SYLLABUS

Version 2020

AT*SQA
ASSOCIATION FOR TESTING & SOFTWARE QUALITY ASSURANCE
Global Certification Body for ISTQB and ASTQB

Table of Contents

0. Introduction to this Syllabus

0.1. Purpose of this Document

This syllabus forms the basis for the AT*SQA certification for DevOps. AT*SQA is an International Standards Organization (ISO) compliant certification body for software testers. AT*SQA provides this syllabus as follows:

1. To training providers, to produce courseware and determine appropriate teaching methods.
2. To certification candidates, to prepare for the exam (as part of a training course or independently).
3. 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.

AT*SQA may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

Please note that specific tool references are provided to help the reader understand the technology and to provide examples of tools that are in common usage. This is not an endorsement for these tools by AT*SQA.

1. Introduction to DevOps – 90 mins.

Keywords

Agile, lean, Theory of Constraints, Toyota Kata, value stream

Learning Objectives for Introduction to DevOps

1.1 Overview

LO-1.1.a (K1) Recall who can be in the “Dev” and “Ops” roles in a DevOps project

1.2 Origins

LO-1.2.a (K2) Explain the different principles and methodologies that formed the basis of DevOps

LO-1.2.b (K2) Explain the relationship between Agile and DevOps

1.3 Myths

LO-1.3.a (K2) Explain why the myths around DevOps are not valid

1.4 Benefits

LO-1.4.a (K1) Recall the benefits of a DevOps implementation

1.5 Transitioning to DevOps – People, Processes and Tools

none

1.1. Overview

DevOps first came to people’s attention in 2008 when Patrick Debois [Debois11] invited like-minded people to a conference to discuss how the delivery of product as well as infrastructure could use the concept of the Agile software development lifecycle. DevOps stands for “Development” and “Operations.” It refers to people (primarily the development team and the operations team) working together to build, deliver, and run robust software. Under DevOps, the software development process should not only focus on writing code but, at the same time, should also consider other aspects of development such as testing, deployment, and delivery. The successful delivery of high-quality software should be used to measure the success of software development, which is frequently not the case with most development models.

The reason for the popularity and movement towards DevOps may be better understood with an example. Let's consider a software company with a traditional development approach. A software development process could be as follows:

- The development team writes code to implement new features and/or fix bugs. The new code featuring the changes resides on the development environment and may or may not be tested by the development team.
- The code is then deployed in the test environment (which should mimic the production environment, but in reality it often does not) where it is verified by the QA team.
- The QA team might find issues; developers fix those issues; QA tests again and this goes back and forth several times.
- The new code is then provided to the operations team for deploying into the production environment. The operations team is now responsible for the release, delivery and sometimes even management and maintenance of the code.

There are many issues with this approach. The activities involved from creating the software to finally deploying it in the production environment often last weeks or months. Priorities of the Development team, QA team and IT Operations teams may not be aligned and may actually be in conflict. The Development and QA teams are focused on new development and testing of the release, while the Operations team cares about the stability of the production environment. The Development and Operations teams may not be aware of each other's work and work culture. The teams work in different types of environments, leading to the possibility that the development environment does not reflect the production environment. In that case, a product that works successfully in development may fail when deployed to production. While the Operations team is busy with deployment activities, the Development team may get a request for a new feature or a defect fix, or may begin developing a new product. In this case, if the Operations team needs help with deployment, the Development team may not be available to assist. This gap in communication and necessary collaboration can lead to product failure in production and issues with quality throughout the organization. It is common to see these types of collaboration gaps. Poor collaboration can cause many issues in the application deployment to different environments. Issues are often resolved informally, via email, chat, calls, meetings, and so on, resulting in quick fixes with insufficient documentation and a high likelihood of the same problems recurring with the next deployment.

The classic organizational structure tends to put strict borders between the Development team, QA team and Operations team. In this model, no matter how efficient a process is within a single department, it may not be the same for the next one. Furthermore, if there are walls between departments, problems will be reflected in the quality and maintainability of the product. DevOps breaks down the silos and barriers so that the Development (including QA) and the Operations teams have a more efficient way to work together across the entire software lifecycle, from development to delivery. DevOps requires collaboration between the different disciplines of software development.

DevOps is a cultural shift focused on building and operating at high velocity. It is a development approach that involves activities that are 'continuous': *continuous development, continuous integration, continuous delivery, continuous testing, continuous deployment,* and *continuous monitoring.*

Development roles in DevOps are not limited to people titled as developers or software engineers. The roles include everyone who is involved in making the software before it reaches production, which includes product managers, business analysts, product owners, and quality assurance testers. Similarly, operations roles include everyone who is involved in managing production, including deployment, execution and maintenance. This includes system operators, systems engineers, DBAs, network engineers, and security experts. DevOps requires people in these roles to work together from the beginning of a software development project until it is delivered, breaking down the walls that stand between different departments.

1.2. Origins

DevOps represents a convergence and application of different principles, from manufacturing to leadership, in the IT industry to streamline the release of IT products. The roots of DevOps can be found in Lean, Toyota's Kata, the Theory of Constraints, Kaizen, as well as in the evolution of Agile. Each of these is discussed briefly below.

1.2.1. Lean

The idea of Lean originated from lean manufacturing in the Toyota Production System in the 1990s. Lean manufacturing focuses on parts of the system that add value, by eliminating waste everywhere else. Lean software development applies similar concepts to the software industry. Waste that can be eliminated in these areas include unnecessary software features, communication delays between stakeholders, slow application response times, inefficient defect management, and overbearing bureaucratic processes.

Although software development is different than manufacturing, Lean software development has proven to be useful to optimize and enhance value streams. A value stream is a set of steps from the start of value creation until the end result is delivered to the customer. Other features in Lean software development focus on eliminating waste, building in quality, creating knowledge, deferring commitment, delivering quickly, respecting people, and optimization as a whole.

DevOps shares many of the Lean goals and integrates many of the Lean principles by focusing on improving work culture, fostering collaboration between developers and operations, optimizing workflow, assuring quality at the source, and respecting every individual.

1.2.2. Toyota Kata

A kata is a small, well-structured protocol or routine that becomes second nature through practice. In 2009, Mike Rother wrote *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results*, which framed his twenty-year journey to understand and codify the Toyota Production System. The point of the kata is to acquire habits through practice, rather than to just memorize the routine.
This is often known as the Improvement Kata and is a process to promote continuous improvement. It is a structured way to create a culture of continuous learning and improvement at all levels, through daily habits or routines.

