. During the iteration, progress is managed via the movement of these tasks across the task board into columns such as "to do", "work in progress" and "done". To preserve flow, Work in Process (WIP) limits are often applied to different states on a Kanban Board, controlling team behavior to finish tasks (in progress) instead of starting to work on new tasks. DevOps teams typically use tools to maintain their story cards on task boards, which can automate dashboards and status overview. However, some teams do not create specific tasks for the individual activities but may just use the story card and annotate comments in this card for tests, referencing tools or wiki's where the testing may be documented. Testing tasks on the task board typically relate to the acceptance criteria defined for the user stories and the non-functional requirements that are a constraint for every backlog item. As test automation scripts and exploratory tests for a test task achieve the pass status, the task moves into the "done" column of the task board.

The Definition-of-Done (DoD) serves as exit criteria against which progress is being measured. The DoD should also be related to testing activities and shows all criteria that need to be satisfied before development and testing of a user story can be called 'Done'. Note that testing criteria related to the test tasks form only a part of what the team agrees to complete as the Definition-of-Done. Definition-of-Done criteria are typically applied at multiple levels, e.g., at story level and epic level. The whole team reviews the status of the task board regularly, often during the daily stand-up meetings, to ensure tasks are moving across the board at an acceptable rate. If any tasks (including testing tasks) are not moving or are moving too slowly, this than triggers a team-based discussion where the issues that may be blocking the progress of those tasks are analyzed.

The daily stand-up meeting involves all members of the DevOps team including testers. At this meeting, they communicate their current status and actual progress to the rest of the team. Any impediments that may block development or test progress are communicated during the daily stand-up meetings, so the whole team is aware of them and can act accordingly. In this way project risk management is integrated in these daily

meetings. On a regular basis the team may decide to perform a ROAMing exercise to support the monitoring of the status of project risks [ROAM]. Any project risk, including those for testing, e.g., the lack of availability of test environments, can be communicated and addressed during the daily stand-up. (Note, monitoring product risks is part of monitoring product quality and thus discussed hereafter with the specific goal SP 2 Monitor Product Quality Against Plan and Expectations.) Daily standup meetings for daily task management, and DevOps team practices related to task course correction are good proven techniques that meet the intent of TMMi specific practices in the Test Monitoring and Control process area.

At the end of the iteration a sprint review is held against the agreed sprint goals. The team demonstrates what has been achieved, for example what stories or epics are done. Integration through the CI/CD pipeline is of course continuous. The accomplishments of testing, e.g., against the Definition-of-Done, will be part of the iteration review. Demos are organized with stakeholders to discuss the business value and quality of the product being delivered. Stakeholders are represented by the product owner in iteration planning, iteration reviews (demos) and retrospectives. The product owner is involved through discussions on the product backlog and will provide feedback and input to the elaboration of user stories and the design of tests throughout the iteration. Other stakeholders are involved at the end of each iteration during the iteration review. No specific monitoring of stakeholder involvement is required as the representation of stakeholders by a product owner is embedded in the Agile way of working.

2.3.2 SG2 Monitor Product Quality against Plan and Expectations

For product quality monitoring largely the same mechanisms are used as for progress monitoring (see SG1 above). Product quality monitoring using on-line dashboard is not just done by teams during their development and testing, it is also a continuous process in which the quality of software products is monitored in production. Ops roles are continuously monitoring the quality of the product using observability, usually defined as Service Level Indicators (SLI's). Regression or deviation in SLI's trigger corrective actions to preserve quality and user experience. In DevOps, for product risk monitoring the focus lies on a review of list of product risks, including security risks/threats, in regular meetings rather than the review of any detailed documentation of risks. Newly identified product risks or changed product risks, e.g., as a result of exploratory testing, will be discussed and required testing actions will be agreed upon. When needed, an additional ROAMing exercise may be called upon [ROAM]. The status of the various product risks is typically shown by means of charts on the dashboard.

DevOps teams use defect-based metrics similar to those captured in traditional development methodologies, such as test pass/fail rates, defect discovery rates, defects found and fixed, to monitor and improve the product quality. The number of defects found and resolved during an iteration and those found with the continuous integration activities, as well as the number of unresolved defects possibly becoming part of the backlog for the next iteration should be monitored during the daily stand-up meetings. To monitor and improve the overall product quality, many DevOps teams also use customer satisfaction surveys to receive feedback on whether the product meets customer expectations.

