Certified Tester
Quality in DevOps (CT-QDO)
Syllabus

v1.0

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
v1.0	2026/04/17	General Release

Table of Contents

Acknowledgments

This document was formally released by the General Assembly of the ISTQB® on <date>

It was produced by a team from the International Software Testing Qualifications Board: Vipul Kocher (Chair Specialist Technologies and Approaches Working Group), Tamás Horváth (Vice-Chair Specialist Technologies and Approaches Working Group), Szilárd Széll (product owner), Kari Kakkonen, Rik Marselis, Katja Obring, Tal Pe'er, Daniel Poľan, Yaron Tsubery, Tomas Tumasonis, Aneta Derková

The following persons in alphabetical order are the authors of this syllabus: Kari Kakkonen, Rik Marselis, Katja Obring, Szilárd Széll, Yaron Tsubery.

The following persons in alphabetical order, were important contributors to this syllabus: Mantas Aniulis, Aneta Derkova, Matthias Hamburg, Tamas Horvath, Vipul Kocher, Gary Mogyorodi, Tal Pe'er, Daniel Poľan, Patrick Quilter, Jean-Francois Riverin, Tomas Tumasonis and Galit Zucker.

The team thanks the following organizations for contributing to the content of this syllabus: ASTQB, AT*SQA, DevOps United.

The team thanks Gary Mogyorodi for the technical edit and the review team and the Member Boards for their suggestions and input.

The following persons participated in the reviewing, commenting, and balloting of this syllabus: Mariam Abdellatif, May Abu-Sbeit, Gergely Agnecz, Amer Alatrash, Laura Albert, Chris van Bael, Jürgen Beniermann, Earl Burba, Tamás Csákó, Wim Decoutere, Walter Eder, Raymond Gillespie, Ole Christian Hansen, Josephine Irungu, Mattijs Kemmink, John Kurowski, Jędrzej Kwapiński, Anders Larsen, Roberta Lippelli, Sebastian Malyska, Vincenzo Marrazzo, Gary Mogyorodi, Frank Neirynck, Ingvar Nordström, Giorgio Pisani, Andrew Pollner, Nishan Portoyan, Meile Posthuma, Patrick Quilter, Miroslav Renda, Piet de Roo, Adam Roman, Jan Sabak, Adam Scierski, Márton Siska, Péter Sótér, Benjamin Timmermans, Giancarlo Tomasig, Stephanie Ulrich, Linda Vreeswijk, Dominik Weber, Geoffrey Wemans, Claude Zhang

Special alignment

TMMi® and the ISTQB® have entered an alliance to further promote the Software Testing profession together. During the development of this document, the TMMi® technical committee and the ISTQB® Certified Tester Quality in DevOps syllabus working party have worked together. TMMi® is a test improvement model. It provides a pre-defined improvement approach with priorities based on the TMMi® model structure. In the "TMMi® in the DevOps world" document ((E. v. Veenendaal, 2025)), the focus is on providing an interpretation of the TMMi® improvement goals for those working in a DevOps context. It does not provide a detailed description of good DevOps engineering practices. The ISTQB® Certified Tester Quality in DevOps syllabus is a content-based document. It aims to provide a detailed description of good DevOps engineering practices, e.g., test activities. The two documents are, therefore, highly complementary. TMMi® identifies which processes need improvement, while the ISTQB® Certified Tester Quality in DevOps syllabus outlines engineering best practices for implementation.

0 Introduction

0.1 Purpose of This Syllabus

This syllabus forms the basis for the International Software Testing Qualification for the Certified Tester Quality in DevOps (CT-QDO) syllabus. The ISTQB® provides this syllabus as follows:

1. To member boards, to translate into their local language, and to accredit training providers. Member boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
2. To certification bodies, to derive examination questions in their local language adapted to the learning objectives for this syllabus.
3. To the training providers to produce courseware and determine appropriate teaching methods.
4. To certification candidates, to prepare for the certification exam (either as part of a training course or independently).
5. To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

0.2 The Certified Tester Quality in DevOps

The ISTQB® Certified Tester Quality in DevOps (CT-QDO) certification supports people involved in software development based on DevOps in applying proper quality and test activities to achieve the right quality level for their solutions.

The CT-QDO certification is designed for professionals engaged in quality and testing within a DevOps environment. The idea is to provide DevOps specialists with the opportunity to acquire an internationally recognized certification in the field and to establish the specific advantages of certified DevOps quality specialists through specific skills in DevOps quality and testing methodology.

DevOps has become the mainstream way of modern software development in many domains. It extends Agile software development concepts of autonomy, frequent deliveries, and a focus on early customer feedback to achieve the right quality of the software. Quality assurance (QA) and testing have a strong emphasis on the DevOps culture. In a DevOps setting, quality engineering (QE), including QA and testing, takes place during all software development lifecycle (SDLC) phases. DevOps covers concepts such as continuous delivery (CD) and release on demand, which software testers should understand to develop effective quality and test practices.

0.3 Career Path for Testers

The ISTQB® scheme supports testing professionals at all stages of their careers, offering both breadth and depth of knowledge.

The Certified Tester Quality in DevOps builds on the qualification of an ISTQB® Certified Tester Foundation Level (*ISTQB® CTFL*, v4.0). The ISTQB® Agile related syllabi and ISTQB® Test Automation related syllabi are useful for understanding this syllabus. Where the ISTQB® Certified Tester Foundation Level syllabus provides the basic knowledge and competencies in software testing, the ISTQB® Agile

related syllabi expand on the Certified Tester Foundation Level syllabus and explain how testing is performed in an Agile team, and the ISTQB® Test Automation related syllabi explain how test automation is built and utilized for testing.

As this syllabus focuses on the DevOps culture, it expands the ISTQB® product portfolio towards DevOps as software development goes in this direction bringing benefits like faster time to market, happier customers, and better quality.

Individuals who achieve the ISTQB® CT-QDO certification may also be interested in the other ISTQB® Specialist certifications, especially the ISTQB® Certified Tester Foundation Level Agile Tester (*ISTQB® CT-AT*, v1.1), the ISTQB® Certified Tester Agile Technical Tester (*ISTQB® CT-ATT*, v1.1), and the ISTQB® Certified Tester Agile Test Leadership at Scale (*ISTQB® CT-ATLaS*, v2.0), as well as the ISTQB® Certified Tester Test Automation Strategy (*ISTQB® CT-TAS*, v1.0) syllabus, and also, ISTQB® Certified Tester Test Automation Engineer (*ISTQB® CTAL-TAE*, v2.0).

0.4 Business Outcomes

This section lists the business outcomes expected of a candidate who has achieved the CT-QDO certification.

A CT-QDO certified person can:

Code	Description
QDO-BO1	Explain How Quality Assurance Is Supported by and Contributes to the DevOps Concepts
QDO-BO2	Understand the Concepts of Implementing DevOps in an Organization
QDO-BO3	Contribute to All Value Stream Stages to Assist in Quality Engineering
QDO-BO4	Contribute to the DevOps Loop Implementation
QDO-BO5	Describe How Automation Supports Quality Assurance in DevOps
QDO-BO6	Implement Test Automation in DevOps Teams
QDO-BO7	Implement Manual Testing in DevOps Teams
QDO-BO8	Select Tools Supporting DevOps Within the Organization
QDO-BO9	Understand Technologies and Practices to Support Quality in DevOps

0.5 Learning Objectives, Hands-on Objectives, and Cognitive Level of Knowledge

Learning objectives and hands-on objectives support the business outcomes and are used to create the Certified Tester Quality in DevOps (CT-QDO) exams.

In general, all contents of this syllabus are examinable, except for the Introduction, Hands-on Objectives and Appendices. The exam questions will confirm knowledge of keywords at the K2 level (see below) or learning objectives at the respective level of knowledge. The specific learning objectives and their levels of knowledge are shown at the beginning of each chapter and classified as follows:

- K1: Remember

- K2: Understand
- K3: Apply

Further details and examples of learning objectives are given in *Section 7*.

For all terms listed as keywords just below chapter headings, the correct name and definition from the ISTQB® Glossary (ISTQB, 2025) shall be understood (K2), even if not explicitly mentioned in the learning objective.

The specific Hands-on Objectives, are shown at the beginning of each chapter. The level of an HO is classified as follows:

- H0: This can include a live demonstration of an exercise or a recorded video. Since this is not performed by the trainee, it is not strictly an exercise.
- H1: Guided exercise. The trainees follow a sequence of steps performed by the trainer.
- H2: Exercise with hints. The trainee receives an exercise with hints to solve it within the allotted time.
- H3: Unguided exercises without hints.

0.6 The Certified Tester Quality in DevOps Certification Exam

The CT-QDO certificate exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction, Hands-on objectives, and Appendices. Standards and books are included as references, but their content is not examinable, beyond what is summarized in the syllabus itself from such standards and books.

Refer to the Exam Structures and Rules document (*ISTQB® Exam Structures and Rules*, v1.2) for further details.

The entry criteria for taking the ISTQB® Certified Tester Quality in DevOps exam is that candidates have

- An ISTQB® Certified Tester Foundation Level certificate
- An interest in software testing and DevOps

Candidates are also strongly encouraged to:

- Have at least a minimal background in either software development or software testing, such as six months' experience as a system or user acceptance tester or as a software developer.
- Take a course that has been accredited to ISTQB® standards (by one of the ISTQB-recognized member boards).

0.7 Accreditation

An ISTQB® Member Board may accredit training providers and training material owners whose course material follows this syllabus. Training providers should obtain accreditation guidelines from the Member Board or the body that performs the accreditation. An accredited course is recognized as conforming to this syllabus and is allowed to have an ISTQB® exam as part of the course.

The accreditation guidelines for this syllabus follow the general Accreditation Guidelines (*ISTQB® Generic Accreditation Guidelines*, v2.4) published by the Processes Management and Compliance Working Group of the ISTQB®.

0.8 Handling of Standards

International standardization organizations like IEEE and ISO have issued standards associated with quality characteristics and software testing. Such standards are referenced in this syllabus. The purpose of these references is to provide a framework (as in the references to ISO 25010 regarding quality characteristics) or to provide a source of additional information if desired by the reader.
Please note that the ISTQB® syllabi uses the standard documents as a reference. Standard documents are not intended for examination. Refer to *Chapter 10* for more information on standards.

0.9 Level of Detail

The level of detail in this syllabus allows internationally consistent courses and exams. In order to achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Certified Tester Quality in DevOps syllabus
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area of software testing; it reflects the level of detail to be covered in CT-QDO training courses.

The syllabus uses the terminology (the name and meaning) of the terms used in QA and testing according to the ISTQB® Glossary.

0.10 How This Syllabus Is Organized

There are four chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter; timing is not provided below the chapter level. For accredited training courses, the syllabus requires a minimum of 20.2 hours of instruction divided over at least three days. The minimum training time distributed across the four chapters is as follows:

- Chapter 1: 140 minutes, DevOps Fundamentals
  - Explain How Quality Assurance is Supported by and Contributes to the DevOps Concepts
  - Understand the Concepts of Implementing DevOps in an Organization
- Chapter 2: 360 minutes, Quality Assurance and Testing in DevOps
  - Contribute to All Value Stream Stages to Assist in Quality Engineering

  - Contribute to the DevOps Loop Implementation
- Chapter 3: 510 minutes, Automated and Manual Tasks in DevOps
  - Describe How Automation Supports Quality Assurance in DevOps
  - Implement Automated Testing in DevOps Teams
  - Implement Manual Testing in DevOps Teams
- Chapter 4: 200 minutes, Tools and Practices in DevOps
  - Select Tools Supporting DevOps Within the Organization
  - Understand Technologies and Practices to Support Quality in DevOps

1 Fundamentals of DevOps - 140 minutes

Keywords

quality assurance, testing

Quality in DevOps Specific Keywords

CALMS, change fail percentage, change lead time, cross-functional DevOps team, deployment frequency, DevOps, failed deployment recovery time, feedback loop, flow, service level agreement, service level indicator, service level objective, site reliability engineering, value stream, wall of confusion

Learning Objectives of Chapter 1:

1.1 Explain How Quality Assurance is Supported by and Contributes to the DevOps Concepts

QDO-1.1.1 (K1) Recall key concepts of DevOps

QDO-1.1.2 (K1) Recall the wall of confusion concepts in DevOps

QDO-1.1.3 (K2) Explain DORA metrics of delivery performance and operational performance

QDO-1.1.4 (K2) Differentiate the elements of CALMS in terms of quality assurance

QDO-1.1.5 (K2) Explain how quality assurance supports the three ways of DevOps

QDO-1.1.6 (K2) Explain the benefits, risks, and pitfalls of DevOps

1.2 Understand the Concepts of Implementing DevOps in an Organization

QDO-1.2.1 (K2) Distinguish the DevOps team roles

QDO-1.2.2 (K2) Explain the concept of site reliability engineering (SRE) related to DevOps

QDO-1.2.3 (K2) Distinguish the different DevOps team patterns and anti-patterns

Hands-On Objectives:

1.2 Implementing DevOps in an Organization

QDO-1.2 (H0) Summarize How DevOps Principles are Visible in a Case Study

1.1 Quality Assurance (QA) within DevOps

DevOps is the combination of cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity, thus evolving and improving products at a faster pace than organizations using sequential development models and infrastructure management processes (Amazon, 2024).

DevOps first came to attention in 2008 when Patrick Debois (Debois, 2011) invited like-minded individuals to a conference to discuss how the delivery of software, and infrastructure, could use Agile software development. The term DevOps emerged after the 2009 Velocity Conference introduced concepts of rapid, frequent, and successful delivery (Debois, 2011).

This chapter discusses the DevOps concepts and how quality engineering (QE), including quality assurance (QA) and testing, relates to them. QA refers to all activities focused on providing confidence that quality requirements will be fulfilled, i.e., throughout the software development lifecycle (SDLC). QE refers to all quality-related activities that contribute to creating software that meets the customer's needs. DevOps views software development as a value stream guiding software from concept to market launch (*ISTQB® CT-ATLaS*, v2.0). This chapter also explains DevOps teams, meaning any team that applies DevOps practices.

1.1.1 Key Concepts of DevOps

DevOps stands for "Development and Operations". It refers to people working together to build, deliver, and run software. Under DevOps, the software development process should, in addition to writing code, also consider other aspects of development such as analysis, continuous integration (CI), functional testing, non-functional testing, especially security, deployment, delivery, release, infrastructure management, maintenance, and production monitoring. Software development success should be measured based on the delivery outcomes, and on the use of the right quality software.

DevOps is a culture promoting collaboration, communication, and continuous improvement, rather than a framework, standard, role, function, separate team, tool, or goal (DASA, 2024). DevOps strongly advocates automation and monitoring at all phases of software delivery. This often comes in the form of continuous integration/continuous delivery (CI/CD) pipelines. DevOps promotes autonomy, leadership, collaboration, and innovation, all while driving business success through shorter delivery cycles, increased deployment frequency, and reliable high-quality releases that meet customer needs.

The DevOps practices are based on the following principles (DASA, 2019):

1. Customer-centric action: Have short feedback loops with real customers.
2. Create with the end in mind: Explicitly focus on building working products delivered to customers.
3. End-to-end responsibility: Create DevOps teams that design, build, test, and run software.
4. Cross-functional autonomous teams: Create DevOps teams with a variety of skills, including testing, to be autonomous.
5. Continuous improvement: Embrace a culture of improving the product, process, and people's skills every day to adapt based on feedback from experimentation.
6. Automate everything you can: Strive for an automated CI/CD pipeline that delivers with short cycle times.

DevOps is often represented as an infinity loop, with both manual and automated steps. The typical DevOps loop includes several steps that could be grouped as continuous discovery, CI, CD, continuous deployment, and release on demand. It is described in *Section 2.2*, and multiple variations of the DevOps loop exist.

1.1.2 Breaking the Wall of Confusion

The wall of confusion is a metaphor for the separation between development and operations teams in sequential development models. Traditionally, QA and testing are on the development team's side of the wall.