The steps to improvement are as follows:

1) Understand the vision and direction for the project
2) Analyze current conditions
3) Establish target conditions
4) Plan → Do → Check → Act (PDCA) towards the target

Daily habits/routines help people strive towards the goal. When leveraging the Improvement Kata, the team is required to set the direction. It provides the opportunity for the team to thoroughly understand what target they are trying to achieve and to solve problems and experiment toward achieving that target in a methodical way.

1.2.3. Theory of Constraints

The concept of "Theory of Constraints" was first introduced in 1984 in Dr. Eliyahu M. Goldratt's book *The Goal* and further detailed in his sequel, *The Theory of Constraints*. It is a methodology for identifying and systematically taking actions to improve and remove the most limiting factors (i.e., the bottlenecks) that stands in the way of achieving goals and success. In DevOps, the "Theory of Constraints" has become a reputed model for eliminating waste, recognizing opportunities for better collaboration, and streamlining the flow of work down the value stream.

1.2.4. Kaizen

Kaizen, meaning “improvement”, was initially practiced in Japan after World War II to eliminate waste in manufacturing processes. The goal behind Kaizen is to continuously improve in small increments and to engage all stakeholders in the improvement process. As with other practices born from manufacturing, it seeks to improve the efficiency of all the activities that contribute to the final product. When applied to software development, it is similar to the other approaches that are designed to drive efficiency and create an inclusive environment where stakeholders are invited to engage and participate in the improvements.

1.2.5. Agile

In 2001, the Agile methodology was first identified via the “Agile Manifesto”, although such practices were already being applied in eXtreme Programming (XP) and other approaches. The goal of the Agile methodology was to move away from the rigid,

waterfall-oriented, documentation-heavy world of software development, which resulted in software development projects being delivered late, over-budget, and failing to meet stakeholder expectations. The goal was to move to an iterative approach, where there was constant interaction with the customer or end-users (called product owners) [Gregory19].

The “Agile Manifesto” outlines software development practices to fit within the following values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The Agile methodology is primarily concerned with the creation of the software, not the actual delivery. Once a product is developed using Agile, it is often handed off to the Operations team for successful delivery.

Agile has become one of the core drivers for DevOps. As code was developed in smaller increments, the need for faster, easier and more reliable testing and delivery to the various environments, including production, became important. Traditional Operations teams and processes where not able to adapt to these needs, which resulted in major bottlenecks being created at the dev-to-test and test-to-production handoffs. Environment controls and constraints became significant bottlenecks in the smooth flow of the software from point to point in its journey from development to deployment.

DevOps bridges this gap and removes the bottlenecks between Development and Operations to ensure successful delivery. It does so by bringing the Operations and Development teams together during the entire software lifecycle, from development to delivery.

1.3. Myths

There are several myths around DevOps. It is a good idea to be clear about what DevOps is not. DevOps is not about:

- Removing or replacing Ops: DevOps does not mean that Ops is going away or that developers are taking over Ops responsibilities. Development and Ops both have experiences and knowledge to contribute. Neither role is being eliminated.
- A bunch of fancy tools: DevOps is not about introducing a lot of tools without any processes in place. Unwise use of tools without proper process and people can do as much damage as the wise use of tools can bring benefit.

Even though DevOps leverages automation and tools, it is an organizational change that is supported by those tools, not the tools themselves.

- A job title: DevOps is a not a job title
- Replacing Agile: DevOps principles and practices are compatible with an Agile methodology, and, in many cases, DevOps is a logical extension of Agile. DevOps does not replace Agile, in fact it encourages an organization to embrace Agile before moving to DevOps.
- A separate team: DevOps is a unification of teams to reach a common goal efficiently
- Doing all the work with half the people: DevOps does not replace people, it just allows people to work more efficiently and collaboratively
- One (right) way to do DevOps: There are many ways to implement DevOps and the ideal implementation depends on the organization's own goals and needs.

1.4 Benefits

Proper implementation of DevOps has been shown to return significant benefits to an organization. These include the following:

- The ability to assess quality at every stage of development and deployment, resulting in a high-quality product with fewer defects
- Collaboration and management throughout the application lifecycle
- The ability to release small increments of functionality more often
- Capturing feedback early and frequently to improve both the product and the process
- On time delivery and lower cost of delivery thanks to a higher quality product and more efficient delivery process
- Reduction in vendor and third-party issues by increasing collaboration and transparency
- Faster time to market by employing efficient processes and effective communication

1.5. Transitioning to DevOps – People, Processes and Tools

Successfully transitioning to DevOps requires having the right people with the right responsibilities and training, efficient and well-understood processes, and tools to assist with the automated processes. The processes needed for DevOps are discussed in Chapter 2. Chapter 3 describes testing in DevOps and how it is

implemented throughout the software lifecycle. Chapter 4 covers how tools are integrated into the processes. Chapter 5 discusses how people fit into the DevOps model as well as some management aspects, challenges and cultural changes.

2. Fundamentals – 200 mins.

Keywords

binary files, CAMS, class files, continuous build integration, continuous delivery, continuous deployment, continuous lifecycle, continuous feedback, continuous improvement, continuous innovation, continuous monitoring, continuous testing, DevOps pipeline, microservice, package files, source code, three-way thinking, virtualization

Learning Objectives for Fundamentals

2.1 Core Values

LO-2.1.a (K1) Recall the four core values of DevOps
LO-2.1.b (K2) Explain how automation relates to DevOps

2.2 Three-Way Thinking

LO-2.2.a (K2) Explain how Three-Way Thinking should be employed in the DevOps process

2.3 The DevOps Continuous Lifecycle

LO-2.3.a (K1) Recall the elements of the continuous lifecycle
LO-2.3.b (K2) Explain the expected benefits of the phases of the continuous lifecycle
LO-2.3.c (K2) Explain how the elements in the continuous lifecycle should be applied to a given scenario

2.4 The DevOps Pipeline

LO-2.4.a (K1) Recall the stages of the DevOps pipeline

2.5 Risk-Based Testing

LO-2.5.a (K1) Recall how risk-based testing can help determine what to test

2.6 DevOps Technologies

LO-2.6.a (K2) Explain how the DevOps technologies can be applied to a given scenario
LO-2.6.b (K2) Understand how virtualization can be applied to facilitate test and deployment automation

2.1. Core Values

The acronym CAMS is often used to describe the four core values of DevOps: Culture, Automation, Measurement, and Sharing.