Testing exit criteria, e.g., for test coverage and product quality, are part of the Definition-of-Done (DoD). The adherence to agreed exit criteria is typically monitored through the task board, whereby a story can only be indicated as "done" if it complies to its DoD criteria, which usually include the criteria that the given backlog item is deployed into production (it is "done" when it is deployed, and not the other way around). Typically, specific exit criteria are defined for continuous testing. Defects are monitored continuously as part of continuous testing within the CI/CD pipeline. The fulfillment of expected quality characteristics of the product

in production is monitored using observability. Automated quality gates (based on defined thresholds) are often implemented at different stages of the DevOps pipeline to block deployment when product quality does not meet expectations. These gates can assess metrics such as test pass rate, performance benchmarks, code coverage, and static code analysis results.

Performance and reliability are often important non-functional quality characteristics and as such are part of monitoring product quality. Performance testing (e.g., load and stress tests) can be integrated in the pipeline to monitor system performance against expected benchmarks. Alerts are triggered when performance thresholds are breached, notifying teams to address issues immediately. Also (log) monitoring tools can be used to track real-time error rates, memory leaks, and system crashes in production. Early identification of these issues can help ensure the product meets reliability expectations.

The daily stand-up meeting is the mechanism used to almost continuously perform product quality reviews. Demos are organized with stakeholders to perform validation and discuss the business value and quality of the product being delivered. Product quality is verified and validated against the defined DoD quality criteria.

Following the process area Test Planning, specific practices on monitoring entry, suspension and resumption criteria are likely to be non-relevant. For more information refer to the specific goal 2 Establish a Test Approach of the Test Planning process area, where an explanation has been given why these criteria are typically likely to be non-relevant in a DevOps environment.

2.3.3 SG3 Manage Corrective Actions to Closure

DevOps teams would very quickly notice issues such as deviations from expectations in a burn-down chart and/or lack of progression of (test) tasks and stories on the task board. Most importantly if the quality of the software in production is degrading it is noticed instantly, and usually triggers corrective actions, like roll-back to a known stable version. These and other issues, e.g., issues blocking test progress, are communicated during the daily stand-up meetings, so the whole team is aware of them. This than triggers a team-based discussion where the issues are analyzed. The team, together with the product owner, will decide on corrective actions to be taken. Managing corrective actions in DevOps based projects is primarily a responsibility of the self-organizing team. The team can define and implement appropriate corrective actions, or escalate any issues as ‘impediments’ to the Scrum master. Corrective actions for product quality are sometimes automated, and the root causes are monitored to avoid their re-occurrence. This is part of the continuous improvement culture of the DevOps team. The Scrum master typically has the responsibility to support the team to manage issues to closure. Typical events to discuss and manage corrective actions are daily stand-ups and retrospective meetings. Corrective actions that have been agreed can be managed to closure as ‘tasks’ or ‘backlog items’ via the product backlog or (within the iteration) via the task board.

2.4 Process Area 2.4 Test Design and Execution

The purpose of Test Design and Execution is to improve the test process capability during test design and execution by establishing test design specification, using test design techniques, performing a structured test execution process and managing test incidents to closure.

Although the underlying objective (“mitigate the risks and test the software”) is the same for a DevOps and a traditional sequential project, the approach taken on how to test is typically very different. In DevOps, flexibility and being able to respond to change are important starting points. Also, test analysis and design, test implementation and test execution are not subsequent test phases, but are rather performed in parallel,

overlapping and iteratively, supported by fast feedback cycles. The level of detail of test documentation established is another key difference, mainly relying on documentation as part of the test case automation code. With manual testing, typically experienced-based and defect-based techniques are used in DevOps projects, to complement test automation. Additionally, manual testing could be used in crowd testing, after deploying the software to production, and releasing it to a pre-selected group of people, who follow some instructions/test charters and provide additional feedback to development on dedicated quality characteristics, usually usability or accessibility. Specification-based test design techniques may also still be applicable and used, especially as part of the collaborative requirement analysis, listing possible example scenarios to enhance the User Story acceptance criteria. With more emphasis on unit and integration testing, white-box techniques such as statement and decision testing are also much more popular, as well as contract testing against APIs. A final major difference is the level of the regression risk resulting from a high change rate, which calls for more regression testing at the various test levels. Ideally, regression testing is highly automated, triggered and executed automatically within the CI/CD pipeline, in multiple test stages. There are many differences, but in the end, in the context of the process area Test Design and Execution it is always about creating tests, mitigating products risks, running tests and finding defects.