While the development team aims to change the software continuously to release new software features, operations would prefer stability, thus minimizing the number of changes to production. This results in silos of conflicting goals, lack of communication, manual processes, and inconsistent environments. In short, there are different mindsets (i.e., motivation and goals), processes, and tools. The differences are stronger the more the teams are siloed. In DevOps, the wall of confusion is broken down by integrating the teams to work together, improving communication, and increasing collaboration. Leadership is needed to improve or remove inefficient processes, activities, and practices (e.g., prevent a blame culture by emphasizing shared responsibility) (DASA, 2024).

QA and test should be performed throughout the process, across both Dev and Ops, making testing a key factor in breaking down the wall.

Adopting DevOps can break down the walls between various siloed teams (e.g., security, business analysis, or usability).

1.1.3 DORA's Performance Metrics

The DevOps Research & Assessment (DORA) program publishes the annual State of DevOps Report. The research (DORA, 2024) shows four metrics on delivery performance that provide an effective way of measuring the software delivery process.

Two metrics relate to throughput (i.e., the velocity of making changes):

- Change lead time: The duration from code commit to successful production deployment. It reflects the efficiency of the delivery process, and tends to be short for high performing teams.
- Deployment frequency: The frequency of software changes deployed to production. It reflects how agile and responsive the delivery process is and tends to be high for high performing teams.

Two metrics relate to stability (i.e., the quality of changes delivered and the DevOps team's ability to fix failures):

- Change fail percentage: The percentage of deployments that cause failures in production, requiring hotfixes or rollbacks. It shows how reliable the delivery process is and tends to be small for high performing teams.
- Failed deployment recovery time: The time it takes to recover from a failed deployment. It shows how resilient and responsive the organization is and tends to be short for high performing teams.

High performing DevOps teams tend to score well on all four metrics. For that, they need good QE practices, including test automation, CI/CD automation, collaboration, test-first approaches, a whole-team approach (*ISTQB® CTFL*, v4.0), well-selected deployment strategies, and continuous monitoring. The faster the value stream, the better.

DORA has added reliability as an indicator for operational performance. According to DORA, reliability consists of metrics or quality characteristics such as availability, latency, performance efficiency, and scalability. The DevOps teams achieve better delivery performance outcomes (e.g., less personnel burnout) by focusing on reliability (DORA, 2022). The four performance delivery metrics relate to process, while reliability relates to software in use.

QA and testing contribute directly to better reliability, especially through test-first approaches, non-functional testing, chaos engineering (see *Section 4.2.6*), and quality hunting (see *Section 3.3.4*).

1.1.4 Elements of CALMS in the Context of QA

CALMS stands for culture, automation, lean, measurement, and sharing (Kim; Humble, et al., 2016). It is a framework that assesses the organization's ability to adopt DevOps processes and measure success during a DevOps transformation.

Culture reflects that merely changing processes and technology is not enough to make a lasting change. A collaborative, problem-solving culture is essential. The DevOps culture is an extension of Agile software development, where Agile teams have the autonomy to deliver value to customers while striving for necessary collaboration with other teams. This autonomy includes testing and a whole-team approach to ownership of quality, so testing is not outside of the DevOps team (Crispin et al., 2008).

Automation refers to automating as much as possible in the CI/CD pipeline. This includes build, deployment, provisioning, process automation, static analysis, and test automation, so "everything as code". Automation in the CI/CD pipelines enables the DevOps team to push small changes through continuously while maintaining good quality and speeding up the feedback loop.

Lean (Humble; Molesky, et al., 2020) means streamlining processes and removing waste (e.g., unnecessary steps in the process) so that the process is as efficient as it can be. Continuous improvement is part of lean and applies also to test activities, which should not become bottlenecks for the flow of work. So, rethink when and how testing is done in the process.

Measurement provides data for decisions on how software is used and how the SDLC works. Measurement involves using actionable metrics on quality and efficiency to improve products and continuously improve the SDLC.

Sharing is about the open culture of discussing challenges and successes of software development in DevOps teams. It is also about sharing good practices and ideas for learning and continuous improvement. QA and testing play a vital role in creating information to share (e.g., feedback from automated testing related to functionality and testability).

Each element of CALMS enhances the role of QA and testing in ensuring quality is built into every SDLC phase, enabling faster delivery of high-quality software while reducing risks.

Other versions of CALMS exist (e.g., CALMR in SAFe (*Scaled Agile Framework*, 2023a), where R stands for recovery).

1.1.5 The Three Ways of DevOps

To see how QA fits into software development practices, it is essential to examine DevOps principles. DevOps represents a cultural and technical movement that aims to unify development and operations, but its core principles extend far beyond just these two domains. These principles, known as the three ways of DevOps, provide a framework that helps organizations transform their software delivery process while

maintaining high-quality standards. Understanding these principles explains how QA practices evolve and integrate within a DevOps context.

In “The Phoenix Project” (Kim; Behr, et al., 2018) the three ways of DevOps are as follows:

1. The first way: flow
   This principle focuses on the smooth flow of work from development to operations, ensuring that work moves quickly and efficiently through the process. Flow emphasizes the importance of minimizing handover moments, reducing batch sizes, and removing bottlenecks to create a streamlined and predictable process.
2. The second way: feedback loop
   This emphasizes creating fast and continuous feedback loops. This enables fast detection and resolution of issues, preventing them from escalating. It also encourages a culture where everyone can learn from mistakes and continuously improve the product and delivery process.
3. The third way: continuous learning and experimentation
   This principle highlights the importance of fostering a culture of continuous improvement and experimentation. It encourages innovation, calculated risk-taking, and learning from both successes and failures. This principle supports the idea that continuous learning is essential for adapting to changes and improving the overall system.

QA is integral to the three ways of DevOps:

Flow:
QA enables smooth code progression through test-first approaches.

Feedback:
QA creates rapid feedback loops via continuous test automation, standardized pre-production environments, and monitoring to gather real-world usage data for improvements.

Continuous Learning:
QA helps innovation through targeted testing of new features, organized quality events such as quality hunting, and comprehensive regression testing.

1.1.6 Benefits, Risks, and Pitfalls of DevOps

DevOps brings opportunities, but organizations should pay attention to risks and pitfalls when implementing DevOps.

Benefits

DevOps can return significant benefits to organizations (DORA, 2024). These include the following:

- Empowering DevOps teams to assess quality at every phase of development, deployment, and production
- Capturing feedback early and often to improve both the software and the SDLC
- Delivering on-time with lower costs due to higher-quality software and more efficient processes
- Responding to changing needs faster
- Recovering from failures in production faster
- Improving time to market by employing efficient and effective processes and communication

- Making scaling of software faster with growing numbers of users due to infrastructure as code (IaC) (see *Section 4.2.3*)
- Improving collaboration in DevOps teams throughout the SDLC
- Reducing employee stress by enabling more frequent releases of small functionality increments
- Reducing vendor and third-party issues by increasing collaboration and transparency in all DevOps team's activities
- Improving test automation return on investment (Humble; Farley, 2010)

Risks and Pitfalls

While the benefits of DevOps are quite obvious, many things can go wrong. DevOps teams should anticipate these risks so that they do not become pitfalls that make the DevOps transformation fail. Typical risks and pitfalls include:

- Lacking management support
- Removing or replacing Ops instead of transforming it
- Replacing Agile software development instead of extending it
- Believing that there is just one way to do DevOps
- Using DevOps as a job title only
- Creating a DevOps team, but ignoring the need for change in other teams and in the organization
- Reducing the number of people by increasing automation instead of making people more efficient
- Introducing CI/CD pipeline tools only and forgetting DevOps cultural change
- Implementing a CI/CD pipeline with little or no test automation
- Using only automated testing, where manual testing would be beneficial

1.2 Implementing DevOps in an Organization

DevOps can be implemented in various ways in an organization. Regardless of how DevOps is implemented, there is always demand for QE including testing. Team members need to understand those concepts to work effectively with them.

This section discusses the typical roles in DevOps teams, including variations of a tester role, Site Reliability Engineering (SRE) in DevOps, and team topologies, patterns, and anti-patterns that affect DevOps' success. The team member experience related to the development environment is also considered. This is referred to as developer experience (DevEx).

Hands-On Objective QDO-1.2 (H0) Summarize How DevOps Principles are Visible in a Case Study

For this hands-on objective, the H0 level exercise supports the students in preparing them for further hands-on objectives by giving a context of a case study. The imaginary organization described in the

case study should be considered as only a partial DevOps organization so that the exercises based on the further hands-on objectives can implement improvements to the ways of working within this organization.

1.2.1 Roles in a DevOps Team

A role is a set of responsibilities and tasks, and not necessarily a job title. The core roles in a cross-functional DevOps team are business analyst, developer, tester, and operations engineer, often supported by a product owner and a scrum master. All team members together are responsible for QE.

Members of a cross-functional DevOps team can select any task that aligns with their knowledge and skills. This means that a person can have various roles over time and may even perform multiple roles in parallel. The team members may collaborate as a pair on a task. This approach optimizes workflow in the DevOps team.

A cross-functional DevOps team has all the knowledge and skills needed to perform any task that is common for the DevOps team. No part of the knowledge and skills should be with just one team member. In this way, a cross-functional DevOps team can still function when one of the team members is temporarily unavailable.

However, some specialized tasks may be beyond the skills available in the DevOps teams, (e.g., the setup and maintenance of the platform). For this reason, organizations may choose to have specific platform teams that provide self-service tools and reliable infrastructure that abstract complex operational tasks, enabling faster and more consistent software delivery by the DevOps teams. This way, the platform team reduces the operational burden on DevOps teams and enables quicker, safer, and more reliable deployments.

The quality of the work products of the DevOps team is maintained through collaboration, continuous learning, and improvement. Any work product is handled by at least two team members. The review process guarantees this (e.g., with pull requests (PRs)) (see *Section 2.1.4*). If the work product is not accepted, the reviewer gives feedback so the author can improve it, on their own or in collaboration with other team members. Any team member may be an author, a contributor or a reviewer of work products.

A cross-functional DevOps team consisting of a diverse group of skilled and motivated people that collaborate and take responsibility together is an effective way to create and maintain software. This is the foundation of the DevOps culture.

1.2.2 Site Reliability Engineering (SRE)

SRE is a set of practices that apply software engineering aspects to IT operations and infrastructure. SRE originates from the Google organization (Beyer et al., 2016). SRE focuses on ensuring a system continuously complies with the functional and non-functional requirements and needs. Its primary goal is maintaining service reliability by balancing release velocity with stability, typically through automation, monitoring, and proactive capacity planning. By creating automated feedback loops and using the feedback to make improvements, SRE supports the team to ensure systems stay within defined service levels, minimizing outages and optimizing efficiency, bridging the gap between development and operations in production environments.

It is common for a customer and a supplier to define the required service levels to ensure a software system's operational performance. A service level describes a specific aspect of a software system that is

relevant to stakeholders. The parties involved formalize their agreements in service level agreements (SLAs). SLAs are binding commitments that help prioritize development and operations efforts, and support in designing relevant test cases, ensuring the service consistently aligns with business goals and user needs.

The service level objectives (SLOs) and the service level indicators (SLIs) are important concepts in SRE. They are necessary to meet those SLAs. The SLOs describe what a software system must comply with, and the SLIs are used to measure whether the SLOs are met. An SLO includes the target value or range of values for a service level.

An SLI is a well-defined quantitative measure of the service level aspects that are provided. One or more SLIs are used to determine whether the SLOs are met, ensuring that the service is provided at the agreed upon level. SLIs are measured during development phases through testing. During the operations process, SLIs are measured by monitoring (e.g., with observability and telemetry). The automated execution of regression tests is important for measuring SLIs at various CI/CD pipeline stages.

The creation and implementation of systems and services in SRE follow three distinct phases:

- Planning and designing the required infrastructure: This covers quality characteristics such as reliability, scalability, and security.
- Implementing and deploying of the infrastructure: This includes setting up the infrastructure, deploying the system or service, and ensuring the system works according to the agreed service level.
- Ongoing operation, maintenance, and optimization of a system or service to keep it reliable and efficient over time.

1.2.3 DevOps Team Patterns and Anti-patterns

To meet the primary goal of delivering value for the customer, different organizations may adopt different DevOps team structures.

The organization's team structure depends on:

- The complexity of the software because higher complexity encourages silos
- Technical leadership and their ability to create a shared goal for Dev and Ops teams
- The readiness to change to a DevOps model in the IT operations department
- The necessary skills to lead the transformation to the DevOps model

A suitable integration of DevOps into the team topologies (Matthew Skelton, 2023) will facilitate collaboration and value delivery, while anti-patterns can hinder these goals by highlighting organizational issues.

DevOps team topologies

The following discusses three models. They are independent and not listed by preference.

- Dev and Ops collaboration:
  There is smooth cooperation between the Dev and Ops teams. Both teams learn the basics of how the other team works. This requires a clearly defined shared goal and significant organizational cultural change.

- Fully shared ops responsibilities: Operations people are fully integrated into the Dev teams. Organizations like Facebook and Netflix made it popular in the early 2010s. In this model, everybody is focused on the same purpose. The teams that build a feature are also responsible for maintaining its operations.
- SRE team:
  Organizations such as Google, define the collaboration between Dev and Ops teams as the SRE team that collaborates with Dev teams. For a detailed explanation of SRE teams, refer to *Section 1.2.2*. Dev teams need to provide test evidence that their work products meet the standards set by the SRE team without being directly involved in the operational concerns. This pattern can easily slip into anti-patterns and silos.

DevOps team anti-patterns

- Dev and Ops silos:
  This team structure has an "over the wall" mentality. "Done" is typically defined as feature complete and not released into production.
- Dev does not need Ops:
  New Dev teams in environments that use cloud services might think they do not need Ops.
- DevOps as a tools team:
  A separate DevOps team implements the tooling needed for DevOps. This may benefit the Dev department's tooling, but it falls short of the impact that true DevOps teams could have through the lack of integration into the everyday working practices of the development teams, where many benefits of the DevOps ways of working are found.

2 Quality Assurance (QA) and Testing in DevOps - 360 minutes

Keywords

A/B testing, acceptance test-driven development, Agile software development, automated testing, behavior-driven development, canary release, continuous integration, continuous testing, holistic testing, quality engineering, quality policy, sequential development model, static analysis, static testing, test policy

Quality in DevOps Specific Keywords

blue-green deployment, CI/CD pipeline, continuous delivery, continuous deployment, continuous discovery, continuous learning and experimentation, continuous monitoring, dark launch, pull request, release on demand, value stream

Learning Objectives in Chapter 2:

2.1 Contribute to All Value Stream Stages to Assist in Quality Engineering

QDO-2.1.1 (K3) Implement quality assurance activities in a DevOps context

QDO-2.1.2 (K2) Compare test objectives that support DevOps with sequential development models and Agile software development

QDO-2.1.3 (K2) Explain continuous testing in a DevOps context

QDO-2.1.4 (K2) Explain how pull requests support quality

2.2 Contribute to the DevOps Loop Implementation

QDO-2.2.1 (K2) Explain quality assurance and testing in continuous discovery and its practices

QDO-2.2.2 (K2) Explain quality assurance and testing in continuous integration and its practices

QDO-2.2.3 (K2) Explain quality assurance and testing in continuous delivery and its practices

QDO-2.2.4 (K2) Explain quality assurance and testing in continuous deployment and its practices

QDO-2.2.5 (K2) Explain quality assurance and testing in release on demand and its practices

QDO-2.2.6 (K3) Apply quality assurance and testing knowledge to implement a CI/CD pipeline

Hands-On Objectives:

2.1 Contribution to the Value Stream to Assist in Quality Engineering (QE)

QDO-2.1 (H2) Contribute to the Value Stream to Assist in Quality Engineering

2.2 DevOps Loop Implementation

QDO-2.2 (H2) Contribute to the DevOps Loop Implementation

2.1 Contribution to the Value Stream to Assist in Quality Engineering (QE)

QA and testing are key in the DevOps context, throughout the value stream. A value stream is a series of interconnected processes, activities, and workflows that deliver value to customers through a product, service, or experience. Although QA and testing are performed differently in Agile software development and sequential development models, they both have a large impact on software quality. From an overall perspective, there is more to achieving the right quality level than just QA and testing. QE is overarching QA and testing; it is about all activities that contribute to creating the software that the customer needs.