- **Culture (C):** Culture may present a challenge to DevOps implementation. Wanting to do DevOps by investing in DevOps tools, training staff, and hiring expert consultants, without a proper introduction to DevOps mindset, behaviors, and incentives, may result in failure. In order for DevOps to succeed, an organization's culture must transform into a more collaborative, process-defined, and people-centric environment to drive continuous improvement.

  Some of the underlying inspirations of DevOps, as discussed in Chapter 1, come from Agile and Lean. The C in CAMS points to Agile's core value of 'individuals and interactions over process and tools,' and Lean's streamlining operations and reducing waste. The end goal is to maximize customer value.

  In DevOps, a culture of continuous collaboration, communication, and fostering respect for each other is needed for it to be successful.

- **Automation (A):** DevOps is about speed, flexibility, resiliency and continuous improvement. Automation is a critical part of DevOps. Once the culture is understood, automation is used to accelerate and execute the process smoothly.

  The 'continuous' activities in DevOps (continuous development, continuous integration, continuous testing, continuous deployment, continuous monitoring, continuous feedback and continuous delivery) can be achieved by automation. It is a practical aspect of DevOps and is required for the successful implementation. Automation applies to server builds, database software installations and configurations, network settings, storage allocations, schema builds, code compiles, testing, job scheduling, and more.

- **Measurement (M):** Measurements are important for understanding if the changes being made are useful and beneficial. It is crucial in DevOps to identify the correct measurements and use them to improve the process. For DevOps to succeed, inefficiencies and avoidable steps must be consistently removed. Lean and Kanban techniques can be applied to trim inefficiencies and reduce waste.

  Measurements should be taken during the development and delivery process to see how long each step takes, and to evaluate the need for each step. Based on the measurements, decisions can then be made to remove impediments, smooth the workflow, maximize efficiencies, and drop unnecessary steps.

- **Sharing (S):** A core element of DevOps is sharing information. The S in CAMS encourages DevOps teams in sharing, openness and transparency. It asks for discrete improvement. Business, development, and operations teams must communicate with each other to achieve this. Messages need to be sent and received simultaneously upstream and downstream.

  Each team member needs to understand the business needs, address operational challenges early, and communicate via a feedback loop about what needs to be done to implement better-performing, resilient systems.

2.2. Three-Way Thinking

One of the key concepts of DevOps is 'Three-Way Thinking' [Kim12]. The Three Ways describe the importance of looking at the performance of the system as a whole, enabling and amplifying feedback loops, and focusing on creating a culture of continual experimentation and learning. DevOps uses Three Ways as a set of principles that can be used to implement the practices of DevOps.

2.2.1 The First Way: Systems Thinking

The First Way is systems thinking. Instead of thinking about only one piece of the project, such as development or testing, the First Way encourages thinking in terms of the whole system and includes building in quality by preventing defects from being passed downstream. The First Way enables a left-to-right flow of work from business, through development, to operations, and finally to the user, where the value is actually created. The goal is to speed up the process of flow from left-to-right. This is not achieved by reducing work or personnel (e.g., not doing proper testing, not having QA). A better way would be to have the proper work and roles but have smaller pieces of work flow smoothly from left-to-right. This will result in a faster flow, quality control at each small step, faster feedback, and will ultimately help deliver value faster and with less cost.

The First Way also focuses on removing constraints. A left-to-right flow will almost always have constraints associated with it and bottlenecks will exist.. Instead of jumping over them or putting them aside to take care of later, the First Way considers removing those constraints that are on the path and keeping the continuous left-to-right flow. Practices to accommodate the First Way include continuous build, integration, test, and deployment processes, creating environments on demand, and building maintainable systems.

2.2.2. The Second Way: Constant Feedback Loop

While the First Way emphasizes the flow of work from left-to-right, the Second Way focuses on the processes that enable rapid and constant feedback from right-to-left at all stages of developing the system. It requires continuous feedback to enable faster detection of and recovery from problems and to prevent those problems from happening again. An important point in the Second Way is that the flow from left-to-right should stop once an error is found and the feedback from that error should be

relayed as fast as possible to the source, so the error can be fixed and subsequently prevented.

The Second Way also helps with process improvement. Short, continuous feedback loops identify problems as they occur and allow the team to analyze each problem for further action. This facilitates problem resolution and helps to prevent the same or similar mistakes being repeated. This maximizes the opportunities for an organization to learn and improve. A good way to achieve the short, continuous feedback loops for the Second Way is to have a continuous integration, build and deployment process working together with a fast, automated suite of tests. These two things together will provide quick feedback to developers and allow defects to be detected and eliminated at the point of introduction, rather than further downstream in the process.

2.2.3. The Third Way: Continual Experimentation and Learning

While the First Way addresses workflow from left-to-right and the Second Way addresses the reciprocal fast and constant feedback from right-to-left, the Third Way focuses on creating a culture of continual learning and experimentation. The Third Way enables the creation of a high-trust culture that supports a dynamic, disciplined, scientific approach to experimentation and risk-taking. This creates a learning culture that benefits from both successes and failures. No matter which team someone is working on, Dev or Ops, the Third Way values the cumulative and collective experience of everyone. The outcomes of the Third Way include allocating time for the improvement of daily work, creating rituals that reward the team for taking risks, and purposely introducing faults into the system to increase resilience.

2.3 The DevOps Continuous Lifecycle

For a successful DevOps culture, both the Dev and Ops teams should follow a ‘continuous’ lifecycle in an effective manner. The DevOps continuous lifecycle includes the following:

- Continuous integration
- Continuous delivery
- Continuous testing
- Continuous deployment
- Continuous monitoring
- Continuous feedback
- Continuous improvement
- Continuous innovation

These lifecycle components are important for accomplishing the goal of making each software delivery on-time and successful.

2.3.1. Continuous Integration (CI)

Continuous integration is the practice of merging all developers' working code to a main branch of the codebase in a shared repository at least several times a day. The developers' changes (commits) are validated by creating an automated build and running automated tests against the build. An automated build process includes the following activities:

- Compiling source code into class files or binary files
- Providing references to third-party library files
- Providing the path for configuration files
- Packaging class files or binary files into package files
- Labelling the source code with the build identifier
- Executing automated test cases
- Deploying package files on local or remote machines
- Reducing manual effort in creating the package file

Continuous integration ensures that the application does not break whenever new commits are integrated into the main branch of the codebase. Each check-in made by a developer is verified by either a pull mechanism, where an automated build is executed at a scheduled time; or a push mechanism, where an automated build is executed when changes are saved in the repository. Automated unit tests are executed against the latest changes available in the source code repository.