2.4.1 SG1 Perform Test Analysis and Design using Test Design Techniques

In DevOps, test analysis and design, and test execution are mutually supporting activities that typically run continuously embedded in the Agile way of working of the whole product team. In DevOps projects, testers are part of a team that collaboratively creates and refines user stories. Frequent informal reviews are performed while the requirements are being developed including acceptance criteria for each user story. These criteria are defined in collaboration between business representatives, developers, and testers. Typically, the tester's unique perspective will improve the user story by identifying missing details, alternative scenarios and making them testable. Test analysis is thereby not an explicit separate activity, but rather an implicit activity that testers perform as part of their role within collaborative user story development.

Based on the analysis of the user stories, test conditions are identified. From a testing perspective, the test basis is analyzed to see what could be tested – these are the test conditions [Veenendaal24]. Provided the defined acceptance criteria are detailed and clear enough, they may well take over the role of traditional test conditions. In DevOps context, often techniques like Behavior Driven Development (BDD) and Acceptance Test Driven Development (ATDD) using the Gherkin syntax, can help to define scenarios and identify conditions to test, documenting them as acceptance criteria of the given user story. Likewise with the Non-Functional Requirements specific test conditions are often defined in the format of acceptance criteria. The test conditions are subsequently translated into tests, which are needed to provide coverage of the defined and agreed acceptance criteria. Interesting test conditions can also be captured in test charters to be used with exploratory testing. Of course, the above is not only applicable at story level, but the same goes for testing at epic and feature level. Test conditions at epic and feature level are typically more abstract than those at user story level and span multiple user stories, or represent non-functional requirements. Specification based test design techniques are typically helpful in deriving test conditions from user stories and acceptance criteria. However, in DevOps context most often these test design techniques are used more implicitly than explicitly, whereby the testers, based on their experience, have mastered the techniques and are able to use them with flexibility in context.

With the test-first principle being applied with DevOps, tests that cover the set of test conditions will be identified (and possibly automated) prior to, or at least in parallel with the development of the code. This approach sometimes already starts with backlog refinement. For automated unit testing an approach like Test-

Driven Development (TDD) can be considered. For higher test levels, as already stated Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD) are popular approaches that are also highly related to test automation.

For most manual testing, tests will be identified/refined as the team progresses with test execution. Tests are most often not documented in as much detail as in traditional projects, but rather in a format of test ideas, using exploratory testing. For complex and critical areas, a more traditional approach to test design and test case development (using formal test design techniques) may be the best way to cover the risks. However, also with this approach the amount of documentation will be limited compared to testing in a traditional sequential lifecycle environment. The prioritization of tests follows the prioritization of the user story they are covering. The prioritization of the user stories is based on business value; the highest priority relates to highest business value. It is important that tests are established to cover functional and non-functional risks, but especially in DevOps it is also specifically important to cover the regression risk. Test priority is implemented by adding tests to CI/CD pipeline jobs.

Specific test data necessary to support the test conditions and execution of tests is identified. However, in DevOps, in contrast to traditional environments, the test data needed is typically not first specified as part of a test specification document. Test data are rather provided as soon as the necessary tools and/or functionalities are available. The test data are immediately created to allow an almost immediate start of the execution of manual tests. For automated tests, however, the data will typically be needed to be specified upfront. Test data is usually managed in the same way as test environments and can be deployed to these environments on demand. Test data preparation tools can anonymize, randomize and mask production data as needed by privacy and GDPR regulations.

Traceability between the requirements, test conditions and tests need to be established and maintained. Teams need to make it clear that they have covered the various user stories and acceptance criteria as part of their testing. The team therefore needs tool support to have their requirements (user stories) organized and managed in a way that supports the assignment of identifiers to each user story so that those identifiers can be used to ensure that testing is completed according to the agreed criteria. Typically tests and user stories linked within a tool provide real-time coverage measurement possibilities. Test to story traceability is visible through the CI/CD pipeline test job status and log, automatically generated.