This section elaborates on the activities and practices from the DevOps context, specifically, with a hands-on exercise. The QE objectives in DevOps are compared to Agile software development and sequential development models.
The final section describes the importance of a pull request (PR) process.

Hands-On Objective QDO-2.1 (H2) Contribute to the Value Stream to Assist in Quality Engineering

For this hands-on objective, the H2 level exercise supports the students in learning how to create an overview of the QE activities specific to DevOps for a specific case. This includes what QA and testing deliverables should be defined, aimed at achieving the right quality for the three aspects of quality: products, processes and people.

2.1.1 QA and Testing in a DevOps Context

DevOps practitioners strive to build in quality from the start and to implement a "right first time" approach. This will promote an efficient way of working throughout the SDLC and effectively implement stakeholders' needs. As such, DevOps is a further elaboration of the Agile mindset. To achieve this elaboration, any DevOps team or group of collaborating DevOps teams needs to implement activities related to achieving built-in quality and supplying information about the achieved quality level. A documented quality policy and test policy will support this. The following lists show a grouping of these principles, activities, and practices for the three aspects of quality: products, processes, and people (Marselis et al., 2020):

Quality of Products

The product quality created by the DevOps team(s) depends e.g., on the following principles:

- Achieving business value is the main goal of development
- Implement architecture that enables the productivity of the DevOps team
- Implementing functional and non-functional quality characteristics of the software correctly (Kim; Humble, et al., 2016; *Systems and software engineering (25010)*, 2023)
- Paying attention to the quality engineering activities that need to be applied during all DevOps lifecycle activities, this includes preventive quality activities, such as collaborative user story writing and pair programming, and detective quality activities such as static testing and dynamic testing.

Quality of Processes

The processes that aim to build the right quality level consist e.g., of the following activities:

- Providing fully integrated monitoring, controlling, reporting, and alerting related to the measured indicators for business value and quality level
- Applying a risk-based approach to estimating the efforts and scheduling the tasks and later activities
- Establishing the infrastructure and tooling needed for the QE activities integrated into a CI/CD pipeline
- Implementing continuous improvement practices

Quality of People

People involved need to apply, as part of the Agile mindset, e.g., the following practices to properly deliver the products:

- Hypothesis-driven development: (We believe <a certain action> will result in <an intended goal>. We will have confidence to proceed when <a measurement exceeds a certain target>) (Kim; Humble, et al., 2016)
- A collaborative working environment
- A focus on continuous improvement

To implement these principles, activities, and practices, an organization needs to arrange overarching guidelines for all DevOps teams and stakeholders involved. Major work products are the quality policy and the test policy which may be combined into one document, with a test-first approach for built-in quality and an "automate everything you can" mindset. The quality policy and test policy describe the roles and responsibilities for cross-functional teams, describe the organization of multiple teams, and outline their implementation. Implementing the policies can be based on a product risk analysis, the testing pyramid and/or the Agile testing quadrants (*ISTQB® CTFL*, v4.0) and result in a QE strategy (Marselis et al., 2020). Implementation requires collaboration of the DevOps team members, and between multiple DevOps teams.

2.1.2 Compare Test Objectives in Different SDLC Models

The three ways of DevOps (i.e., flow, feedback, and continuous learning) (Kim; Humble, et al., 2016) require a fully integrated approach to QA and testing. Quality must be built in from the start and testing supplies information whether the right quality level is delivered at the right moment.

In sequential development models, testing is performed at separate test levels and done by specialized test teams.

In Agile software development, the Agile team tests to confirm that a user story is done and acceptance criteria are met. Deferring test tasks to the end of an iteration can lead to inefficiencies and quality risks.

DevOps extends the Agile mindset; the principle of flow states that every work product should be independently finished and delivered, and testing is an integral part of development.

The test objectives are similar in all three of the above-mentioned approaches, in that they supply information about the quality to build confidence, while test objectives in Agile software development and DevOps also include the need to supply early and continuous feedback.

In DevOps, automated test execution is essential in a CI/CD pipeline. Test automation is essential for achieving flow. In contrast to sequential development models, where teams can be successful without any automated test execution, and Agile software development where automated testing is often focused on regression testing, in DevOps, test automation is applied to all test levels and test types. However, even in DevOps, using automated CI/CD pipelines, there will still be some need for manual testing, especially exploratory testing. Manual testing is required because some aspects of quality are difficult to assess in an automated way (e.g., usability). And lastly, manual testing suits stakeholders who want to view test results for themselves before they are convinced that the software quality level is right.

2.1.3 Continuous Testing

Continuous testing integrates test activities throughout the software development lifecycle (SDLC) so that testing occurs from the initial concept through to production deployment and beyond rather than a separate phase.

Core principles of continuous testing

- Test early and often
  Continuous testing emphasizes initiating test activities as early as possible and maintaining them consistently throughout the SDLC. By verifying and validating the software at every phase, DevOps teams assess and report on the quality level early, significantly reducing the cost and impact of defects.
- Cross-functional collaboration
  In continuous testing, quality is everyone's responsibility. The approach requires active participation from all DevOps team members, not just dedicated testers. This collaborative approach ensures diverse perspectives and comprehensive quality assessments. This is similar to the "whole team approach" (*ISTQB® CT-TAS*, v1.0).
- Automation and tooling integration
  To enable efficient continuous testing, DevOps teams rely heavily on automation. Automated tests in the CI/CD pipeline provide quick feedback and enable consistent quality. Tools for various test types are used to achieve comprehensive coverage.
- Production monitoring and observability
  Continuous testing extends beyond pre-production environments into production. Continuous testing emphasizes the importance of monitoring and observability in production environments. DevOps teams use specialized tools to track the software performance in production, detect anomalies, and gather data on user behavior in real time to extend the test phase into production. See section (see *Section 4.2.7*) for more information on monitoring and telemetry.
- Adaptive improvement
  Continuous testing promotes ongoing refinement of test strategies. DevOps teams continuously gather data and feedback from various test activities, using these insights to adapt their test strategies and enhance their test approaches over time.

Framework example

One framework that exemplifies continuous testing is holistic testing developed by Janet Gregory and Lisa Crispin (Janet Gregory, 2023).

This framework helps DevOps teams to structure and apply continuous testing in real development scenarios, simplifying adoption and adaptation for their needs.

2.1.4 Pull Requests

A PR, sometimes referred to as a merge request, is an agreement that requires the author of code to request the review of code by one or more team members before the code can be merged into the main branch. In a PR, a branch is a separate line of development in a version control system (VCS) where changes occur before integrating them into the main codebase.

The concept of PRs evolved from practices in VCSs. Before cloud storage became the de facto standard for VCSs, developers would write new code on their local machine, and other developers would "pull" it to help determine its quality.

Their feedback is incorporated into the work product, and supports higher quality in multiple ways.

- PRs support quality in DevOps by encouraging collaborative development and knowledge sharing between team members, leading to better coding practices and higher awareness of potential issues.
- PRs can serve as one of the quality gates to start an automated test run in the CI/CD pipelines, as discussed in section 2.2.
- PRs help enforce coding standards, adherence to style guides, and best practices through manual code reviews supported by static analysis performed in the CI/CD pipeline.
- PRs improve traceability by providing a clear history of changes, discussions, and decisions made about specific features or fixes, fostering accountability.
- By encouraging small, manageable code changes rather than large, complex updates, PRs limit the amount of change and associated risks.
- Acting as a quality gate, PRs ensure only approved changes make it into the main branch.
- PRs promote accountability by having specific DevOps team members review and approve changes, supporting intentional knowledge sharing, including reviews by all teams affected by the code changes.

2.2 DevOps Loop Implementation

A value stream is a series of activities from idea capture to releasing value to customers. Software development, delivery, and operations are not discrete sets of activities that happen in isolation; instead, they occur in close proximity, especially with Agile software development and DevOps ways of working (DASA, 2019).

A value stream is illustrated as an infinity loop, the so-called DevOps loop. While there are different DevOps loop illustrations with different sequences of activities, the main group of activities is the same. This section discusses how QE, QA, and testing are considered in the groups of activities of the value stream, specifically in continuous discovery, continuous integration (CI), continuous delivery (CD) and continuous deployment, release on demand, and continuous monitoring.

QE, including QA and testing, happens in all of the DevOps loop activities, all contributing to software quality. These activities can go quickly or slowly, depending on the number of knowns or unknowns, and

sometimes a DevOps team will need to pause and go back to a previous activity or two to reiterate it before going forward again (e.g., to understand if a feature makes sense) (Janet Gregory, 2023).

While there can be manual process steps in the value stream, striving for automation is common in DevOps.

The CI/CD pipeline refers to the automation related to the code, including reporting on the outcome of the activities. However, it does not include all activities of the value stream. Other tools can support these activities, like task management tools, where task automation is possible. Task management tools and CI/CD tools are often tightly connected.

Section 2.2.1-2.2.5 describes the DevOps loop activities. Section 2.2.6 describes the steps to incrementally implement CI/CD pipelines in the organization covering these activities.

Hands-On Objective QDO-2.2 (H2) Contribute to the DevOps Loop Implementation

For this hands-on objective, the H2 level exercise supports the students in learning how to design and implement a CI/CD pipeline, what stages, and what capabilities for tooling to be included to represent a DevOps loop.

2.2.1 QA and Testing in Continuous Discovery

Continuous discovery is a practice of frequently discovering and evaluating findings, customer and stakeholder needs, innovative ideas, modern technologies, and other opportunities to decide what value to develop next. DevOps teams focus on building features that drive value by continuously engaging with stakeholders, applying customer-centricity and design thinking, and discovering their needs (Torres, 2024), (*Scaled Agile Framework*, 2023b), (*Scaled Agile Framework*, 2023c).

Usually, three activities take place in continuous discovery: discover, plan, and understand (Janet Gregory, 2023).

- Discover is about adding new features and changes to the software so it will help stakeholders solve their problems and meet their needs. Stakeholders validate these needs and ideas before defining features. This can be achieved, for example, through stakeholder interviews or by developing quick user experience (UX) prototypes. Product owners, supported by the DevOps team, identify the features that will enable the team to create, improve, and deliver the solutions for stakeholders.
- Plan determines quality characteristics and identifies risks that are important for the feature while detailing features and slicing these features into smaller user stories. Since DevOps uses an iterative/incremental approach, the backlog items are identified and detailed incrementally, based on short learning cycles, instead of defining everything upfront.
- Understand is about getting a shared view of the details of user stories within the DevOps team or across multiple teams involved.
  DevOps team members apply a test-first approach, using acceptance test-driven development (ATDD) or behavior-driven development (BDD), which includes describing the specification by example (Adzic, 2011). These example scenarios help developers understand and test the solution. This test-first approach is a collaborative test activity supporting built-in quality. Understanding the backlog item by the DevOps team members is addressed by fulfilling the definition of ready (DoR) (e.g., define and understand acceptance criteria, and estimate backlog items) (Alliance, 2026). A

common definition of done (DoD) drives the identification of all work products to deliver for any backlog item (Alliance, 2026).

A QA best practice is having DoR and DoD checklists in place, defined, agreed upon, and followed by the DevOps team members.

2.2.2 QA and Testing in CI

For each merged change, CI requires building the entire software and running a comprehensive set of automated tests, both static and dynamic against it (Beck, 2000). Coding style breaches, not achieving coverage targets, or slow test cycles could result in a broken build. If the build or the tests failed, the team members stop everything they are doing and fix the build immediately. This "stop-and-fix" approach is a quality management aspect in lean, as discussed in (Humble; Molesky, et al., 2020). This fix could be to roll back to the previous, working version of the software.

The goal of CI is to have software in a working state all the time, so it is potentially shippable. Potentially shippable is a statement about software quality and not about the value or the marketability of the software (Larman et al., 2016).

DevOps teams need to have the following practices in place to implement CI, as described in (Humble; Farley, 2010):

- Include everything in configuration management (e.g., automated test scripts, automated deployment scripts, test data, and environment configurations).
- The automation scripts for the build steps or test steps in the CI/CD pipeline should facilitate investigation in case of build failures.
- DevOps teams check in small, incremental changes and wait for the tests to pass or follow a stop-and-fix approach if any test failed.

As described in (Humble; Farley, 2010), the CI pipeline usually consists of two main parts:

- Actions in the commit stage test that the software works at the technical level. This triggers compilations, automated low-level test execution, and static analysis of the code. Code review by other DevOps team members usually happens only after the actions in the commit stage pass.
- Actions in the automated acceptance stage test that the software works, conforms to the specification, and meets the user's needs. The acceptance stage can involve any test types or test levels required for accepting the software. Test activities can run in parallel in this stage.

When multiple DevOps teams contribute to the software, usually each team has its own team-specific CI part of the CI/CD pipeline, which is connected to a common, organizational level CD part of the CI/CD pipeline (see *Section 2.2.3* and *Section 2.2.4*).

2.2.3 QA and Testing in CD

Continuous delivery (CD) ensures that the software is always in a deployable state, including for production environments. Whether the software is actually deployed to production, is up to a person to decide based on test results and readiness assessments (compared to continuous deployment, where deployment happens automatically if all tests passed, without any human intervention, as described in *Section 2.2.4*).

After CI stages (see *Section 2.2.2*), the software is deployed to production-like test environment(s) for further testing. These tests can utilize test automation, use experience-based test techniques, or both. It is common to run smoke tests after deploying to an environment and before any planned tests are executed to test that the deployment was successful. Tests can run on multiple environments in parallel to provide fast feedback, if the needed resources are available.

Quality is achieved through testing and by the following QE practices:

- Deployment automation using automation tools for consistency, compliance, and to provide an audit trail (see *Section 4.1*)
- Standardized test environment and test data management (see *Section 3.1.6*)
- IaC to manage infrastructure configuration using code for consistency and repeatability (see *Section 4.2.3*)

The release strategy that the DevOps team has adopted (see *Section 4.2.2*) gives guidelines about how often and how to deploy to production (see *Section 2.2.4*), and how to release to users (see *Section 2.2.5*).

2.2.4 QA and Testing in Continuous Deployment

Continuous deployment builds on the CI/CD pipeline and the practices of CI and CD. Continuous deployment is different from continuous delivery (CD) because it automatically deploys every change that passed its automated tests to production. There is no human involvement in deployment decisions. This approach ensures that the software is always in a deployable state and the latest version is deployed to production.

The deployment to production does not necessarily mean that the software is released to users. See *Section 2.2.5* for details. However, the frequently repeated and reliable deployment procedure reduces the risk of failed deployment during software release.

The confidence to always deploy to production is based on the following factors:

- Having QE practices in place to build in quality throughout continuous discovery, CI, and CD
- Having an automated, controlled deployment process without human involvement
- Trusting in the automated test suite in the CI/CD pipeline to achieve agreed, acceptable levels of quality
- Applying deployment and release strategies to control how to deploy the application and provide user access (e.g., dark launch) (see *Section 4.2.2*)
- Having the ability to rapidly respond to the undesired consequences of the deployment (e.g., roll back to a previously defined state of a system or roll forward deploying a newer version of a system to fix defects)

Testing occurs in production before releasing software to the users to further reduce risk and generate feedback on the new functionality or other quality characteristics (see *Section 3.3.3*).

- With a dark launch, the new feature runs in the production environment alongside existing functionalities without exposing it to users, allowing DevOps teams to monitor performance and behavior, test, and observe the system under real-world conditions.

- In blue-green deployment, two identical environments, "blue" (current live) and "green" (new version), are used to minimize downtime and risk during deployment. Testing occurs on the new version (green) while the old version (blue) stays live. After all tests are passed, the green environment becomes live, and the blue environment is ready for the next change and its tests, so they swap roles. Two environments enable fast rollback if issues arise after a new version releases.
- A canary release first exposes a small subset of users to a software change to validate quality, and then incrementally rolls it out to the whole user base if no critical defects exist. The first subset of users can be a random percentage of the total user base, a pre-selected group of known users (i.e., beta testers), a geographic area, or any other grouping.
- A/B testing is a test approach for comparing two or more variations of a feature, interface, or functionality by presenting each variant to different segments of users and analyzing performance metrics to determine which version achieves the desired outcome more effectively, and keep that variant in production, while removing the other(s).