When changing and updating code, a developer takes a copy of the current codebase from the central repository. This copy is then used to make changes. When the changes are completed, the code is checked back into the repository to be incorporated into the next build. This becomes complicated when other developers are also making and committing changes to the repository. If one change should break another change, or break the base code, it needs to be found and fixed as quickly as possible to minimize the impact of the problem. With smaller increments of changes, troubleshooting to find the offending change is easier. As code is checked out and changed, the local copy the developer has obtained becomes out of sync with the repository. It is very important to maintain the code repository with updated code and libraries from the whole team.

The goals of continuous integration are to maintain the code repository, prevent a backlog of changes that have not been incorporated into the repository and, by frequent testing, remove any surprises that might occur when the code is deployed into an environment. The CI process includes the following:

- Commit changes as they are made
- Build each committed change with an automated process
- Test the build with automated tests

- If anything breaks, fix it immediately while everyone remembers what was changed

Most of these continuous integration tasks can be automated; however, it is a process which developers must adopt for CI to work as intended. There are many continuous integration tools available to help in this process. There are tools that work locally (e.g., Jenkins) and there are cloud-based options too (e.g., Travis CI and Circle CI). All of them work on the same principles. The CI system watches a code repository (e.g., Git, SVN). When a new commit to the repository occurs, the CI systems takes the new commit and starts a build process which builds components such as compiled binaries or data files. Once the build is successfully completed, the CI system then spawns tests that can be run on the build artifacts. The results are reported to the developers.

The following are good practices needed to establish a successful CI process:

- A single source code repository should be maintained at all times
- The build should be automated
- Code should accompany unit tests and the build should be self-testing
- Code should be committed to the build as soon as it is ready for integration
- Every commit should be built to verify the integrity of changes
- Users are authenticated and access to the build server is controlled

2.3.2 Continuous Delivery

Continuous Delivery is a process to deploy software into the appropriate environment in an automated fashion and receive continuous feedback to improve the quality of the process and the software. It builds upon an existing CI process where teams produce software in short cycles, ensuring that software can be reliably released at any time. Ideally, at the end of every CI build, the software is delivered to the test environment for the Quality Assurance (QA) team to test, and then to the operations team for delivery to the production environment. In reality, not all builds from CI are ready to go to QA and, similarly, software that comes out of QA may not be ready to go to production. This is often handled by using multiple levels of test environments (e.g., integration, user acceptance testing (UAT), stage). Only the builds that are truly "ready" (QA ready, production ready) will be allowed to go forward to the users. The goal of Continuous Delivery is to deliver new features that developers are creating to the customers and users as soon as possible.

Continuous Delivery requires the creation of a delivery pipeline, often referred to as the DevOps pipeline. To deliver feature-complete software from one environment to another (e.g., development to QA, QA to production) often requires multiple iterations. In order for this to work successfully, orchestration of the deployments of code, content, applications, middleware, and environment configurations is required.

A strong foundation of CI and a good automated test suite with good coverage of the codebase is needed for Continuous Delivery to succeed. The trigger to start Continuous Delivery is manual, but once the delivery process begins, all the activities in the delivery process should be automated.

The following are good practices to establish for a successful Continuous Delivery process:

- Clone all environments as much as possible to mimic the production environment
- Automate deployment

2.3.3. Continuous Testing

Continuous Testing is the execution of tests repeatedly against a codebase. Testing provides the quality gates throughout the DevOps pipeline and increases confidence in the product. The success of Continuous Delivery or Deployment relies on Continuous Testing. The quality of the tests and the testing process determines the quality of the released software. At every stage of the DevOps pipeline, Continuous Testing ensures that what is being delivered through a stage is stable and good enough to progress to the next stage of the pipeline.

For Continuous Testing to succeed, the QA and Operations teams should be integrated with the Development team. Testing should be incorporated as much as possible into the DevOps pipeline. For example, performance and security testing should not be an independent activity but should be incorporated into the pipeline at the earliest possible point where testing can be effective. To facilitate early testing, mocks and stubs are commonly used to replace components that are not available and to perform tests that might be damaging.

It is important to remember that some types of testing, such as system-level integration testing, may be best performed on the whole product, or at least a large subset. This type of testing still belongs in the pipeline, but may come at the end rather than following after each build.

The following are good practices to establish for a successful Continuous Testing process:

- Understand and introduce shift-left testing
- Establish effective test automation throughout the lifecycle
- Use effective testing tools to manage the continuous testing lifecycle from test case creation to defect reporting, fixing and retest
- Continuously review, update, and refactor the existing test cases to increase coverage and efficiency

2.3.4. Continuous Deployment (CD)

Continuous Deployment (CD) takes the Continuous Delivery process one step further. A change that passes all stages of the DevOps pipeline is released to the user. The trigger to deployment is automatic; therefore the whole release process is automated. There is no set release day; all stakeholders can see the changes taking place as soon as the changes are committed and have passed the tests. A failed test prevents a change from going to the production environment.

In Continuous Delivery, an application is automatically staged in preparation for production, but it must be manually deployed to production. Before that, however, the application must be manually reviewed and verified. Continuous Deployment adds automation to the process, making it (theoretically) possible for changes to go live within minutes after the code has been checked into the repository and verified by the automated processes.
One downside of continuous deployment is that it eliminates the safeguards that come with manual verifications. For this reason, application delivery teams that implement Continuous Deployment must adhere to best coding and testing practices to ensure the application is production-ready. Another potential downside occurs when the user(s) is not ready to receive the new release, perhaps because training on a new feature may be required.

The following are good practices to establish for a successful Continuous Deployment process:

- Limit or eliminate human interaction, requiring a well-established pipeline to do this automatically
- Conduct rigorous testing throughout the pipeline to provide enough confidence to deploy features to production automatically
- Require each member of the team to accept ownership and responsibility for the artifacts being deployed to the customer (code, feature, tests)
- Have a well-established and robust process for rolling back to a previous release in the event it becomes necessary

2.3.5. Continuous Monitoring

Continuous Monitoring allows the DevOps team to monitor the application constantly in the production environment to ensure that the software is performing at an optimal level and the environment is stable. Continuous Monitoring helps in diagnosing and fixing errors as soon as they are found. It also helps in understanding the application's usage pattern and lets the DevOps team be proactive rather than reactive to any errors that may occur.