2.4.2 SG2 Perform Test Implementation

Test implementation is about getting everything in place that is needed to start the execution of tests. Typically, the development of test documentation to support test execution is minimized. Instead, automated (regression) test scripts are developed and prioritized. Regression test preparation starts as soon as possible in parallel to other testing activities.

In a DevOps environment detailed test procedures for manual testing are not common practice. Working in a knowledgeable team, tests will most likely be documented at a much higher level of abstraction, e.g., test charters with exploratory testing. This will suffice for those executing the tests since it is expected that they have the required level of domain and technical knowledge to perform the tests. Those executing the tests work within the team, so they will have a better understanding of both of what and how it is coded, and how the functionality is fit for purpose. Since the level of change within and across iterations is typically high, developing detailed test procedures would also create much maintenance work. Typically, much manual testing is performed by using exploratory testing. In exploratory testing, no detailed test procedures are developed, but rather high level one-page test charters will be developed, describing test ideas and guidance

for the tester. However. most DevOps teams use automated testing. Typical approaches being used are TDD, BDD and ATDD. While BDD itself is not test automation, it can serve as the foundation for automating acceptance tests. The scenarios written in BDD (in syntax like Gherkin) are typically linked to test automation frameworks. Using one or more of these approaches automated test scripts will be created, e.g., to detail the Gherkin scenarios, as part of the test implementation activity. The test scripts also serve as the test documentation, having enough comments for understanding, and by using good coding practices. Typically, test suites are created to group the developed test scripts. These suites are subsequently linked to test jobs in CI/CD pipeline.

Specific test data, as specified during the test analysis and design activity, is created preferably in re-deployable format. Test Data is sometimes set up as code, whereby test data is defined in the format of schema files and rules. An intake test is specified to be implemented within the CI/CD pipeline. This test, sometimes called the confidence or smoke test, is used to decide at the beginning of every test execution job in the pipeline whether the test object is ready for detailed and further testing. The smoke test is intended to provide instant feedback to the development team and stakeholders on failure of the test. In addition, an automated deployment health checklist is often defined for every test environment/stage within the CI/CD pipeline. The tasks related to test data preparation, test environment changes or test automation implementation are estimated and planned on the teams task board during iteration planning.

Regarding specific practice 2.4 Develop test execution schedule, Automated test execution is triggered by change of any code within the CI/CD pipeline. Triggering scenarios are identified and coded (web-hooks) to initiate a test execution on change. Manual test executions are scheduled in line with iteration planning, managed as test tasks through the task board and discussed during the daily standup meetings.

2.4.3 SG3 Perform Test Execution

Tests are executed according to the specified CI/CD pipeline jobs on each change, and for new stories being developed by the team following the iteration plan. The order of the test jobs is defined to provide fast feedback. If enough resources are available test jobs can be executed in parallel to amplify feedback. Cloud costs and sustainability are key to optimize the CI/CD pipeline. Defects/incidents are reported, and test logs are generated. In a DevOps project, software is delivered to production as often as possible, triggered by code changes. Testing is an integral part of the process whereby work is performed in cooperation to deliver a quality product. Smoke tests are executed as part of the testing jobs defined in CI/CD pipelines before larger test sets are executed. In case of failure, the execution of the pipeline is not continued to the next job, and feedback is given to development. In case the cause of the failure is identified as a defect, a defect is reported.

Test Execution is done in line with the priorities defined during iteration planning. Some tests may be executed using a documented test procedure, but typically many tests will be executed using exploratory and session-based testing as their framework. The exploratory test sessions are based on predefined test charters and are time-boxed. Tests can also be executed in production, before the software is released to all users. In case of crowd testing, a selected set of beta testers give feedback on some quality aspect of the product before mass release. This requires that the features can be switched on and off (feature toggle) based on predefined prerequisites, like user or geographic location.

With DevOps development there is an increased need to organize and structure regression testing. Regression testing at especially unit and integration test level is part of a continuous integration process. It is basically an automated “build and test” process that takes place on every code change, and detects defects early and quickly. Continuous integration allows for running automated regression tests regularly and providing quick

feedback to the team on the quality of the code and code coverage achieved. Tests are also executed according to the specified CI/CD pipeline jobs triggered by code change. These automated test scripts will provide test results and test logs. Aggregated test logs will typically provide views per epic, users story, test case, test suite, and test job, and for the whole pipeline.