2.2.5 QA and Testing in Release on Demand

Release on demand is a DevOps practice to release new features immediately or incrementally based on business, customers and any other stakeholder needs, decoupling deployment and release. When a new feature is deployed to production, it remains hidden from users until access is provided in a controlled way using feature management practices and tools (see *Section 4.2.2* and *Section 4.2.4*). This approach allows the business to release features to users when market timing is best (e.g., at a given date before the holiday season). It supports business agility, delivering and maintaining the highest value to users when they need it, and for as long as they need it.

Decoupling architecture elements to be releasable separately reduces the risk of service interruptions during deployment and release. This technique identifies specific architecture building blocks, each of which can be released independently (e.g., open-source backend databases follow major release cycles, while front-end microservices are updated multiple times per iteration). The different building blocks can have different release strategies and frequencies.

The release can involve non-software-related activities (e.g., updating user manuals, training materials, release notes, operational procedures, legal contracts, or marketing and sales activities).

After the software is released, operations focus on critical activities that preserve the stability and security of the software (see *Section 1.2.2*). Continuous monitoring practices are used to observe the software in its environment, to provide information if software quality meets agreed upon objectives, and to be able to take corrective action after being alerted to quality degradation (see *Section 4.2.7*). IT service management (ITSM) activities also support quality (e.g., establishing a service desk to provide support to customers and users).

Observability measures system behavior against its functional and non-functional requirements (see *Section 4.2.7*). Additionally, information is gathered from other channels (e.g., customer feedback from the service desk or social media). Collected data helps to understand how the released change provides business benefits (e.g., increased number of subscriptions or improved sales).

The organization can use metrics to apply learning and identify the next potential value to deliver. This can be an enhancement to the latest feature, a non-functional improvement, or the start of a new feature. Additionally, the organization can apply the learning to improve the processes, practices, people, and product architecture.

2.2.6 CI/CD Pipeline Implementation

Implementing a CI/CD pipeline should follow an incremental approach with the following steps as defined by (Humble; Farley, 2010).

1. Model the value stream from commit to release and create a walking skeleton

The value stream mapping (VSM) technique maps the part of the value stream from code check-in to release (see (*ISTQB® CT-ATLaS*, v2.0) for details on VSM). Then the mapped steps are implemented without detailed content in the CI/CD pipeline. When the predefined quality gates are met, this triggers the next step of the mapped process, and the output of a step becomes an input for others (e.g., build artifacts and test databases).

The CI/CD pipeline code is usually stored in the same repository as the application's source code. However, with more complex projects, workflows can be reused throughout repositories.

A walking skeleton shows the flow and connects the steps before adding details.

2. Automate build and deployment process

The CI toolset should trigger a build on every check-in. The build process takes source code as its input and produces programs as outputs. After the build the output is automatically deployed to a test environment and tests are executed.

Provisioning of an environment and test data should be fully automated and reproducible. IaC, virtualization and containerization technology support this step (see *Section 4.2.3* and *Section 4.2.9* respectively). Operations roles are usually involved in developing this automation (see *Section 1.2.1* and *Section 1.2.2*). Test environments are deployed the same way as production environments.

3. Automate commit stage

The next step is implementing a full commit stage (see *Section 2.2.2*), running static analysis and unit tests, and including a selection of component tests, integration tests and system tests on every commit. In this stage, vulnerability scanning and dependency scanning implement security practices. It is common to implement a quality gate to check if the build needs to be stopped if agreed upon thresholds are not met (e.g., not enough code coverage or test execution time is too long). Once the commit stage gets too long (e.g., over five minutes), it makes sense to split it into parallel jobs.

4. Automate acceptance stage

Implementing the acceptance stage (see *Section 2.2.2*) should start with automating a few test cases from all different test types in the CI/CD pipeline and growing the test suites incrementally, instead of trying to automate the majority of the test cases of one test type before moving on to the next one.

Tests run sequentially or in parallel, each in their own environment to optimize feedback. However, consider resource availability, cost, and sustainability goals.

5. Automate the release

Automating this process requires understanding the way the release decision is made during the release process (see *Section 4.2.2*), and it requires collecting information to make this decision, whether it is automated or manual. Feature management tools can support decision makers to manage their decisions

(e.g., updating feature toggles in code) (see *Section 4.2.4*). Automating incrementally applies to these release activities, too.

The implemented process should follow the agreed deployment and release strategy, support fast rollback, and provide an audit trail on all activities for compliance.

6. Evolving the CI/CD pipeline

Implement the CI/CD pipeline incrementally. As the project gets more complex, the value stream will evolve. Measure the efficiency of the CI/CD pipeline using agreed upon metrics (e.g., lead time of any given activity). Continuously evolve and improve the CI/CD pipeline in line with lean principles and systems thinking (e.g., splitting long test executions into parallel jobs, or removing non-value added jobs). See (*ISTQB® CT-ATLaS*, v2.0) and (*ISTQB® CTAL-TM*, v3.0) for continuous improvement practices.

3 Automated and Manual Tasks in DevOps - 510 minutes

Keywords

API testing, automated testing, contract testing, crowd testing, exploratory testing, quality hunting, quality report, quality reporting, regression testing, test automation, test data, test data management, test result, traceability

Quality in DevOps Specific Keywords

build artifact, CI/CD pipeline, failed deployment recovery time, personally identifiable information, quality report, single source of truth, software binary, statistical analysis

Learning Objectives in Chapter 3:

3.1 Describe How Automation Supports Quality Assurance in DevOps

QDO-3.1.1 (K2) Explain how a single source of truth for testware supports quality assurance

QDO-3.1.2 (K2) Explain how automation supports traceability between the test basis and testware

QDO-3.1.3 (K3) Implement uniform quality reporting practices in the CI/CD pipeline with information from both automated testing and manual testing

QDO-3.1.4 (K2) Explain the benefits of test data management automation

QDO-3.1.5 (K2) Explain the benefits of statistical analysis of test results over long periods of time

QDO-3.1.6 (K2) Explain the benefits of standardized, controlled, and automated test environment management

3.2 Implement Automated Testing in DevOps Teams

QDO-3.2.1 (K2) Explain how regression testing integrates into various CI/CD pipeline activities

QDO-3.2.2 (K3) Prepare API testing

QDO-3.2.3 (K2) Compare the extent of test automation in DevOps to Agile software development and sequential development models

3.3 Implement Manual Testing in DevOps Teams

QDO-3.3.1 (K2) Give examples of manual testing for a DevOps organization

QDO-3.3.2 (K2) Explain how exploratory testing can work with a CI/CD pipeline

QDO-3.3.3 (K2) Explain how crowd testing can support the culture of feedback

QDO-3.3.4 (K3) Apply quality hunting events to support the culture of learning

Hands-On Objectives:

3.1 Automation's Support for Quality Assurance (QA) in DevOps

QDO-3.1 (H2) Describe How Automation Supports Quality Assurance in DevOps

3.2 Automated Testing in DevOps Teams

QDO-3.2 (H2) Test Automation in DevOps Teams

3.3 Manual Testing in DevOps Teams

QDO-3.3 (H2) Implement Manual Testing in DevOps Teams

3.1 Automation's Support for Quality Assurance (QA) in DevOps

Automation plays a crucial role in QA in DevOps by improving consistency, traceability, and efficiency across the SDLC. Implementing a single source of truth (SSOT) helps centralize and standardize information, ensuring teams work with accurate and up-to-date information, which enhances collaboration and reduces errors. Automated traceability between requirements, testware, and deployments ensures accountability in every change, improving defect management and compliance. Quality reporting integrated with the CI/CD pipeline provides real-time insights through dashboards, combining automated and manual test results for comprehensive analysis and alerting. This provides fast and reliable feedback on quality for teams and stakeholders. Automation in test data management, statistical analysis, and test environment provisioning accelerates test cycles, helps to provide data security, optimizes resource usage, and enhances decision-making, driving continuous quality improvements in DevOps.

Hands-On Objective QDO-3.1 (H2) Describe How Automation Supports Quality Assurance in DevOps

For this hands-on objective, the H2 level exercise supports the students in learning how to implement automation to support QA in DevOps. This exercise enables students to apply automation concepts to a known organizational context and to reason about how automation supports QA across the DevOps value stream.

3.1.1 Single Source of Truth (SSOT) for Testware

As DevOps organizations grow, managing multiple sources of information becomes increasingly challenging. Fragmented, inconsistent, and duplicated information across DevOps teams and tools leads to errors, poor collaboration, and difficulty maintaining transparency and traceability between configuration items. Applying a single source of truth (SSOT) architecture and practices on an organizational level, DevOps teams can centralize and standardize information, making sure one information type is stored in only one storage location. This ensures that all teams access accurate, consistent, and up-to-date data, which helps to prevent errors caused by relying on incorrect information.

Store development and testing work products in SSOT systems (see *Section 4.1.1*) as follows:

- Information management systems provide information lifecycle management by storing backlog items (e.g., requirements, defect reports, and tasks), and project and product documentation
- Version control systems (VCSs) support controlled change management by storing software source code, build and deployment scripts, infrastructure as code (IaC) configurations, CI/CD pipeline definitions, test cases, test automation code, and test data (e.g., text format data)
- Artifact management systems support the reusability of build artifacts by storing dependencies, software binaries, and test data (e.g., binary data)
- Observability systems provide traceability and access management by storing traces, logs, and metrics

In general, SSOT systems provide DevOps teams with the following benefits:

- Centralization: Acts as the sole location where critical data is stored, updated, and retrieved

- Consistency: Ensures that all teams work with the same up-to-date information, avoiding discrepancies
- Automation: Facilitates automated workflows by integrating with tools and systems across the DevOps toolchain
- Collaboration: Promotes cross-team alignment by providing a shared understanding of information
- Auditability: Provides a clear history of changes, which helps track and ensure compliance

3.1.2 Traceability between the Test Basis and Testware

It is important for DevOps teams to register the relationship between related work products, such as requirements, user stories, code, and testware. Traceability ensures that all SDLC work products are connected, enabling teams to verify that every requirement has been implemented, tested, delivered, deployed, and released. Bi-directional traceability also supports maintenance (*ISTQB® CTFL*, v4.0). For example, when a change is made to a requirement, it is easy to find where to implement such a change. Also, the root cause of a code defect can quickly be found in a user story or requirement. The organization's test policy describes the need for traceability (Marselis et al., 2020).

Tools can efficiently register and monitor the traceability of all related work products. Automation tools are fundamental in providing dynamic updates to traceability matrices and reducing the risks associated with manual tracking. Configuration management tools and/or task management tools, which are capable of registering various relationships between work products provide the automation. These tools provide reports and overviews of the work products available within the organization and detail their usage.

Traceability fosters collaboration between development, testing, and operations roles within the DevOps team by making relationships between work products transparent and auditable. This transparency improves defect tracking and resolution, and also supports compliance with standards and regulations. It is a key enabler of continuous discovery, continuous integration, continuous delivery, continuous deployment, continuous testing and continuous monitoring, and contributes to maintaining quality and agility in DevOps.

3.1.3 Quality Reporting for a CI/CD Pipeline

At any given time, it is important to understand the software quality level throughout the SDLC. This provides feedback to the DevOps team on code quality and to product management on release readiness. Integrating quality reporting into the CI/CD pipeline provides the latest development status. Each stage of the CI/CD pipeline creates information for quality reporting. Some quality data is generated automatically by the CI/CD pipeline (e.g., build success, test passed/failed status), while other data comes from manual testing (e.g., user acceptance testing, exploratory testing, or coverage in a test management tool). This information creates a comprehensive view of the software quality.

Quality reporting should occur consistently for each cycle of the CI/CD pipeline and thus be automated. Manual test results should be included in the quality reporting, typically via integration with a test management tool. The information is collected into metrics and combined into a quality report. The quality report can be delivered using various means (e.g., by a dashboard on a wiki page, in a shared tool, as an email or report based on a template, or as a combination of these) (*ISTQB® CTFL*, v4.0).

Automated quality reporting requires the integration of several tools. These include various CI/CD pipeline tools' logging, monitoring, and reporting functionalities (see *Section 4.1.2*). Task management and test management tools record the manual test results (e.g., the estimated quality level of software

features or passed/failed test results). Production monitoring tools provide telemetry (see *Section 4.2.7*). Report generators should integrate with all these tools to create easy to understand reports for various stakeholders. Create a single source of truth by combining the metrics information from various sources into a single, accessible location (see *Section 3.1.1*).

The quality reporting implementation process has several steps:

1. Determine the key performance indicators (KPIs), and related metrics to track, such as code quality, coverage, performance, build stability, build success rate, deployment frequency, failed deployment recovery time, and security vulnerabilities.
2. Collect data using chosen metrics.
3. Automate data gathering from various CI/CD pipeline stages.
4. Clean and normalize the data for better consistency with data preparation tools (see *Section 3.1.4*).
5. Store the collected data in a centralized database if the tools do not provide it.
6. Visualize metrics using dashboards, such as charts, graphs, and tables that display the key metrics, including different sections for each stage of the CI/CD pipeline.
7. Enable continuous monitoring in the dashboard for real-time updates.
8. Ensure the dashboard accessibility for relevant stakeholders, such as developers, testers, product owners, scrum masters, and managers.

A quality report dashboard can enhance a CI/CD pipeline. Benefits include:

- Tracking key metrics continuously for quick response and resolution
- Analyzing CI/CD pipeline performance trends to identify bottlenecks and optimize the CI/CD pipeline efficiency
- Reviewing security compliance status proactively to ensure the codebase meets security standards
- Fostering a culture of transparency and collaboration through shared dashboards with stakeholders, aligning everyone on the project's quality status and progress
- Supporting retrospectives with data-driven insights to identify areas for improvement

3.1.4 Test Data Management Automation

Managing test data manually can reduce flow and slow down feedback cycles (see *Section 1.1.5*). DevOps teams use test data management automation to create, update, and manage test data within the CI/CD pipeline. Using standardized test data helps to accelerate test cycles, improve coverage, reduce errors, reduce inconsistency of test results, improve data security, and follow privacy regulations.

Effective test data management automation requires collaboration between all roles within the DevOps team to identify what data to focus on and what practices to use.

Key aspects of test data management include:

- Data design: Analyze and define needed test data for a given test objective (e.g., data types, sizes, and complexity).
- Data generation: Create or update test data to enable comprehensive coverage and a variety of inputs (e.g., valid, and invalid data, edge cases, and boundary values).

- Data validation: Automate validation to determine data accuracy and compliance with standards and regulations.
- Data organization: Organize and store test data in a structured way in a VCS and make it available on demand within the CI/CD pipeline or for manual testing.
- Anonymization: Mask or anonymize production data before using it as test data to protect privacy and confidentiality.
- Data refresh and cleanup: Automatically refresh or reset test data as part of test automation.

DevOps teams apply the following practices in test data management automation:

- Synthetic test data creation: Data formatting rules, analyses of the structure of production data, or pre-trained GenAI models can generate artificial data to match test conditions, without using actual production data.
- Test data cloning: Create high-volume and valid test data for performance testing and load testing by cloning existing data.
- Test data subsetting: Select a smaller, representative subset of production data for testing, while keeping references between data items, to reduce storage needs and accelerate test data provisioning.
- Test data masking: Mask or modify personally identifiable information (PII) from production data to make the data anonymous and protect an individual's identity (e.g., name, address, email, phone number, IP address, and location data).

The DevOps team should understand the risks of test data management automation and manage them using relevant processes and well selected and configured tools. These risks include:

- Incomplete coverage: Generated data may not reflect real-world complexity.
- Resource utilization issues: Generating, storing, and distributing large amounts of test data can consume significant resources.
- Breach of security and compliance rules: Exposure of sensitive information by improper data masking may result in fines.

3.1.5 Statistical Analysis of Test Results

Test execution produces various kinds of test results. The short-term use of individual test results is to inform stakeholders about quality and related risks so they can establish their confidence in the business value of the software. The long term use of test results is to get a broader picture of the quality of the software, the processes applied and the people involved. In both cases, by applying statistical analysis, the DevOps team can identify trends and patterns, related to the quality of the software and the residual risks, and use this information to support decision-making. For stakeholders to establish confidence, they need metrics like requirements coverage, code coverage, defect density, and mean time between failures.