The following are good practices to establish for a successful Continuous Monitoring process:

- Identify what should be monitored and how frequently

- Ensure tools and processes are in place to make informed decisions based on data from monitoring

2.3.6. Continuous Feedback, Improvement and Innovation

An important part of DevOps is the focus on improvement. This means that feedback must be continuously gathered from all stakeholders affected by the process. This feedback is then analyzed by the team to determine areas for improvement and to identify opportunities for innovation. DevOps implementation can suffer from a lack of enthusiasm once the tools and processes are in place and running smoothly. It is important for the team to remember that improvements will always be needed as new and more efficient processes and tools become available all the time. Gathering feedback and using it to inspire innovation to achieve overall improvement is a critical aspect of a successful DevOps implementation.

The following are good practices to establish for successful Continuous Feedback, Improvement, and Innovation:

- Understand the collaboration between the different teams
- Implement a collaboration strategy
- Ensure collaborative project management tools are used
- Share continuous feedback between the different teams

2.4. The DevOps Pipeline

The flow of a software product from conception to deployment is sometimes referred to as the DevOps pipeline. This pipeline incorporates all the continuous activities described in Section 2.3, but is often shown in a simplified form with these primary stages:

DevOps Pipeline: Plan → Code → Build → Stage → Deploy
The planning stage in DevOps is a pre-CI/CD stage where requirements are gathered, tested, and polished. Once a set of well-tested requirements is agreed upon, developers start coding those requirements. As they code, developers commit their changes to a source control repository. Each developer's source code should have accompanying unit tests. The commit triggers a build. Once all the code compiles properly and all unit tests pass, the build is considered successful. If any test fails, the build is unsuccessful, and the commit will roll back to the previous successful commit. This is Source Control Management (SCM) and continuously cycles for all subsequent commits. The codebase is built up this way. This Continuous Integration process ensures that passing tests accompany every piece of code written by the developers. An SCM tool helps in this process as it stores the code in repositories, versions the code, and helps in team collaboration. Git is an example of a commonly used SCM tool.

The tested and verified build then moves to a staging environment that mimics the production environment. This is where integrated tests and other necessary tests such as functional, non-functional, performance, security, user acceptance, and exploratory tests, are executed. Some of these tests may be automated while others may rely on manual execution by testers or other stakeholders. Instead of going directly to the staging environment, software is often tested in a series of environments that are targeted for particular types of tests (e.g., security, performance, UAT) before going to the staging environment.

Once the application is thoroughly tested in the staging environment, it is ready to be deployed. Depending on how the DevOps pipeline is set up, it can be deployed to a pre-production environment where the operations team may run more tests, such as operational acceptance tests. It can also be deployed directly to the production environment.

2.5. Risk-Based Testing

Risk is a significant factor in determining the best test approach and the level of testing required. Risk level is generally determined by considering the impact of a possible failure and the likelihood of that failure occurring. This information is used to determine how much testing is required, how much documentation is needed and how the testing should be prioritized. This information may also help to understand which tests must be included in automation suites and how often they should be executed.

Higher risk software generally requires more formal approaches with more complete documentation. Lower risk software can adopt a lighter approach and may have little or no documentation required. Using risk prioritization, commonly called risk-based testing, on every project is a strong approach and helps to prioritize and define all testing activities.

It should be noted that there is a degree of error in assessing risk as it is essentially a qualitative exercise. Contingency plans are helpful when low risks may become high risks. This can happen when the impact and/or the likelihood of the risk occurring changes after the assessment occurred. A continuous evaluation of risk assessments can reduce the chance of such errors.

2.6. DevOps Technologies

All DevOps team members, whether they be a developer, tester or part of the operations team, should be familiar with some of the commonly used technologies that make DevOps successful.

2.6.1. Cloud Computing

One important feature of a reliable DevOps implementation is the ability to create environments. Cloud Computing environments are often used for development and testing because they are fast to create, easy to re-create and help to refine the

requirements for the production environment. Because of the speed required to provide a smooth flow from development to release, environment configuration cannot be a bottleneck. This means the environment creation, configuration and eventual destruction must be automated. Cloud services are ideal for this by providing an easily extended and contracted environment where new test and development environments can be created as needed.

Environment creation and control is one of the primary success factors in a DevOps implementation. Because testing may be occurring in multiple areas at the same time, such as functional testing new features while performance testing an integrated set of features, multiple dedicated environments are needed. In traditional approaches, time is lost waiting for environments to be configured for a specific purpose. By using cloud services, environments can be procured for a lower cost and the time required for use can be limited to what is actually required. Both the Operations team and the Development team need to be engaged to automate the environment creation and configuration activities. While this process immediately benefits development and testing, it also helps the eventual production deployment with documented and tested configurations.

See Section 4.3 for a discussion of Infrastructure as Code.

2.6.2. Microservices

The microservice architectural approach entails building one application as a set of independent services that communicate with each other, but are configured individually. The benefit of building an application this way is that it provides low coupling in the application design. This improves design, troubleshooting, maintenance and may also enable early testing. Because there is low coupling, an individual microservice can be changed with little effect on the rest of the system. Each microservice can be developed and released independently of other system components. With the high rate of deployment, microservices allow for keeping the whole system stable, while fixing problems in isolation.

2.6.3. Virtualization

Virtualization is important for DevOps because it provides identical environments for the Dev and Ops teams. The environments can be launched in a timely and efficient manner, removing a potential bottleneck in the DevOps pipeline. Virtualization enables continuous testing by providing anytime, anywhere access to a complete, simulated production-like test environment.

Before virtualization was available, a production or test environment existed on physical servers on-site. Organizations would purchase these servers and install operating systems on them. The product software would reside on top of the operating systems. Because these servers were often dedicated for a particular environment, it was not unusual for only a small portion of the server resources to be used at any given point in time. As environments multiplied (dev, test, UAT, stage, production), so did the servers and their associated maintenance costs and efforts (e.g., lab space,

power, air conditioning, system administrators). When changes to the environments were required, such as an operating system upgrade or a configuration change, significant downtime was incurred and scheduling became problematic.

Virtualization technology addresses many of these problems. With virtualization, the physical server still has its own operating system. On top of that operating system, a hypervisor layer is installed. The hypervisor, also known as a Virtual Machine Monitor (VMM) is computer software or hardware that creates and runs the virtual machines (VMs). Some popular hypervisor providers are VMware, VirtualBox, and Microsoft Hyper V.