Discrepancies between the actual and expected results are reported as defects. Defect management is usually an integral part of the development system, and provides full traceability to execution and code changes. In addition, test logs are provided automatically through all systems to provide a chronological record of relevant details about the execution of the tests.

With new user stories being developed and tested by the team, there is typically a discussion within Agile projects whether all defects found should indeed be logged. Particularly, in the case where the team is co-located, testers need to talk with developers, and defects that can be fixed immediately and included within the next build may not need to be logged, they just need to be fixed. Some teams only log defects that escape iterations, some log them if they can't be fixed today, some only log high priority defects. If not all defects found are logged, criteria must be available to determine which defects should be logged and which ones shouldn't.

It is considered a good practice also in DevOps teams to log data during test execution to determine whether the item(s) tested meet their defined acceptance criteria and can indeed be labeled as "done". Test data logging should be supported by automation in use. This is also true when applying experienced-based techniques such as exploratory testing. Examples of information that may be useful to document are: test coverage (how much has been covered, and how much remains to be tested), observations during testing (e.g., do the system and user story under test seem to be stable), risk list (which risks have been covered and which ones remain among the most important ones), defects found and other issues, and possible open questions. The information logged should be captured and/or summarized into some form of status management tool (e.g., test management tools, task management tools, task board), in a way that makes it easy for the team and stakeholders to understand the current status for all testing performed. Of course, it is not just the status of testing tasks in isolation that the team needs a status on, rather it needs to know the overall status of the user story. A backlog item is complete when it meets the Definition-of-Done. When a DevOps team uses Continuous Deployment as a practice, the Definition-of-Done usually includes successful deployment as a criterion, as well.

2.4.4 SG4 Manage Test Incidents to Closure

The term "incident" in DevOps is typically used to indicate a real production incident where users are affected. Therefore, we will use the term "defect" here in the remainder of this paragraph to indicate a test incident in the context of TMMi. As DevOps teams are also responsible for Operations, they have to react quickly. Incident management within DevOps is an IT Service Management (ITIL) process: "The practice of minimizing the negative impact of incidents by restoring normal service operation as quickly as possible."

As in DevOps almost everything is codified all defects found are actually defects in some software. If it is an environment issue, it is a defect in Infra as Code of the environment, if it is wrong test data, it is a test automation code defect, if it is other execution unreliability, it can be a pipeline code defect, etc. As stated in the previous paragraph, in DevOps environments, not all defects found will be logged. This specific goal only applies to those defects that are logged and thus need to be managed to closure. In principle, managing incidents in DevOps is simple. In case where the incident is logged on the task board, it is visualized as a defect

blocking a story and its tasks from getting completed. The standard DevOps way of working will be applied as with any other impediment or task that is blocking progress. It will be discussed during stand-up meetings and assigned to be resolved by the team.

In case the decision is made to defer a defect found to another iteration, they become merely another desired option (or change) for the product. Add them to the backlog and prioritize accordingly. When the priority is set high enough, they will be picked up by the team in the next iteration.

It is a good practice to monitor the number of defects found and resolved during an iteration as well as the number of unresolved defects possibly becoming part of the backlog for the next iteration during the daily (stand-up) meetings and at retrospective meetings.

2.5 Process Area 2.5 Test Environment

The purpose of Test Environment is to establish and maintain an adequate environment, including test data, in which it is possible to execute the tests in a manageable and repeatable way.

With the process area Test Environment an adequate test environment is established and maintained, including generic test data, in which it is possible to execute the tests in a manageable and repeatable way. Of course, also in a DevOps software development project, an adequate test environment is indispensable. Using cloud technologies for every CI/CD pipeline stage, an environment can be set up on demand in a controlled state. Because of the short cycle times of an iteration, the test environment needs to be highly stable and available. Problems in the test environment, e.g., in the CI/CD pipeline, will always immediately have an impact on the progress and results of the iteration. Properly managing the configuration and changes to test environment and test data is therefore also of utmost importance.