The DevOps team can use tools such as AI-based predictive analytics to predict future results that the software will produce. Predictive analytics support improving both the process and the people involved when the predicted future situation does not meet the desired situation. The predictive analysis may result in the need for process improvement based on the measured test effectiveness or may trigger the allocation of more or different resources.

A/B testing uses statistical analysis to compare multiple variants of software shown to separate user groups to determine which variant is the most valuable. Another use case for statistical analysis is in evaluating the test results of non-functional tests, such as in performance testing based on load profiles (Marselis et al., 2020).

Statistical analysis is valuable in analyzing the test results of one test run or multiple test runs (e.g., to see the trends in software quality and to determine if the quality increases after every iteration). Test process improvement can use statistical analysis. This helps with detecting inconsistent tests, and analyzing test automation keywords with long test execution times, which supports optimization of test execution time and related cost. A dashboard can display statistical analysis outcomes using tools.

The tools for statistical analysis should be integrated into the CI/CD pipeline.

3.1.6 Standardized, Controlled, and Automated Test Environment Management

Standardized, controlled, and automated test environment management is a cornerstone of software development. By standardizing test environments, organizations eliminate the variability caused by inconsistent configurations, ensuring that tests yield reliable and reproducible test results. This consistency reduces the time spent troubleshooting defects that arise from environmental differences, allowing DevOps teams to focus on actual defects in the code rather than setup discrepancies.

Automation streamlines the provisioning and management of test environments, replacing time-consuming manual processes with rapid, repeatable setups. This enables faster test cycles, where environments can be spun up and torn down as needed. This could include test data management (see *Section 3.1.4* and *Section 4.2.9*). DevOps teams can test and iterate on their code more frequently, speeding up software delivery through the CI/CD pipeline and increasing overall productivity.

Controlled environments enable resource efficiency by ensuring that environments are right-sized and tailored to the needs of specific tests. This reduces the risk of over-provisioning, where excessive resources are allocated unnecessarily, as well as under-provisioning, which can cause performance bottlenecks. On-demand environment provisioning lowers operational costs and uses resources only as needed.

Using IaC to create replicable environments (see *Section 4.2.3*) eliminates ambiguity about configurations, improving collaboration among testers, developers, and operations. The use of IaC supports the goal to remove the "wall of confusion". These environments then serve as blueprints that can be easily replicated in the cloud, so they can be scaled easily, accommodating increased testing demands or larger team sizes.

3.2 Automated Testing in DevOps Teams

Regression testing is crucial in DevOps for maintaining quality where there are frequent deployments. Regression testing involves automating tests to cover new and existing features, balancing optimization and coverage based on risk levels and targeted quality levels.

API testing verifies system interfaces for functional suitability, reliability, performance efficiency, security, and other quality characteristics. It offers fast feedback and integrates well into CI/CD pipelines.

In sequential development models, test automation occurs after development, while in Agile software development it occurs throughout the SDLC. DevOps relies heavily on test automation across all

SDLC phases, creating fast feedback loops and addressing various test needs with practices like containerization.

Hands-On Objective QDO-3.2 (H2) Test Automation in DevOps Teams

For this hands-on objective, the H2 level exercise supports the students in learning how to define the extent and role of test automation in a DevOps context. The focus is not on tools or implementation, but on strategic decisions about:

- What should be automated
- Where automation is essential, useful, or optional
- What should not be automated and why

3.2.1 Regression Testing in CI/CD Pipeline

The purpose of the CI/CD pipeline is to deliver software into production with good quality. Regression testing determines if code changes do not adversely affect the existing application functionality. It is a critical safety net due to the frequent code changes and deployments. Practically, all test automation in the CI/CD pipeline is first used to cover new functionality and then can become regression testing.

Regression tests usually use multiple test environments of the same type and run them in parallel for shorter test execution times. Each test environment supports each test type to have a suitable number of resources and integrations to other environments available (see *Section 3.1.6*).

All tests can theoretically run with every CI/CD pipeline execution. However, even if this were fast enough, it would be bad for business viability and sustainability as it consumes unnecessary resources (i.e., time, cost, and energy). In practice, the DevOps team may create multiple regression test suites with different coverage levels and run them for different purposes, with different frequencies, or based on the changes in the system under test. Some regression tests can run before the merge into the codebase, some after the merge, and some overnight. The DevOps team usually selects regression tests based on the risk (e.g., they can have the CI/CD pipeline run component tests based on heat maps, showing changed areas of code in the latest commit). Tags and filters enable dynamic choice of regression tests. Another example is running regression tests for existing features that are affected by a change.

The CI/CD pipeline should run a full regression test suite before releasing software. The DevOps team should balance the optimization of the number of regression tests and the need for good coverage before a release. Depending on the targeted quality level, regression tests may run only occasionally (e.g., for a security test report). Regression testing should be as fully automated as possible and be part of the CI/CD pipeline design (see *Section 2.2.6*).

3.2.2 Key Points of API Testing

API testing focuses on the interfaces between systems and how they communicate. It involves verifying functional suitability, reliability, performance efficiency, security, and other API quality characteristics. APIs are the backbone of modern applications, and testing them as early as possible prevents costly defects.

API testing is used to determine if the data exchange between systems is consistent, correct, timely, and secure.

Key aspects of API testing

- Functional testing: Determines if the API behaves as expected for all inputs and produces correct outputs.
- Reliability testing: Determines if the API functions as expected under different scenarios.
- Performance testing: Measures response times, throughput, and scalability.
- Security testing: Validates authentication, authorization, and data protection mechanisms.

There are many API testing tools available, and most programming languages provide libraries that can be used.

API testing benefits include:

- Faster feedback compared to GUI testing
- Easy integration into CI/CD pipelines
- Testing early in the SDLC (i.e., shift left)

But there are also challenges:

- Ensuring thorough coverage for complex APIs
- Keeping tests updated with evolving API contracts
- Handling dependencies on external APIs or services

Contract testing

Contract testing is a type of integration testing that tests each application in isolation to determine if the messages each sends or receives conform to a shared understanding that is documented in a "contract" (Fellows, 2022).

Contract testing validates the agreement (i.e., contract) between a provider and its consumers. The provider is accessed through an API, and instead of testing the implementation or business logic behind the API, contract testing checks that the API adheres to a defined structure, format, and set of expectations for request and response payloads, headers, status codes, and data types.

Contract testing is important in microservices architectures, where many independently developed services must integrate reliably.

Contract testing supports the fast feedback loop in DevOps by providing fast, isolated checks to determine if changes will not adversely affect downstream systems.

Implementing contract testing

1. Define the contract: Start by clearly specifying the expected API structure and behavior, including request formats, response fields, status codes, and data types.
2. Write contract tests: Based on the defined contract, consumers specify how they intend to interact with the API. These specifications drive test cases that validate the provider's implementation. Both, the consumer and the provider, use the same contract to verify that the implementation is correct on the provider's side, and that the consumer is calling the API correctly.

3. Run in the CI/CD pipeline: Integrate contract tests into the CI/CD pipeline to provide early feedback and detect breaking changes before deployment.
4. Version and manage contracts: In organizations with multiple teams or services, contracts should be versioned and stored in version control (see *Section 3.1.1*). This allows changes to be tracked, reviewed, and coordinated across services.

Considerations and trade-offs with contract testing

- Contract tests do not cover all functional aspects of API testing, and they supplement other test types.
- Changes to the contract require careful coordination between different teams.
- The development team has to invest in tooling and define shared standards.

3.2.3 Compare Test Automation in Different SDLC Models

Test automation applies to any SDLC, though its timing and emphasis vary (*ISTQB® CTAL-TAE*, v2.0).

Sequential development models introduce test automation beyond unit testing in the testing phase after the development is complete and support a limited feedback loop between the development and testing phases. Agile software development integrates test automation throughout the SDLC. In DevOps, test automation becomes a core enabler of CI/CD, with a broader scope that covers various test levels and test types. It comes with a heavy reliance on IaC (see *Section 4.2.3*) for provisioning test environments and often includes testing in production through practices like blue/green deployment and canary releases (see *Section 2.2.4*).

In sequential development models, test automation beyond unit tests is generally limited to the later phases of development. Through the incremental approach within iterations in Agile software development, there is a higher focus on unit testing, component testing, and component integration testing. In DevOps, the reliance on test automation is key in all CI/CD pipeline stages. This creates fast, automated feedback loops through pre-merge testing, testing in production, and monitoring.

Test automation comes with its unique challenges depending on the SDLC. The main challenge with test automation in DevOps is often managing the complexity of different requirements for the separate test levels (e.g., fully integrated system components in end-to-end testing versus mocked services for component testing). Containerization helps address these challenges by providing isolated, reproducible environments (see *Section 4.2.9*). Cultural and process considerations influence the use of test automation in different SDLCs. Sequential development models see testing as a distinct phase. Agile software development promotes a collaborative mindset between developers, testers, and product owners in regard to testing. DevOps blurs traditional role boundaries further by involving testers in code reviews, monitoring, and observability setups and involving developers in testing and monitoring activities.

3.3 Manual Testing in DevOps Teams

Despite the strong emphasis on automation in DevOps, manual testing remains essential for gathering critical information that automated testing cannot provide. Manual testing allows stakeholders to directly assess usability, gain confidence in software quality, and evaluate aspects that are difficult to automate, such as user experience (UX). Manual testing plays a crucial role within CI/CD pipelines and in production environments through A/B testing and feature toggles (see *Section 4.2.4*).

Key manual testing approaches in DevOps include exploratory testing, crowd testing, and quality hunting, each offering unique benefits for assessing software quality dynamically. These test approaches complement automated testing by providing diverse insights, ensuring that software meets both technical and user expectations.

Hands-On Objective QDO-3.3 (H2) Implement Manual Testing in DevOps Teams

For this hands-on objective, the H2 level exercise supports the students in learning how to identify, implement and debrief manual test approaches in DevOps, e.g., with exploratory testing, crowd testing, or quality hunting, to complement automated testing.

3.3.1 Examples of Manual Testing

One of the DevOps principles is to automate as much as possible. This leads to the belief that there is no room for manual testing in the ultimate CI/CD pipeline. However, testing is about gathering information about the test object quality, so that stakeholders can establish their confidence in the solution, and thus, these stakeholders need various kinds of information (Marselis et al., 2020).

Automated testing in the CI/CD pipeline can provide a lot of information on the test object quality level (see *Section 2.2.4* and *Section 2.2.5*). However, manual testing remains a vital activity for various reasons. One is that manual testing gives other kinds of information than automated testing (e.g., determining the usability of software is difficult to automate but easy to perform with manual testing). Secondly, some stakeholders obtain extra confidence in the software quality level if they see it with their own eyes. Thirdly, manual testing is flexible, uses human judgment, and can be more cost efficient than automated testing, depending on the specific situation.

For those reasons, manual testing is required, both during the CI/CD pipeline activities and possibly during live operation (e.g., in A/B testing or when software has been deployed to the production environment but is not released to the users yet, by applying feature toggles) (see *Section 4.2.4*). The initial manual testing, is static testing of requirements and/or user stories (e.g., in refinements using 4-amigos sessions - with representation of business analysis, development, testing and operations) and reviews of code and related work products as part of pull requests (PRs) (see *Section 2.1.4*). Other relevant approaches to manual testing in DevOps are exploratory testing (see *Section 3.3.2*), crowd testing (see *Section 3.3.3*), and quality hunting (see *Section 3.3.4*). These test approaches provide early dynamic testing that complements the automated testing in the CI/CD pipeline.

3.3.2 Exploratory Testing within CI/CD Pipeline

Some think that exploratory testing just means testing without pre-documented test cases or test procedures, but there is more to it than that. Exploring involves rigorous and analytical investigation (Hendrickson, 2013). Exploration is the counterpart to automation. Where automated feedback comes from tools, the feedback from exploration comes from people (Clokie, 2017).

In exploratory testing, tests are simultaneously designed, executed, and evaluated while the tester learns about the test object. See (*ISTQB® CTFL*, v4.0) for the description of exploratory testing. Exploratory testing can be performed as dynamic testing by a user representative and a team member to investigate behavior in specific situations and gain confidence. Although exploratory tests are manually executed, they can be triggered from the CI/CD pipeline. Tools may be used to support test case creation during exploration (not beforehand), to support test execution, to record test results, and to store certain types of

evidence (e.g., screenshots). The exploratory test results should be registered in the CI/CD pipeline to be included in the quality gates and test reporting, together with automated test results.

Exploratory test charters can be defined in an early phase (e.g., during user story refinement), to have a backlog of testing work items as part of the DevOps team backlog (see *Section 2.2.1*). Test charters can also be created when a certain need for manual testing arises (e.g., based on unexpected automated test results in the CI/CD pipeline).

Exploratory tests can be executed in various test environments, ranging from a specific temporary testing environment created from the CI/CD pipeline to the staging environment or in the production environment (e.g., for some specific purposes controlled by feature toggles) (see *Section 2.2.5*).

IaC may be used to explore the impact on software from various versions of infrastructure. This approach provides documented evidence of compatibility across platforms, allowing teams to identify and trace infrastructure-specific defects to their root causes before deployment.

3.3.3 Crowd Testing

Crowd testing refers to using a virtual group of people who are usually external to the organization and may be geographically distributed, who represent real users, use real devices, and individually execute tests simultaneously. Crowd testing is used, for example, in beta testing after a canary release, in A/B testing, in post-release feedback collection, and in real-world device testing. Crowd testing usually applies a test approach similar to exploratory testing, where the members of the crowd get a test charter to guide their testing without preparing detailed test cases upfront.

The key to the success of crowd testing is designing and executing many tests, quickly gathering extensive software quality information.

The focus of crowd testing is on breadth first, to cover many different personas, or combinations of software and hardware (as in beta testing of games or mobile apps).

A main advantage of crowd testing is that many different situations and configurations can quickly be tested. Specialized organizations manage the selection of crowd testers, distribute the test charters, and gather the test results.

A disadvantage when using testers outside the DevOps team is that the preparation requires more effort from the DevOps team (especially when the tests need to be performed in a certain order), and the preparation may need to be more in-depth (e.g., by defining specific personas) compared to the DevOps team members performing the tests themselves.

Reporting on the crowd testing results may be a challenge, for example, because of the large amount of data gathered in a short timeframe. The DevOps teams should review the defect reports to prevent unjustified feedback. The crowd testers need good instructions about what information to gather and how to register their test results. Some organizations that organize crowd testing actively support giving these instructions and performing these reporting tasks.

Risks and challenges to consider when applying crowd testing are:

- Confidentiality: The larger the crowd and the more it is outside the DevOps team sphere of influence, the harder it becomes to manage confidentiality.
- Knowledge: Not every application is suited to crowd testing from a knowledge perspective. Many applications used within an organization require specific knowledge of their products and processes to run the software.

- Motivation: The rewards method may influence the effectiveness (e.g., testers who are paid "by the bug" will often look for easy to find defects instead of looking for the most critical ones).

3.3.4 Quality Hunting Events

Software can only be delivered with the right quality at the right moment if there is a clear view of what quality level is required to get the pursued business value. The people involved need a clear view of what the quality level currently is. Static testing and dynamic testing together determine the current quality level. The image of quality also depends on the impression that stakeholders have from their personal experience. To support building such personal experience, a DevOps team can organize quality hunting, a collaborative activity with a gamification approach (note: this is different from bug hunting) (Ruigrok et al., 2025).

Quality hunting is an event where quality hunting teams (e.g., half the size of an Agile team) compete to give the most valuable assessment of their test object quality level in a limited time. The main goal of quality hunting is to get a shared view of the test object quality level. By having multiple quality hunting teams competing to get the best view of the current quality level, these teams are challenged to create fresh and innovative ways to assess the quality and to give a good insight into one or more quality level aspects. During quality hunting findings are recorded similar to the debriefing information in exploratory testing (see (*ISTQB® CTFL*, v4.0)). Defects found during quality hunting are handled according to the organization's defect management process. Information from multiple teams usually complements each other, and extends the information obtained with testing and other quality engineering (QE) activities.

The stakeholders benefit from quality hunting because, in a short time, many ideas and new angles for relevant quality aspects are investigated so they get a broad and varied overview of their system quality.