The hypervisor allows the installation of multiple virtual machines on a single physical machine. Each virtual machine can have a different operating system and can run different applications. To the software, each virtual machine looks and acts just as a physical server would. There are significant cost savings and efficiency gains by running multiple virtual machines within one physical server. Memory and disk space on the physical machine are allocated among the virtual machines, allowing maximum utilization of the physical resources.

Virtualization in the Cloud

In recent years, with the invention of efficient cloud infrastructure, more and more companies are moving towards deploying VMs in the cloud with providers such as Amazon (AWS), Google (Google Could Provider, GCP), and Microsoft (Azure). This eliminates the need to purchase physical machines as servers. Having a virtualized environment in the cloud is cost-effective. Each physical machine is divided into multiple VMs and each one only uses its allocated CPU, memory and storage resources. The payment is on demand and a pay per use model. It is also easy to scale. If more memory or disk space is needed, it is simply a change to a configuration parameter rather than a physical installation. If more instances of the application are needed, additional VMs can be configured and launched. The ability to rapidly configure and to pay for only what is needed has significantly increased agility for many organizations.

The hypervisor-based virtualization model has some limitations. Each virtual machine created on top of the virtual layer still needs to have an operating system installed, with its own memory management, device drivers, daemons, and other resources. With hypervisor virtualization, portability of the application is not always efficient. The limitations associated with hypervisor virtualization can be resolved with more recent ways to virtualize, such as container-based virtualization.

Container-based Virtualization

Container-based virtualization wraps the software in a complete file system that contains everything it needs to run: code, system tools, system libraries – anything that can be installed on a server. Container-based virtualization requires a server(s) which could be either a physical machine or a virtual machine. The operating system is installed on the server. In the operating system, a container engine is installed, which allows the operating system to run multiple guest instances. Each guest

instance is called a container. Within each container, an application with all its libraries and dependencies can be installed. An example of a popular implementation of container-based virtualization is Docker containers, an open-source development platform.

One of the key differences between the hypervisor-based virtualization model and the container-based virtualization model is the replication of the kernels. In the hypervisor model, each application is running in its own copy of the kernel and the virtualization happens at the hardware level. In the container model, there is only one kernel which will supply different binaries and runtime to the applications running in isolated containers. Each container will share the base runtime kernel which is the container engine. In this model, virtualization happens at the operating system level. Containers share the host's OS making the virtualization more efficient and light-weight. The container-based virtualization model also allows creation and running of multiple environments, as well as running different versions of an application in each container, in an efficient manner without interfering with one another.

Container-based virtualization is more cost effective than hypervisor-based virtualization. Container-based virtualization does not create an entire virtual operating system. Instead, only the required components are packaged up inside the container with the application. Therefore, containers consume less CPU, RAM and storage space than VMs, allowing more containers to be running on one physical machine than VMs. Containers house the minimal requirements for running the application, which can spin up as fast as a process. A container can be several times faster to boot than a VM. Lastly, and a very important benefit, is the ease of portability. Because containers are essentially independent self-sufficient application bundles, they can be run across machines without compatibility issues. Applications that are running in Docker containers can run on a developer’s laptop as well as on an on-premise infrastructure. They can also run in the cloud or can be moved to a Platform as a Service (PaaS), such as OpenShift by Red Hat or Bluemix by IBM.

The risk with container-based virtualization is the potential corruption or crash of the kernel by one application. Because the kernel is shared, this type of corruption could affect other applications running on the same kernel.

Service Virtualization

Service virtualization is used when a service is needed by the software under test, but is not available for some reason. In this case, a virtualized service can be created which will act like the real service to any interfacing software. Various tools exist to create these virtual services (sometimes called “stubs on steroids”), including Tricentis by Tosca, SoapUI by SmartBear, IBM Green Hat, and LISA by CA.

Virtualized services are particularly helpful when testing API-driven applications, cloud-based applications and service-oriented architectures.

3. Testing - 245 mins

Keywords

API testing, application, behaviour-driven development, feature file, integration testing, test-driven development

Learning Objectives for Testing

3.1 DevOps and Testing

LO-3.1.a (K1) Recall who is responsible for testing in the DevOps pipeline

3.2 Testing During Planning

LO-3.2.a (K1) Recall how acceptance criteria can guide testing

3.3 Testing During Coding and Building

LO-3.3.a (K1) Recall the role of unit tests in the DevOps pipeline
LO-3.3.b (K2) Explain the use of static analysis in the DevOps pipeline
LO-3.3.c (K1) Recall the order of testing and development in TDD
LO-3.3.d (K2) Explain the use of Arrange-Act-Assert and Red-Green-Refactor
LO-3.3.e (K3) Apply TDD in a given scenario (e.g., select the proper test for a requirement)
LO-3.3.f (K3) Apply BDD in a given scenario (e.g., select the proper feature file for a story)

3.4 Testing During Staging

LO-3.4.a (K2) Explain the types of testing that should occur during staging

3.5 Testing During Deployment

LO-3.5.a (K1) Recall the purpose of testing during deployment

3.6 Exploratory Testing

LO-3.6.a (K1) Recall the purpose of exploratory testing in the DevOps environment

3.7 Continuous Testing in the DevOps Pipeline

LO-3.7.a (K1) Recall how DevOps testing helps to reduce quality debt

3.1. DevOps and Testing

DevOps changes the definition of the traditional tester. With continuous integration, continuous delivery, and the overall goal of increased efficiency, DevOps requires

testers to take responsibility for quality across the entire development lifecycle, not only during the traditional testing phase.

When following an Agile lifecycle, testing becomes an activity that the development team (business analysts (BAs), developers, testers) owns together. Various activities during development have an influence on testing, from requirements - where collaboration and testing is done via Behavior-Driven Development (BDD) - to customer involvement during acceptance testing. All of these change the scope and pace of testing and involve the whole development team. Testing in a DevOps environment takes this even further. DevOps pushes the development team to actively engage with broader teams within their organization. This involves not only the teams that develop the product, but also operations teams that release and support the product.

Testing in DevOps spans the entire software development and delivery lifecycle. Testers are no longer just focused on functional testing and feature verification. The DevOps pipeline consists of various types of testing done throughout the different stages. The way to approach testing in DevOps is to break down the pipeline into a few parts and apply different types of testing in each part.