2.5.1 SG1 Develop Test Environment Requirements

Specification of test environment requirements is performed early in the project, and continuously thereafter based on continuous learning. It is important to understand the product value stream, production environment, technical landscape and development tool chain in use to determine the environment needs and subsequently develop requirements that drive environment implementation. The requirements specification is reviewed to ensure its correctness, suitability, feasibility and accurate representation of a 'real-life' operational environment. Early requirements specification has the advantage of providing more time to acquire and/or develop the required test environment based on virtualization, containerization and other cloud native aspects that are often used in DevOps context. Environments in the build pipeline (the CI part of the pipeline) are usually deployed in a small and simple scale and continuously developed in line with growing performance and reliability expectations. However, in the release pipeline (the CD-part of the pipeline) there may be a need for an end-to-end-business-process test, and as such a corresponding much more complex test environment.

Test environments' needs are identified and specified during refinement sessions, in line with infrastructure-as-code and configuration-as-code practices and standards as defined by CI/CD playbooks or by a platform team. The priority of the test environment and data requirements are based on the corresponding backlog item priority. Sometimes a separate technical spike with acceptance criteria is specified to define more complex environment requirements. Requirements are typically written in Infrastructure/configuration-as-code compliant languages, following the organization CI/CD practices and terminology. The requirements are

mapped to the value stream or CI/CD pipeline stages. Development tool chains are aligned to the operational and deployment technology to ensure they are necessary and complete.

2.5.2 SG2 Perform Test Environment Implementation

Implementing the deployment pipeline and the production and test environment should start as soon as possible to have an initial version of the environments available upon the start of the first iteration. Test environments are production-like environments including generic test data to perform the relevant tests levels in the given CI/CD test stages. Efficient DevOps teams reuse production automation for test environments as well, deploying them on demand. All tasks related to environment implementation should be handled within the backlog as any other backlog item.

In DevOps context infrastructure-as-code is often used to define and implement environment components and configuration-as-code to define and implement environment variables and configurations, both via automated scripts and pipelines. It is often important to have the capability for on-demand environment provisioning, using self-services within CI/CD pipelines, also for exploratory test sessions. Generic test data for coding environments is created using automated test data generation and management tooling. A smoke test is typically run within the pipeline after deploying an environment instance and before the environment is used for additional activities.

2.5.3 SG3 Manage and Control Test Environments

Management and control of the test and production environments will take place throughout a DevOps project. Environments are managed and controlled to allow for uninterrupted test execution using available tooling to allow creation, configuration, allocation and use of environments and data. Systems management, performed on test environments to support provisioning, configuration, allocation, usage and decommissioning, consists of the following practices:

- manage infrastructure through code, automation and monitoring
- manage access to the test environment by providing log-in details
- provide technical support on progress disturbing issues during test execution
- provide monitoring and telemetry, which can be used to analyze test results and defect causation.

Also provisioning, allocation, amendment and use of test data is managed and automated where possible, to support the test process.

The creation and allocation of test environments is automated, coordinated and measured to achieve maximum availability and efficiency. Self-service or on-demand environment provisioning (spin-up/down) and monitoring of environment instances, especially for persistent environments are implemented. Problems that occur when provisioning or using test environments are logged and managed to closure by the DevOps teams. Test environment incidents are logged when a problem is observed, either automatically as part of failed tests or by the environment users.

As a main conclusion for the process area Test Environment, the specific goals and practices are all still applicable and do not change in essence.

References

[Baah] A. Baah (2017), Agile quality assurance, Bookbaby

[Beyer] B. Beyer, C. Jones, J. Petoff and N.R. Murphy (2016). Site Reliability Engineering: How Google Runs Production Systems, 1st. O'Reilly Media, Inc.

[Cohn] M. Cohn (2009), Succeeding with Agile: Software Development using Scrum, Addison-Wesley

[DORA16] A. Brown, N. Forsgren, J. Humble, N. Kersten and G. Kim (2016), 2016 State of DevOps Report, Puppet Labs, DORA (DevOps Research & Assessment]

[DORA24] DORA [2024], DORA's software delivery metrics: the four keys, https://dora.dev/guides/dora-metrics-four-keys/, DORA (DevOps Research & Assessment]

[Veenendaal12] E. van Veenendaal (2012), PRISMA: Product Risk Assessment for Agile Projects, in: Testing Experience, Issue 04/12, December 2012

[Veenendaal24] Erik van Veenendaal, Rex Black, and Dorothy Graham (2024), Foundations of Software Testing - ISTQB Certification (5th edition), Cengage

[Wake] B. Wake (2003), INVEST in Good Stories, and SMART Tasks, xp123.com/articles/invest-in-good-stories-and-smart-tasks