Quality hunting is about getting the most valuable information about the software quality level. The game element consists of a (usually small) reward for the quality hunting team that gives the most informative report and/or the most specific angle about the quality level. Quality hunting is a different and broader activity than bug hunting (introduced in (*ISTQB® CT-MAT*, v2019)). That is a gamified test approach that motivates finding defects, which may lead testers to try to find irrelevant defects without getting a view on the quality level and risk level.

Quality hunting is a manual testing activity integrated into the CI/CD pipeline step to test the overall business process.

The interaction between the different quality hunting teams when discussing their quality hunting results contributes to the culture of collaboration and learning in DevOps.

4 Tools and Practices in DevOps - 200 minutes

Keywords

fault injection, fault tolerance, hotfix, performance testing, security testing, unit testing

Quality in DevOps Specific Keywords

build artifact, branching, branching strategy, CI/CD pipeline, chaos engineering, containerization, dependency scanning, feature toggle, infrastructure as code, merging, observability, release management, release strategy, software bill of materials, source code management, version control, virtualization

Learning Objectives in Chapter 4:

4.1 Select Tools Supporting DevOps Within the Organization

QDO-4.1.1 (K1) Recall the capabilities of tools in DevOps

QDO-4.1.2 (K2) Explain the tools supporting quality assurance and testing in DevOps

4.2 Understand Technologies and Practices to Support Quality in DevOps

QDO-4.2.1 (K1) Recall the non-testing activities within the CI/CD pipeline

QDO-4.2.2 (K2) Explain the different key release strategies in DevOps

QDO-4.2.3 (K2) Summarize the concepts of infrastructure as code

QDO-4.2.4 (K1) Recall feature toggles as a way to separate deployment from release

QDO-4.2.5 (K1) Recall branching strategies

QDO-4.2.6 (K1) Recall chaos engineering

QDO-4.2.7 (K1) Recall telemetry and observability

QDO-4.2.8 (K1) Recall software bill of materials to support configuration management

QDO-4.2.9 (K1) Recall containerization

Hands-On Objectives:

4.1 Tools Supporting DevOps within the Organization

QDO-4.1 (H2) Tools and Practices in DevOps

4.2.3 Infrastructure as code (IaC)

QDO-4.2.3 (H0) Summarize the Concepts of Infrastructure as Code (IaC)

4.1 Tools Supporting DevOps within the Organization

DevOps tools enable capabilities such as continuous discovery, continuous integration (CI), continuous testing, continuous delivery (CD), continuous deployment, and continuous monitoring by automating workflows, enhancing collaboration, and improving software delivery reliability. These tools are typically in categories such as version control, CI/CD, configuration management, management, monitoring and logging, testing, security and compliance, collaboration, and planning. These tools streamline essential activities. A well-integrated DevOps toolset boosts speed, efficiency, and reliability, helping the organization meet its business goals by supporting high-quality, rapid, and secure software releases. An inclusive and holistic source of information about DevOps tools is the DevSecOps tools periodic table (digital.ai, 2023).

Hands-On Objective QDO-4.1 (H2) Tools and Practices in DevOps

For this hands-on objective, the H2 level exercise supports the students in choosing different tools for the CI/CD pipeline including both test automation and other automation tools.

4.1.1 Capabilities of Tools

This section explores how tools for QE, including QA and testing, are integrated into the value stream's activity groups, specifically in continuous discovery, continuous integration (CI), continuous delivery (CD), continuous deployment, and release on demand (*Section 2.2*). Below is a summary of the key categories, their descriptions, and related capabilities of these tools.

1. Information management: Manages version control, collaboration, and tracking changes in documents and backlog items, ensuring a single source of truth (SSOT) (e.g., project and product documentation, requirements, defect reports, and tasks)
   Capabilities: version control, traceability, transparency, and change control
2. Source code management: Manages version control, collaboration, and tracking changes in codebases, ensuring an SSOT
   Capabilities: version control, branching, merging, and conflict resolution
3. Continuous integration: Automates the integration of code changes into shared repositories of running software, fostering early defect detection and continuous feedback loops
   Capabilities: CI servers, automated build pipelines, and integration testing
4. Continuous deployment: Automates the release of validated code to any environment, reducing manual errors and ensuring consistent deployments
   Capabilities: deployment pipelines, deployment automation, roll forward, and rollback mechanisms
5. Environment and infrastructure management: Automates the provisioning and configuration of consistent environments across development, testing, and production
   Capabilities: Infrastructure as code (IaC), environment provisioning, and configuration management
6. Build artifact management: Facilitates efficient storage, retrieval, and distribution of build artifacts, dependencies, and packages, ensuring consistency and repeatability across environments
   Capabilities: build artifact storage, dependency management, dependency scanning, scanning for vulnerabilities, and secure distribution

7. Code quality and security: Enhances software quality and resilience by detecting defects and vulnerabilities early, ensuring adherence to secure coding practices throughout development Capabilities: static analysis, static and dynamic application security testing (SAST/DAST), software composition and dependency analysis (SCA), supply-chain security validation ((*ISTQB® CT-SEC*, v1.0)), unit testing, and code coverage assessment
8. Integration and test automation: Validates interactions between system components and overall functionality while automating tests for efficiency Capabilities: contract testing, API testing, component integration testing, UI testing, and reusable test automation frameworks
9. Performance testing and security testing: Evaluates application scalability, fault tolerance, and security by simulating load conditions and scanning for vulnerabilities (*ISTQB® CT-PT*, v2018) and (*ISTQB® CT-SEC*, v1.0)
   Capabilities: performance testing, load testing, stress testing, spike testing, volume testing, endurance testing, vulnerability scanning, and penetration testing
10. Telemetry (monitoring and observability): Provides real-time insights into application and infrastructure health, helping identify defects proactively
    Capabilities: performance monitoring, log aggregation, log analysis, distributed tracing, and alerting
11. Chaos engineering and fault injection: Simulates failures to test system resilience and fault tolerance, identifying weaknesses under stress, or any unexpected conditions
    Capabilities: fault injection, resilience testing, and fault tolerance validation
12. Test data: Generate, manage, and secure test data to support reliable and compliant test processes
    Capabilities: synthetic test data generation, anonymization, masking, sub-setting and cloning production-like test data, on-demand provisioning of test data, and compliance with privacy regulations (e.g., General Data Protection Regulation (GDPR))
13. Test management: Organize and control the test process by managing test cases, test plans, test results, and traceability
    Capabilities: test case authoring and version control, test plan creation and scheduling, traceability between requirements, test cases, and defects, real-time reporting and dashboarding, and integration with CI/CD pipelines and defect tracking systems (see (*ISTQB® CTFL*, v4.0))
14. Collaboration and communication: Enhances team productivity and alignment through streamlined communication and project management tools
    Capabilities: real-time messaging, project management, document sharing, and versioning
15. Operations quality and reliability: Supports production environments by ensuring operational reliability through backup, restore, and incident management (AXELOS, 2019) Capabilities: data backup, recovery, incident management, and vulnerability scanning

These categories form a structured framework within the CI/CD pipeline, with each tool serving a unique purpose to create a continuous, reliable, and efficient development and operations environment. Aligned with the DevOps workflow, each category integrates seamlessly into the DevOps loop, fostering feedback loops for continuous improvement from development to deployment and production.

4.1.2 Tools Supporting Quality Assurance (QA) and Testing

In a DevOps environment, tools supporting QA and testing are fundamental in delivering reliable, high performing and secure software at a rapid pace. These tools seamlessly integrate into the CI/CD pipeline, embedding quality into every SDLC phase (see *Section 2.2*). Automating and streamlining various test

activities empowers DevOps teams to detect defects early, maintain code quality, and provide information about software quality. The tools can be grouped by function.

During the coding and integration phases, tools focus on ensuring the control and integrity of the codebase and its components (see *Section 4.2.8*).

- Static analysis and code quality tools evaluate source code without execution, helping to enforce coding standards, detect potential defects, and identify code smells or security vulnerabilities.
- Unit testing tools verify individual units of code, giving developers fast feedback.
- Dependency scanning tools analyze third-party libraries for security risks, outdated packages, or license issues, helping maintain a secure and compliant codebase.
- Integration testing and API testing tools verify that connected components or services work together correctly and meet interface expectations.

As the work on the software moves closer to deployment, tools designed for functional testing and UI testing, performance testing, security testing, and test data management take precedence.

- Functional testing and UI testing tools simulate user behavior to validate that the software meets functional requirements from the end-user perspective.
- Performance testing tools assess a system's behavior under different load conditions, identify performance bottlenecks, and ensure scalability.
- Security testing tools proactively detect vulnerabilities, such as injection defects or misconfigurations, reducing exposure to threats.
- Test data management tools generate and manage realistic, anonymized, or synthetic test data, ensuring test conditions reflect real-world conditions without violating data privacy.

Frameworks and tools supporting automation, test management, and environment provisioning play a vital role in enabling effective testing. (See (*ISTQB® CTFL*, v4.0) section 6.1 Tool Support for Testing)

- Environment provisioning tools automate the creation of test environments that mirror production configurations, supporting reliable and consistent test execution.
- Configuration management tools ensure consistency in environment settings and application configurations across development, testing, and production environments. (See (*ISTQB® CTFL*, v4.0), section 5.4. Configuration Management) This consistency reduces errors and enhances the accuracy of test results, supporting reliable and repeatable test processes.

Together, these tools support DevOps teams in achieving and maintaining the required quality.

4.2 Technologies and Practices to Support Quality in DevOps

Technologies and practices in DevOps support quality by integrating non-testing activities, effective branching strategies, infrastructure automation, release management and feature toggling. Non-testing activities, such as building, packaging, and deployment, automate key steps within the CI/CD pipeline, while release strategies like canary releases and blue-green deployments control the rollout process. IaC enables environment consistency, feature toggles decouple release from deployment, release management and branching strategies optimize collaboration. These technologies and practices create

a cohesive environment that ensures consistency, reliability, security, and speed in software delivery.

4.2.1 Non-testing Activities within a CI/CD Pipeline

A CI/CD pipeline automates several processes in the SDLC. Breaking these processes into several sequential stages ensures smooth and efficient delivery. Each stage has specific tasks involving non-testing activities critical to the CI/CD pipeline's success.

Here is a list of these non-testing activities: (see *Section 2.1.4* and *Section 3.1.1*)

1. Source code management: Overseeing code commits and version control within a shared repository to maintain collaboration and organization.
2. Branching and merging: Performing tasks such as branching, merging, and creating pull requests (PRs) to efficiently organize and review code changes.
3. Build artifacts and dependency management: (see *Section 4.2.8*) Resolving software dependencies, compiling code, and generating versioned build artifacts. These build artifacts are stored for later stages in the CI/CD pipeline.
4. Conflict resolution and coding standards: (see *Section 3.1.6* and *Section 4.2.3*) Addressing code conflicts, provisioning integration environments, and running static analysis to enforce coding standards and ensure code quality.
5. CI/CD pipeline configuration: Setting up CI/CD pipelines and automating the provisioning of isolated, consistent environments, often achieved through containerization tools. Additionally, managing environment-specific configurations, feature toggles via command-line arguments or environment variables, to support flexibility and scalability (see *Section 4.2.4*).
6. Release management: (see *Section 2.2.5*) Coordinating and approving releases, managing rollback strategies (see *Section 4.2.2*), and tagging or labeling versions. This includes implementing deployment strategies such as blue-green deployment or canary releases.
7. Monitoring and feedback: Using tools to monitor system health and establish feedback loops for continuous improvement. This involves configuring logging and telemetry systems, analyzing metrics to assess application performance and trends, and setting up alerts for critical events. (see *Section 4.2.7*) These activities provide critical insights into the application's health and stability post-deployment in production.

4.2.2 Key Release Strategies

DevOps release strategies outline controlled, efficient, reliable delivery of software updates and new features to users. These release strategies focus on the following points:

- to minimize risks by reducing downtime, enabling gradual exposure to changes, and proactively detecting incidents (AXELOS, 2019)
- to optimize deployment processes, by increasing release velocity, eliminating repetitive tasks, and shortening development-to-production time
- to deliver seamless customer value by ensuring uninterrupted UX and enabling rapid incident resolution by operational teams (AXELOS, 2019)

Understanding these release strategies can help organizations select the most suitable strategy based on their specific and/or operational needs.

DevOps has several key release strategies, each with its own purpose and use case.
These strategies include:

1. Canary releases (see *Section 2.2.4*)
2. Blue-green deployments (see *Section 2.2.4*)
3. Rolling releases: Gradually replacing system instances with updated versions, ensuring continuous service availability. Commonly used in large-scale systems, this approach mitigates risks of widespread disruptions.
4. Phased rollout: Deploying updates incrementally by region, user group, or other segmentation criteria. This allows monitoring and issue resolution in phases.
5. Progressive delivery: Combining elements of canary releases and phased rollouts, incrementally releasing updates to targeted users or regions for precise validation and feedback collection.
6. Hotfix deployment: An urgent release to resolve critical defects in production, prioritizing speed over thorough testing.
7. Dark launching (see *Section 2.2.4*): Deploying features to production but marking them inaccessible to users. This allows teams to test and monitor performance under real-world conditions without affecting the UX.

The key release strategies vary based on an organization's unique needs and context, as many release strategies exist. However, the strategies mentioned above represent the most essential choices aligned with modern DevOps practices. Additionally, organizations often combine multiple release strategies to optimize their daily operations (e.g., releasing a hotfix using phased rollout).

4.2.3 Infrastructure as code (IaC)

IaC manages and provisions infrastructure through machine-readable definition files, treating infrastructure as software. It uses descriptive models and version-controlled code for consistency and repeatability. QA in IaC involves automated static testing, peer reviews, and configuration validation to determine the degree of reliability, security, and compliance before deployment.

With IaC, teams edit the related IaC source code instead of modifying changes to the target environment. This practice integrates seamlessly with DevOps workflows, enabling automated provisioning and management, ensuring consistency, scalability, and efficiency. IaC supports CD by enabling reproducible, predictable infrastructure environments (see *Section 3.1.6*).

IaC relies on several core concepts to deliver its benefits effectively. First, IaC uses declarative and imperative approaches to define infrastructure. The declarative approach focuses on specifying the desired state of infrastructure (e.g. ensuring that three servers are always running), while the imperative approach provides detailed instructions on how to achieve that state (e.g., installing software and configuring settings).

Another critical concept is idempotency (Kundu, 2019), which ensures that IaC scripts produce the same results even when applied multiple times, preventing configuration drift (Moustakis, 2024). IaC integrates seamlessly with configuration management systems, allowing teams to track changes, collaborate efficiently, and roll back to earlier configurations if needed (see *Section 3.1.1* and *Section 4.1.1*). IaC

eliminates the need for repetitive manual tasks through automation, ensuring consistency and reducing human error. It supports environment parity, keeping identical configurations across development, testing, and production environments. This reduces the number of situations like "it works on my machine" and enables smoother transitions between test levels.

Hands-On Objective QDO-4.2.3 (H0) Summarize the Concepts of Infrastructure as Code (IaC)

For this hands-on objective, the H0 level exercise supports the students in understanding the IaC concept.

4.2.4 Feature Toggles

Using feature toggles can help DevOps teams to modify system behavior to deliver new functionality to users rapidly but safely without changing the code (Fowler, 2017). It provides a structured approach to managing feature rollout, enabling incremental feature releases, risk mitigation, and controlled experimentation. In a production environment, feature toggles define which users or environments can access a specific feature. For example, a feature can be disabled for most users while being activated for testers, ensuring controlled validation before broader deployment. Using feature toggles supports the separation of deployment from release.

Feature toggles are categorized by their intended use:

- Release toggles: Enable new features in line with release on demand goals (e.g., a canary release) see *Section 2.2.4*
- Operational toggles: Allow real-time control over system behavior
- Experiment toggles: Support A/B testing by exposing distinctive features to user segments
- Permission toggles: Control feature access for specific users or roles

Feature toggles introduce complexity. Managing feature toggles requires systematic tracking and organization to avoid excessive feature toggles accumulating over time. Since feature toggles add conditional logic to the codebase, it is essential to monitor, document, and retire obsolete feature toggles to maintain system clarity. DevOps teams should test feature toggles implementations with both on and off settings and test interdependent features to reduce risk.