The following is a simple DevOps pipeline that is considered in this syllabus:
Plan → Code → Build → Stage → Deploy

This pipeline is sometimes expanded or modified as shown below to include testing and monitoring steps:
Plan → Code → Build → Test → Release → Deploy → Operate → Monitor
Plan → Create → Verify → Package → Release → Configure → Monitor → Repeat

Regardless of the pipeline steps, testing must be fully integrated into the lifecycle. The next sections discuss how this integration should occur in the simple pipeline.

3.2. Testing During Planning

Testing during the planning stage includes gathering the requirements from the right stakeholders, documenting those requirements accurately, and ensuring testability by defining proper acceptance criteria. Requirements defects can range from ambiguous to incomplete to incorrect. The time and effort invested in implementing incorrect or poorly defined requirements is difficult to recover and costly for the project.

Reviewing requirements, either formally in a review meeting or informally as tests are being defined, is a form of pre-CI/CD static testing. Static testing is testing without executing any code and can be performed manually (reviews) or automated by use of tools (static analyzers).

When the requirements are in the form of user stories, the definition and review of the acceptance criteria help to define the story and the expectations from the

implementation of that story. Clear definition of the acceptance criteria eliminates the disconnect between what the developer implements and what the user wants. By collaborating to clearly define the acceptance criteria, the developer knows what to develop, the tester knows what to test and what outcomes to expect, and the user knows what they are getting in the final product.

3.3. Testing During Coding and Building

When it comes to the "Create" step of the DevOps pipeline, the focus for testing should be on static testing of the code (static analysis), unit testing and integration testing. In each phase of code development, there must be a testing activity to ensure that only high quality software moves to the next phase (such as moving from unit testing to integration testing).

3.3.1. Static Analysis of Code

During the "Code" and "Build" stages of the DevOps pipeline, static analyzers are used for code reviews before integrating the code as part of a build.
Static analysis tools (often referred to as linters, sanitizers or static analyzers) are used to perform static testing. They help improve the quality of code by removing unnecessary fluff and performing automated checks. A few examples of checks that they can perform include the following:

- Adherence to code style conventions and standards
- Inefficient use of resources, such as creating memory leaks by not freeing memory after use
- Incorrect usage of Application Programming Interfaces (APIs)
- Common security vulnerabilities, such as those identified by the Open Web Application Security Project (OWASP) or Common Weakness Enumeration (CWE)

The tools will identify issues in the code and may recommend changes that should be made to correct the problems. Developer action is to ensure that the tools' recommendations are valid and perform any necessary corrective actions.

3.3.2. Unit Testing

Unit tests are designed to verify functionality at the "unit" or smallest testable module level. These tests are usually written by developers, sometimes as they develop the code, and are executed each time the unit under test is being built. Developers are responsible for ensuring that each unit of code has associated tests and may also be responsible for achieving a defined level of code coverage (such as 85% statement coverage). Unit tests must be maintained and updated as new functionality is added to the unit.

Unit tests should be automated and are an essential foundation for all other types of testing in the DevOps pipeline. Unit tests are simple tests; easy to write, and fast to

execute. When put together, they help ensure the module works as expected. Unit testing primarily falls in the “Create” stage of the DevOps pipeline.

3.3.3. Test-Driven Development (TDD)

A common practice for writing unit tests is Test-Driven Development (TDD) [Beck04]. TDD is a programming methodology that deliberately combines design, testing, and coding in a rapid iterative process where a test-first approach is taken to develop a product. The test, which verifies that the proper functionality is implemented, is written and run before the code itself is written. When first written, the test will fail. Once the code is written, the test is executed and the code is debugged until the test passes. Once the test passes, the code and unit tests are refactored to improve efficiency and maintainability. TDD helps developers think through the desired behavior of each unit they are building, including inputs, outputs, and error conditions that may be associated with that unit before production code is written.
TDD follows these steps for a new feature (user story):

1) The developer first creates a failing test
2) Test(s) is executed (the new test should fail since there is no code to test it against)
3) Developer writes code to implement the feature
4) Test(s) is executed (repeat steps 3 and 4 until all tests pass)
5) Refactor code (if necessary)

There are many unit test frameworks available for different languages which can be used to write unit tests using the TDD approach. These frameworks differ in their APIs, approaches, and terminology, but all share some common elements. The most popular approach, generally called xUnit, was pioneered by the SUnit framework for Smalltalk, and later made popular by the JUnit framework for Java. xUnit has been ported to or implemented in a large number of languages. Almost all frameworks provide libraries and platforms that can accommodate features such as:

- Arrange-Act-Assert
- Red-Green-Refactor

Arrange-Act-Assert
Regardless of the language or unit test framework, a unit test typically follows this three-step pattern:

1. Arrange – prepare the execution environment
2. Act – execute the operation being tested
3. Assert – check the result produced by the operation

The Arrange step may instantiate objects, initialize the application state, and inject data as necessary. The Arrange step may not be needed if the environment is already in place, but sometimes setting up the execution environment can be difficult

due to a reliance on complex dependencies in the production code. Service virtualization may be used at this stage to substitute for services that are not yet working. TDD encourages loose-coupling and minimizing dependencies, which significantly contributes to simplifying the setup code.

The Act step usually consists of just one line of code that triggers the operation being tested. It is mandatory.

The Assert step, which is also mandatory, performs the actual check of expected outputs or other post-conditions. In most unit test frameworks, an error or exception causes an assertion to fail. Many have specific types of exceptions that test for specific kinds of failure. When no error occurs, the test passes.

To improve isolation, a single unit test should usually have a single assertion. It is important to remember that a unit test is only valuable if it has accurate assertions that are verified with each execution.

Red-Green-Refactor

The iterative process is the cornerstone of TDD. The purpose of each iteration is to make a small, focused, carefully considered change to the program by writing the tests for it first. Following a color-coded convention, the iterative cycle is often called the Red-Green-Refactor cycle:

- **Red** – Write a failing test that describes an unimplemented expectation. Run the test to ensure that it fails.
- **Green** – Write production code that satisfies only the expectation described by the test and causes it to pass. Run all the relevant tests to ensure that they all pass. If any fail, make the changes necessary to cause all the tests to pass.
- **Refactor** – Improve the design and structure of both the production code and the test code without changing functionality, while ensuring that all the relevant tests continue to pass.

An Example

The following is an example used to demontrate developing a feature using TDD. The example used here is a common interview question asked by employers and is called the "*FizzBuzz*" problem.