4.2.5 Branching Strategies

Branching allows parallel development within and between DevOps teams, providing multiple work streams simultaneously minimizing conflicts and integration defects. Integrating these code changes from one branch to another branch is called merging. A merge conflict occurs when both branches modify the same section of code. The chances of merge conflicts depend on multiple factors (e.g., the size of the changes to merge, or the number of developers who introduce changes). Resolving these conflicts can take much time and effort.

While VCSs support branching, merging, and conflict detection, DevOps teams should implement proper branching strategies to reduce the risk of conflicts while ensuring collaboration and quality.

Since all the code and testware contributing to software development is in a VCS (see *Section 3.1.1*), branching strategies also apply to it, and must be understood by every DevOps team member.

From the multiple branching strategies defined, DevOps teams usually apply the following two to achieve CD capability:

- Trunk-based development: After successful tests during the commit stage, every change is pushed directly to the main branch (i.e., formerly referred to as trunk or master). This approach requires a mature team and high level of test automation within the CI/CD pipeline.
- Feature branches: They are created for specific changes and deleted after merging. A PR is used to merge changes to the main branch, or to get feedback from other DevOps team members during development (see *Section 2.1.4*). These feature branches should have a short life span, a maximum of a couple of days, with the aim of decreasing it to a few hours.

4.2.6 Chaos Engineering

Chaos engineering is a discipline that involves experimenting on a system to build confidence in its ability to withstand turbulent conditions in production (Ranganathan, 2023). This practice is crucial for determining the resilience and reliability of software systems by intentionally injecting failures and observing how the system responds.

Chaos engineering simulates failures at various levels, ranging from individual computing instances (e.g., virtual machines or containers running applications) to entire geographic regions of cloud infrastructure, where multiple data centers operate together (Netflix, 2025). These experiments help teams to understand how services behave during disruptions and to determine service continuity. The insights gained from these experiments lead to improvements in system design, operational practices, and incident response strategies.

Chaos engineering emphasizes the importance of understanding the system's behavior under stress. DevOps teams can uncover hidden defects that may not be apparent during normal operation or in a test environment. This proactive discipline identifies potential weaknesses before they impact users, enhancing software quality.

Chaos engineering is made possible by a highly effective, fast, and reliable CI/CD pipeline, which supports deploying changes to production and rollback painlessly, minimizing user impact. Public cloud providers usually offer fault injection tools to support chaos engineering in a controlled way.

4.2.7 Telemetry and Observability

When software is released to the production environment, continuous monitoring gathers information on the software quality, and when the quality is not progressing as expected, control actions are initiated. Continuous monitoring observes the software in its environment, including other software, hardware, and the people involved. Continuous monitoring in many organizations starts with the production environment, but it may be relevant for any other environment, especially test environments. Telemetry (gathering data) and Observability (analyzing data) enable continuous monitoring (which checks against KPIs).

Observability uses logging, tracing, and metrics to provide information and create a coherent view of the software's state. This information is used for reactive and proactive maintenance, auditability, controllability, and debugging purposes (Marselis et al., 2020). The information is automatically gathered through telemetry. Telemetry collects measurements for functional and non-functional quality characteristics, which serve different purposes and goals.

Examples of telemetry are:

- Production telemetry measures how software runs, especially when cloud infrastructure is used (e.g., to measure memory usage, CPU load, latency, and requests counts).
- User telemetry measures user interaction, which gives essential information about the success of the software (e.g., click tracking for feature usage, form completion rate tracking, navigation flow analysis, and session duration monitoring).
- Endpoint telemetry and other security monitoring approaches enable threat detection, forensic analysis, and compliance reporting.

Monitoring (i.e., using telemetry and observability) can apply artificial intelligence (AI), such as machine learning, to the log data, to better and more easily understand software behavior and notice any anomalies.

Note that team telemetry can be used to measure the DevOps team maturity (see section *Section 1.1.3* for DORA metrics) and suggest how the DevOps team can improve (see the end of section *Section 2.2.5* for more information about team learning).

4.2.8 Software Bill of Materials (SBOM)

While developing software, DevOps teams often use third-party components. These include frameworks, middleware and libraries that provide specific functionality, and code from other DevOps teams. Reuse speeds up development and brings consistent software development within organizations. Reusability (as part of maintainability) (see (*Systems and software engineering (25010)*, 2023)) is an important quality characteristic and sign of properly built components. When reusing components, it is essential for the SDLC management, future maintenance, and for transparency to have information about each component including the components that the DevOps team creates itself, and the relationships between components.

The following information is required:

- Component identification (e.g., name or number)
- The current component version
- The component's author and/or owner
- The component functionality
- Consistency of the component with other components
- Dependencies of the component on other components
- Conflicts of the current version with previous versions
- Cryptographic hash(es) to authenticate the component(s)
- Component licensing conditions

This information is registered in an SBOM (Marselis et al., 2020). DevOps teams should maintain an SBOM for their software using tooling integrated into the CI/CD pipeline. To ensure that the SBOM is up to date, it should be part of the DevOps team's definition of done. Apart from maintainability, an SBOM supports vulnerability management which is part of security by associating known vulnerabilities to components used. Whenever a vulnerability is detected, the CI/CD pipeline will stop, to prevent insecure software deployment. After that, the DevOps team can take necessary actions to resolve such vulnerabilities.

4.2.9 Containerization

Containerization is a lightweight form of virtualization that packages an application and its dependencies into a single executable unit called a container.

Containers allow the DevOps team to build consistent environments for every test level and eliminate the pitfalls of testing locally, as in “it works on my machine” (see *Section 3.1.6*). Another benefit is that it allows for multiple tests to run in parallel without interfering with one another and keeping test data clean (see *Section 3.1.4*). A container can be deployed on demand, and removed once the test run has finished. This helps to provision environments when they are actively being used and to quickly provide additional environments when demand increases. Since containers are defined and stored as code in a shared codebase, any environment changes made centrally can be automatically applied to future environments with the same configuration.

Containerization comes with a few known challenges. Because each container is a clean representation of the environment, handling stateful application testing can be difficult and requires extra setup steps. Security can be a concern through the use of third-party components, and needs diligent dependency management (see *Section 4.2.8*).

5 Quality in DevOps specific terms

Term Name	Definition
build artifact	A work product that is generated by the build process and can be used for deployment.
blue-green deployment	A deployment strategy in which two identical production environments, one active and one idle are maintained to test a new software version in the idle environment before switching all traffic from the active environment.
branching	The process of working on software changes independently by duplication of source code under version control into a change related branch and merging them later into another branch.
branching strategy	A set of rules for managing branches in version control systems.
CALMS	The abbreviation that stands for culture, automation, lean, measurement, and sharing used to structure DevOps practices.
change fail percentage	The number of deployed changes resulting in failures, divided by the total number of deployed changes expressed as a percentage.
change lead time	The time from when a commit is made until it reaches production.
chaos engineering	The practice of randomly injecting failures to gather information about system resilience.
CI/CD pipeline	The automated series of steps to deliver a new software version.
code smell	A characteristic of source code that indicates a potential design or maintainability problem, without necessarily preventing the code from functioning
containerization	A lightweight form of virtualization that packages an application and its dependencies into a single executable unit called a container.
continuous delivery (CD)	An automated software development procedure in which code changes are automatically built, tested, and are deployable to production.
continuous deployment	An automated software release procedure where code changes that passed all tests are automatically deployed to production without manual intervention.
continuous discovery	An ongoing process of user research and stakeholder feedback to make informed decisions, improve product, and ensure delivery of real value to users.

Term Name	Definition
continuous learning and experimentation	A DevOps principle that fosters a high trust culture of ongoing improvement by encouraging teams to experiment, take calculated risks, and learn from successes and failures.
continuous monitoring	An automated process of collecting performance indicators to understand user and system behavior in real time.
cross-functional DevOps team	A group of people with different but overlapping sets of knowledge, skills, and capabilities working together toward a common goal to create and operate a system.

Term Name	Definition
dark launch	A deployment strategy in which a new feature or software change is deployed without exposing it to users, allowing teams to test and monitor performance in production.
dependency scanning	The process of identifying and analyzing software dependencies to reduce risks introduced by external components.
deployment frequency	The rate of software releases to production.
DevOps	A mindset, culture, and set of technical practices that support the integration, automation, and collaboration needed to effectively develop and operate a system.
failed deployment recovery time	The duration needed to restore normal operation after a failed deployment.
feature toggle	A mechanism to modify system behavior without changing code.
feedback loop	The continuous exchange of information to improve products, services, and processes.
flow	The smooth, linear, and fast movement of work products from step to step in a relevant value stream.
idempotency	The property of an operation that produces the same result whether executed once or multiple times.
infrastructure as code (IaC)	The practice of managing and provisioning IT infrastructure using machine-readable configuration files and applying software development practices to it.
merging	The practice of integrating code changes from one branch to another branch.
observability	The ability to measure the internal state of a system by examining its output.
over-provisioning	The practice of allocating more computing resources than necessary to a component or system to improve performance or reliability.
personally identifiable information (PII)	Any information connected to a specific individual that can be used to uncover that individual's identity
pull request (PR)	A mechanism that allows notifying team members about changes that were made to a codebase, proposing these changes to be reviewed, discussed, and merged.

Term Name	Definition
release management	A process of planning and scheduling releases, including testing, and deploying software.
release on demand	A practice to release new features based on stakeholders' needs, decoupling deployment and release.
release strategy	A strategy to determine how to deploy software to production, how to release it to users, and how often.

Term Name	Definition
service level agreement (SLA)	An agreement with a customer that a service level objective will be met over a certain period.
service level indicator (SLI)	A quantifiable measure of service reliability
service level objective (SLO)	A reliability target for a service level indicator
single source of truth (SSOT)	A practice for data normalization using one source for a specific data element.
site reliability engineering (SRE)	An approach that applies software development practices and principles to infrastructure and operations while balancing service reliability against the pace of new feature development using automated feedback loops.
software bill of materials (SBOM)	A machine-readable record detailing software components and their supply chain relationships, ensuring transparency, auditability, and traceability.
software binary	The file resulting from compiling source code that is written in a compiled language.
source code management (SCM)	The practice of tracking, managing, and storing changes to software, enabling version control, parallel development, and conflict resolution among contributors.
statistical analysis	A technique for gathering, analyzing, interpreting, presenting, and deriving conclusions from data.
telemetry	An automated collection, aggregation, and analysis of metrics from applications and infrastructure in production to monitor system health and support defect identification.
under-provisioning	The practice of allocating fewer computing resources than necessary to a component or system, leading to performance degradation and potential system failures.
value stream	The end-to-end sequence of activities required to deliver a product or service to a customer, including the flow of information and materials.
version control	A process that records changes to configuration items over time.
version control system (VCS)	A software tool that automates version control.
virtualization	A technique to enable the delivery of virtual services which are deployed, accessed and managed remotely.
wall of confusion	The miscommunication between development and operations teams within an organization.

6 Trademarks

List the trademarks referred to in this syllabus in alphabetical order.

DASA® is a registered trademark of the DevOps Agile Skills Association.
DORA® is a registered trademark of the DevOps Research and Assessment.
ISTQB® is a registered trademark of the International Software Testing Qualifications Board.
TMMi® is a registered trademark of the TMMi Foundation.

7 Appendix A – Learning Objectives/Cognitive Level of Knowledge

The specific learning objectives applying to this syllabus are shown at the beginning of each chapter. Each topic in the syllabus will be examined according to the learning objective for it.

The learning objectives begin with an action verb corresponding to its cognitive level of knowledge, as listed below.

Level 1: Remember (K1)

The candidate will remember, recognize, and recall a term or concept.

Action verbs: Recall, recognize.

Examples
Recall the concepts of the test pyramid.
Recognize the typical objectives of testing.

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic and can summarize, compare, classify, and give examples for the testing concept.

Action verbs: Classify, compare, differentiate, distinguish, explain, give examples, interpret, summarize

Examples	Notes
Classify test tools according to their purpose and the test activities they support.	
Compare the different test levels.	Can be used to look for similarities, differences or both.
Differentiate testing from debugging.	Looks for differences between concepts.
Distinguish between project and product risks.	Allows two (or more) concepts to be separately classified.
Explain the impact of context on the test process.	
Give examples of why testing is necessary.	
Infer the root cause of defects from a given profile of failures.	
Summarize the work product review process activities.	

Level 3: Apply (K3)

The candidate can carry out a procedure when confronted with a familiar task or select the correct procedure and apply it to a given context.

Action verbs: Apply, implement, prepare, use

Examples	Notes
Apply boundary value analysis to derive test cases from given requirements.	Should refer to a procedure/technique/process etc.
Implement metrics collection methods to support technical and management requirements.	
Prepare installability tests for mobile apps.	
Use traceability to monitor test progress for completeness and consistency with the test objectives, test strategy, and test plan.	Could be used in a LO that wants the candidate to be able to use a technique or procedure. Similar to 'apply'.

Cognitive levels of learning objectives are based on (Anderson et al., 2001).

8 Appendix B – Business Outcomes traceability matrix with Learning Objectives

This section lists the traceability between the Business Outcomes and the Learning Objectives of Quality in DevOps.

Business Outcomes: Quality in DevOps		QDO-BO1	QDO-BO2	QDO-BO3	QDO-BO4	QDO-BO5	QDO-BO6	QDO-BO7	QDO-BO8	QDO-BO9
QDO-BO1	Explain How Quality Assurance Is Supported by and Contributes to the DevOps Concepts	6								
QDO-BO2	Understand the Concepts of Implementing DevOps in an Organization		3							
QDO-BO3	Contribute to All Value Stream Stages to Assist in Quality Engineering			4						
QDO-BO4	Contribute to the DevOps Loop Implementation				6					
QDO-BO5	Describe How Automation Supports Quality Assurance in DevOps					6				
QDO-BO6	Implement Test Automation in DevOps Teams						3			
QDO-BO7	Implement Manual Testing in DevOps Teams							4		
QDO-BO8	Select Tools Supporting DevOps Within the Organization								2	
QDO-BO9	Understand Technologies and Practices to Support Quality in DevOps									9

Business Outcomes: Quality in DevOps		QDO-BO1	QDO-BO2	QDO-BO3	QDO-BO4	QDO-BO5	QDO-BO6	QDO-BO7	QDO-BO8	QDO-BO9
LO Number	Learning Objective (K-Level)									
1	Fundamentals of DevOps - 140 minutes									
1.1	Explain How Quality Assurance is Supported by and Contributes to the DevOps Concepts									
QDO-1.1.1	Recall key concepts of DevOps (K1)	X								
QDO-1.1.2	Recall the wall of confusion concepts in DevOps (K1)	X								
QDO-1.1.3	Explain DORA metrics of delivery performance and operational performance (K2)	X								
QDO-1.1.4	Differentiate the elements of CALMS in terms of quality assurance (K2)	X								
QDO-1.1.5	Explain how quality assurance supports the three ways of DevOps (K2)	X								
QDO-1.1.6	Explain the benefits, risks, and pitfalls of DevOps (K2)	X								
1.2	Understand the Concepts of Implementing DevOps in an Organization									
QDO-1.2.1	Distinguish the DevOps team roles (K2)		X							
QDO-1.2.2	Explain the concept of site reliability engineering (SRE) related to DevOps (K2)		X							
QDO-1.2.3	Distinguish the different DevOps team patterns and anti-patterns (K2)		X							
2	Quality Assurance (QA) and Testing in DevOps - 360 minutes									
2.1	Contribute to All Value Stream Stages to Assist in Quality Engineering									
QDO-2.1.1	Implement quality assurance activities in a DevOps context (K3)			X						
QDO-2.1.2	Compare test objectives that support DevOps with sequential development models and Agile software development (K2)			X						

Business Outcomes: Quality in DevOps		QDO-BO1	QDO-BO2	QDO-BO3	QDO-BO4	QDO-BO5	QDO-BO6	QDO-BO7	QDO-BO8	QDO-BO9
QDO-2.1.3	Explain continuous testing in a DevOps context (K2)			X						
QDO-2.1.4	Explain how pull requests support quality (K2)			X						
2.2	Contribute to the DevOps Loop Implementation									
QDO-2.2.1	Explain quality assurance and testing in continuous discovery and its practices (K2)				X					
QDO-2.2.2	Explain quality assurance and testing in continuous integration and its practices (K2)				X					
QDO-2.2.3	Explain quality assurance and testing in continuous delivery and its practices (K2)				X					
QDO-2.2.4	Explain quality assurance and testing in continuous deployment and its practices (K2)				X					
QDO-2.2.5	Explain quality assurance and testing in release on demand and its practices (K2)				X					
QDO-2.2.6	Apply quality assurance and testing knowledge to implement a CI/CD pipeline (K3)				X					
3	Automated and Manual Tasks in DevOps - 510 minutes									
3.1	Describe How Automation Supports Quality Assurance in DevOps									
QDO-3.1.1	Explain how a single source of truth for testware supports quality assurance (K2)					X				
QDO-3.1.2	Explain how automation supports traceability between the test basis and testware (K2)					X				
QDO-3.1.3	Implement uniform quality reporting practices in the CI/CD pipeline with information from both automated testing and manual testing (K3)					X				
QDO-3.1.4	Explain the benefits of test data management automation (K2)					X				
QDO-3.1.5	Explain the benefits of statistical analysis of test results over long periods of time (K2)					X				
QDO-3.1.6	Explain the benefits of standardized, controlled, and automated test environment management (K2)					X				

Business Outcomes: Quality in DevOps		QDO-BO1	QDO-BO2	QDO-BO3	QDO-BO4	QDO-BO5	QDO-BO6	QDO-BO7	QDO-BO8	QDO-BO9
3.2	Implement Automated Testing in DevOps Teams									
QDO-3.2.1	Explain how regression testing integrates into various CI/CD pipeline activities (K2)						X			
QDO-3.2.2	Prepare API testing (K3)						X			
QDO-3.2.3	Compare the extent of test automation in DevOps to Agile software development and sequential development models (K2)						X			
3.3	Implement Manual Testing in DevOps Teams									
QDO-3.3.1	Give examples of manual testing for a DevOps organization (K2)							X		
QDO-3.3.2	Explain how exploratory testing can work with a CI/CD pipeline (K2)							X		
QDO-3.3.3	Explain how crowd testing can support the culture of feedback (K2)							X		
QDO-3.3.4	Apply quality hunting events to support the culture of learning (K3)							X		
4	Tools and Practices in DevOps - 200 minutes									
4.1	Select Tools Supporting DevOps Within the Organization									
QDO-4.1.1	Recall the capabilities of tools in DevOps (K1)								X	
QDO-4.1.2	Explain the tools supporting quality assurance and testing in DevOps (K2)								X	
4.2	Understand Technologies and Practices to Support Quality in DevOps									
QDO-4.2.1	Recall the non-testing activities within the CI/CD pipeline (K1)									X
QDO-4.2.2	Explain the different key release strategies in DevOps (K2)									X
QDO-4.2.3	Summarize the concepts of infrastructure as code (K2)									X
QDO-4.2.4	Recall feature toggles as a way to separate deployment from release (K1)									X

Business Outcomes: Quality in DevOps		QDO-BO1	QDO-BO2	QDO-BO3	QDO-BO4	QDO-BO5	QDO-BO6	QDO-BO7	QDO-BO8	QDO-BO9
QDO-4.2.5	Recall branching strategies (K1)									X
QDO-4.2.6	Recall chaos engineering (K1)									X
QDO-4.2.7	Recall telemetry and observability (K1)									X
QDO-4.2.8	Recall software bill of materials to support configuration management (K1)									X
QDO-4.2.9	Recall containerization (K1)									X

9 Appendix C – Release Notes

This syllabus is the first release of the International Software Testing Qualification Board Certified Tester Quality in DevOps (CT-QDO) syllabus.

10 References

Standards

Systems and software engineering (25010): Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model, 2023. 2023-01. Standard. International Organization for Standardization.

ISTQB® Documents

ISTQB® CT-AT: Agile Tester Syllabus, 2014. Version 1.1.

ISTQB® CT-ATLaS: Agile Test Leadership at Scale Syllabus, 2023. Version 2.0.

ISTQB® CT-ATT: Agile Technical Tester Syllabus, 2019. Version 1.1.

ISTQB® CT-MAT: Mobile Application Testing Syllabus, 2019. Version 2019.

ISTQB® CT-PT: Performance Testing Syllabus, 2018. Version 2018.

ISTQB® CT-SEC: Security Testing Syllabus, 2016. Version 1.0.

ISTQB® CT-TAS: Test Automation Strategy Syllabus, 2024. Version 1.0.

ISTQB® CTAL-TAE: Advanced Level Test Automation Engineer Syllabus, 2024. Version 2.0.

ISTQB® CTAL-TM: Advanced Level Test Management Syllabus, 2024. Version 3.0.

ISTQB® CTFL: Foundation Level Syllabus, 2023. Version 4.0.

ISTQB® Exam Structures and Rules, 2025. Version 1.2.

ISTQB® Generic Accreditation Guidelines, 2024. Version 2.4.

Books

ADZIC, Gojko, 2011. *Specification by Example: How Successful Teams Deliver the Right Software*. Shelter Island, NY: Manning Publications. ISBN 9781935182553.

ANDERSON, L.W.; KRATHWOHL, D.R., 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman. ISBN 9780801319037.

AXELOS, 2019. *ITIL Foundation: ITIL 4 Edition*. London, UK: TSO (The Stationery Office). ISBN: 978-0113316076.

BECK, Kent, 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall Richard, 2016. *Site Reliability Engineering: How Google Runs Production Systems*. 1st. O'Reilly Media, Inc. ISBN 149192912X.

CLOKIE, K., 2017. *A Practical Guide to Testing in DevOps*. Leanpub. Available also from: https://leanpub.com/testingindevops.

CRISPIN, L.; GREGORY, J., 2008. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Pearson Education. Addison-Wesley Signature Series (Cohn). ISBN 9780321616937. Available also from: https://books.google.hu/books?id=68_IhPvoKS8C.

HENDRICKSON, E., 2013. *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*. Pragmatic Bookshelf. The Pragmatic Bookshelf. ISBN 9781937785024. Available also from: https://books.google.nl/books?id=jxqpMQEACAAJ.

HUMBLE, J.; FARLEY, D., 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education. Addison-Wesley Signature Series (Fowler). ISBN 9780321670229. Available also from: https://books.google.hu/books?id=6ADDuzere-YC.

HUMBLE, J.; MOLESKY, J.; O'REILLY, B., 2020. *Lean Enterprise*. O'Reilly Media. ISBN 9781492092223. Available also from: https://books.google.hu/books?id=BmbyDwAAQBAJ.

KIM, G.; BEHR, K.; SPAFFORD, G., 2018. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. IT Revolution Press. ISBN 9781942788300. Available also from: https://books.google.hu/books?id=H6x-DwAAQBAJ.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J., 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press. ITpro collection. ISBN 9781942788072. Available also from: https://books.google.hu/books?id=ui8hDgAAQBAJ.

LARMAN, Craig; VODDE, Bas, 2016. *Large-scale scrum: More with LeSS*. Addison-Wesley Professional.

MARSELIS, R.; GEURTS, D.; RUIGROK, W.; VEENENDAAL, B. van, 2020. *Quality for DevOps teams*. Sogeti Nederland B.V. ISBN 9789075414905. Available also from: https://books.google.hu/books?id=ICHXDwAAQBAJ.

RUIGROK, W.; EGELMEERS, J.; MARSELIS, R., 2025. *Amplified Quality Engineering*. Sogeti Nederland B.V. ISBN 9789075414929.

Articles

DEBOIS, Patrick, 2011. Devops: A software revolution in the making. *Cutter IT Journal*. Vol. 24, no. 8, pp. 3–5.

Web Pages

ALLIANCE, Agile, 2026. *Agile Glossary and Terminology* [online]. Agile Alliance, 2026-01-31 [visited on 2026-01-31]. Available from: https://agilealliance.org/agile101/agile-glossary/.

AMAZON, 2024. *What is DevOps?* [online]. Amazon, 2024-10-15 [visited on 2024-10-15]. Available from: https://aws.amazon.com/devops/what-is-devops/.

DASA, 2019. *Embracing digital disruption by adopting DevOps practices* [online]. 2019-01-01. [visited on 2024-07-09]. Available from: https://www.dasa.org/wp-content/uploads/DASA-White-Paper-1-Embracing-Digital-Disruption-1.pdf.

DASA, 2024. *Challenges in Breaking The Wall of Confusion: DASA DevOps Fundamentals, Knowledge Bytes* [online]. DASA, 2024-07-05 [visited on 2024-08-09]. Available from: https://www.dasa.org/blog/challenges-in-breaking-the-wall-of-confusion/.

DIGITAL.AI, 2023. *DevSecOps Tools Periodic Table* [online]. [visited on 2024-11-13]. Available from: https://digital.ai/learn/devsecops-periodic-table/.

DORA, 2022. *2022 Accelerate State of DevOps Report* [online]. DORA, 2022-12-01 [visited on 2024-08-12]. Available from: https://dora.dev/research/2022/dora-report/2022-dora-accelerate-state-of-devops-report.pdf.

DORA, 2024. *DORA's software delivery metrics: the four keys* [online]. DORA, 2024-05-30 [visited on 2024-08-12]. Available from: https://dora.dev/guides/dora-metrics-four-keys/.

FELLOWS, Matt, 2022. *Pact - Getting started* [online]. 2022-08-30. [visited on 2025-06-24]. Available from: https://docs.pact.io/.

FOWLER, Martin, 2017. *Feature Toggles* [online]. 2017-10-09. [visited on 2017-10-09]. Available from: https://martinfowler.com/articles/feature-toggles.html.

ISTQB, 2025. *Glossary* [online]. ISTQB, 2025-05-15 [visited on 2025-05-15]. Available from: https://glossary.istqb.org/.

JANET GREGORY, Lisa Crispin, 2023. *Holistic Testing* [online]. Gregory/Crispin [visited on 2024-09-05]. Available from: https://agiletestingfellow.com/.

KUNDU, Sourav, 2019. *Idempotency in Infrastructure as Code* [online]. [visited on 2025-01-24]. Available from: https://skundunotes.com/2019/04/19/idempotency-in-infrastructure-as-code/.

MATTHEW SKELTON, Manuel Pais, 2023. *What Team Structure is Right for DevOps to Flourish?* [online]. Skelton/Pais, 2023-03-21 [visited on 2024-09-03]. Available from: https://web.devopstopologies.com/.

MOUSTAKIS, Ioannis, 2024. *Idempotency in Infrastructure as Code* [online]. 2024-10-31. [visited on 2025-01-26]. Available from: https://spacelift.io/blog/what-is-configuration-drift.

NETFLIX, 2025. *Chaos Engineering* [online]. netflixtechblog, 2025-01-05 [visited on 2025-01-05]. Available from: https://netflixtechblog.com/tagged/chaos-engineering.

RANGANATHAN, Rahul, 2023. *Building Resilient Systems with Chaos Engineering* [online]. Medium, 2023-07-27 [visited on 2025-01-05]. Available from: https://medium.com/google-cloud/building-resilient-systems-with-chaos-engineering-91f5b6adc972.

Scaled Agile Framework: CALMR, 2023a [online]. © Scaled Agile, Inc., 2023-03-14 [visited on 2024-08-12]. Available from: https://scaledagileframework.com/calmr/.

Scaled Agile Framework: Customer Centricity, 2023b [online]. © Scaled Agile, Inc., 2023-03-14 [visited on 2024-10-31]. Available from: https://scaledagileframework.com/customer-centricity/.

Scaled Agile Framework: Design Thinking, 2023c [online]. © Scaled Agile, Inc., 2023-03-14 [visited on 2024-10-31]. Available from: https://scaledagileframework.com/design-thinking/.

TORRES, Teresa, 2024. *Blog: Continuous Discovery Framework* [online]. https://userpilot.com/, 2024-09-16 [visited on 2024-10-31]. Available from: https://userpilot.com/blog/continuous-discovery-framework-teresa-torres/#What-is-continuous-discovery?.

VEENENDAAL, Erik van, 2025. *TMMi® in the DevOps world* [online]. [visited on 2025]. Available from: https://www.tmmi.org/download/tmmi-in-devops/.

The previous references point to information available on the Internet and elsewhere. Even though those references were checked at the time of publication of this syllabus, the ISTQB® cannot be held responsible if the references are unavailable anymore.

11 Further Reading

ARIOLA, W.; DUNLOP, C., 2014. *Continuous Testing*. CreateSpace Independent Publishing Platform. ISBN 9781494859756. Available also from: https://books.google.hu/books?id=MAtcngEACAAJ.

BELMONT, J.M., 2018. *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI*. Packt Publishing. ISBN 9781789133073. Available also from: https://books.google.hu/books?id=hh9sDwAAQBAJ.

BLANK-EDELMAN, D.N., 2018. *Seeking SRE: Conversations About Running Production Systems at Scale*. O'Reilly Media. ISBN 9781491978818. Available also from: https://books.google.hu/books?id=tmhqDwAAQBAJ.

BRYKCZYNSKI, Bill, 1992. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes*. Vol. 24, no. 1, pp. 82–89.

DUNLOP, C.; PLATZ, W., 2019. *Enterprise Continuous Testing: Transforming Testing for Agile and DevOps*. Independently Published. ISBN 9781699022948. Available also from: https://books.google.hu/books?id=ms1ZywEACAAJ.

DUVALL, P.M.; MATYAS, S.; GLOVER, A., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education. Addison-Wesley Signature Series. ISBN 9780321630148. Available also from: https://books.google.hu/books?id=PV9qfEdv9L0C.

FORSGREN, N.; HUMBLE, J.; KIM, G., 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press. ISBN 9781942788355. Available also from: https://books.google.hu/books?id=Kax-DwAAQBAJ.

GREGORY, J.; CRISPIN, L., 2019. *Agile Testing Condensed: A Brief Introduction*. Leanpub. ISBN 9781999220518. Available also from: https://books.google.hu/books?id=yP-yzwEACAAJ.

KIM, G., 2019. *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. IT Revolution Press. ISBN 9781942788775. Available also from: https://books.google.hu/books?id=kNSSDwAAQBAJ.

KINSBRUNER, E., 2018. *Continuous Testing for DevOps Professionals: A Practical Guide From Industry Experts*. CreateSpace Independent Publishing Platform. ISBN 9781727132175. Available also from: https://books.google.fi/books?id=_l_ruwEACAAJ.

MURPHY, N.R.; BEYER, B.; JONES, C.; PETOFF, J., 2016. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media. ISBN 9781491951170. Available also from: https://books.google.hu/books?id=tYrPCwAAQBAJ.

NOVOTNÝ, Vít, 2017. Using Markdown inside TEX documents. *TUGboat*. Vol. 38, no. 2, pp. 214–217.

ROSSEL, S., 2017. *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing. ISBN 9781787284180. Available also from: https://books.google.hu/books?id=1xhKDwAAQBAJ.

SCHEAFFER, J.; RAVICHANDRAN, A.; MARTINS, A., 2018. *The Kitty Hawk Venture: A Novel About Continuous Testing in DevOps to Support Continuous Delivery and Business Success*. Apress. ISBN 9781484236611. Available also from: https://books.google.hu/books?id=asRIDwAAQBAJ.

VADAPALLI, S., 2018. *DevOps: Continuous Delivery, Integration, and Deployment with DevOps: Dive into the core DevOps strategies*. Packt Publishing. ISBN 9781789131253. Available also from: https://books.google.hu/books?id=N5RRDwAAQBAJ.

VEENENDAAL, Erik P.W.M. van; WELLS, B., 2012. *Test Maturity Model Integration TMMi: Guidelines for Test Process Improvement*. Tutein Nolthenius. ISBN 9789490986100. Available also from: https://books.google.fi/books?id=6y8QnQEACAAJ.

ZHAN, C.; ZHAN, Z., 2021. *Practical Continuous Testing: Make Agile/DevOps Real*. CreateSpace Independent Publishing Platform. ISBN 9781507742112. Available also from: https://books.google.hu/books?id=6QG1zgEACAAJ.