Write a program that prints the numbers from 1 to the input number (up to 100). For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz. An example input and output would be:

Input: 15
Output: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz

This example only walks though one test case. A complete implementation of FizzBuzz will require more test cases. It is highly encouraged that when you have learned how to create the one test, you practice and write the complete test cases and the complete FizzBuzz program using the TDD approach.

NOTE: This example uses pseudocode which will not run “as is” in a coding environment. In order to implement this, the correct syntax for the chosen language would have to be used. Psuedocode allows the developer to think about the logic without having to worry about the syntax of a particular language.

The steps to developing the first feature of FizzBuzz are as follows:

1) Write a test that tests if a given number results in "Fizz". Note the code that results in “Fizz” is not written yet.

```
@Test
function testMultipleOfThree:
     result = translateNumber(3);
     assert("Fizz", result);
 }
```

2) Run the above test and it should fail because the translateNumber() function has not been written yet.

3) At this point it is necessary to implement the translateNumber() function.

```
@production code
  function translateNumber(input):
      multipleOfThree = ((input % 3) == 0);
      multipleOfFive = ((input % 5) == 0);

      if (multipleOfThree && multipleOfFive) {
          return "FizzBuzz";
      }
      else if (multipleOfThree) {
          return "Fizz";
      }
      else if (multipleOfFive) {
          return "Buzz";
      }
      return String.valueOf(input);
```

4) Run the test again. You may have to do this multiple times while you are on step 3 and iterate over step 3 and 4 several times until the test passes.

5) Refactor the code in step 1 and 3 to make sure good coding standard and programming practice is followed.

6) To complete this assignment, you would then need to put the code into a loop that counted from 1 to the input number to provide the output given in the example.

3.3.4. Behavior-Driven Development (BDD)

TDD works very well at the unit level, but the tests still reflect the developer's perspective. In order to consider the goals of implementation, the perspectives of other stakeholders must be considered and this is accomplished using the Behavior-Driven Development (BDD) approach. Dan North, the founder of BDD explains BDD as *"using examples at multiple levels to create a shared understanding and surface uncertainty to deliver software that matters."* [North19]

BDD extends TDD by writing test cases in a natural language that non-programmers and domain experts can read. It uses the underlying concept of TDD, but instead of thinking about writing a failed test, BDD starts with writing a feature. It then follows the same process as TDD to write a test for the feature and code to pass the feature test. In this way the feature is traced all the way up to a requirement.

Although it still has much in common with TDD from a technical perspective, BDD intentionally uses different terminology to reflect its distinct goals and approach. For example, TDD relies on unit tests to verify implementation details whereas BDD relies on executable scenarios to verify behaviors. The low-level technical implementations are often similar, but the formulations are very different.
BDD follows these steps:

1) Identify the business feature (often provided as a user story or epic)
2) Identify scenarios under the selected feature
3) Define steps for each scenario
4) Run feature and fail
5) Write code and implement the feature
6) Run feature and pass. Repeat steps 4 and 5 several times if necessary
7) Refactor code

BDD features are usually defined in a Given/When/Then format, which is a semi-structured way of writing the test cases as shown in the following:

- Given – describes the state of the environment before the behavior is triggered
- When – describes the action or actions that trigger the behavior
- Then – describes the expected result or results of the behavior

BDD scenarios can be coded and executed using common unit testing frameworks available in most modern programming environments. When coded in this way, BDD scenarios are essentially written as unit or integration tests and BDD may be applied

in much the same way as TDD for testing in the pipeline. After the feature file is written to capture customer requirements properly, developers and testers start implementing the necessary test cases and test suites. Like TDD frameworks, there are many BDD test frameworks available. There are many TDD frameworks that extend the functionality to write features and derive test cases from the features using test runners such as jUnit, TestNG and Mocha.

An Example

The following example takes the FizzBuzz user story and implements it using the BDD approach to create the feature file. The feature file is the scenario or set of scenarios that will be used to test the feature itself, as described in the story.

```
Feature (user story): Play FizzBuzz
    As a Math game nerd
    I would like to play the math game Fizz Buzz
    so that I can learn how to calculate multiples

Scenario: When to Fizz
    Given I am a FizzBuzz player
    When I ask to FizzBuzz for 3
    Then I should get a Fizz

Scenario: When to Fizz Buzz FizzBuzz
    Given I am a FizzBuzz player
    When I ask to FizzBuzz for <value>
    Then I should get a <display>

Examples:
```

value	***display***
1	1
2	2
3	Fizz
4	4
5	Buzz
6	6
7	7
8	8
9	Fizz
10	Buzz
11	11
12	Fizz
13	13
14	14
15	FizzBuzz

This feature file is then used by the BDD framework and each of the Given/When/Then steps are converted to test steps. Code is written to create the test cases for those steps in a very similar way to how test cases were created using TDD.

3.3.5. Integration Testing

It is often the case that integration or API tests require custom data from different sources, such as databases and/or third party APIs. For this reason, it is best to mock (stub) or virtualize any external connections to free the testing from any downtime or unavailable services. The more complex or unstable the integrating components, the more important it is to have stubs or virtualized services in place to avoid significant schedule issues.

3.3.6. API Testing

API testing is an approach to testing that focuses on the interfaces between software components rather than the user interface that sits on top of the APIs. API testing can identify a variety of defects including functional issues where the right data is processed incorrectly or where incorrect data is not detected and reported properly. By testing at the API level rather than from the user interface level, testing can start sooner and is less vulnerable to changes that may occur with the User Interface (UI). It also allows a wider range of data to be tested and provides greater control for testing error handling.

3.4. Testing During Staging

A critical factor to remember for testing during "Staging" is that the environment in which testing is to take place must be representative of the production environment. Once there is a stable build, the build can be tested in a staging environment using functional, non-functional, performance, security, and any other type of testing. Functional testing is testing the implementation of the functional requirements for a system, to ensure the system does "what" it is supposed to do. Non-functional testing examines other quality attributes of the system to determine "how" the functionality is supplied.

3.4.1. Functional Testing

Functional testing is discussed in the Essentials of Software Testing syllabus [EoST]. In the DevOps pipeline, functional testing plays an important role at the "Staging" phase, although some functional testing will have already occurred during earlier testing. At this point, the entire system should be available for testing end-to-end. Applying black-box testing techniques at this point in the testing will help to achieve the desired coverage and ensure the implemented functionality meets the specified requirements (e.g., acceptance criteria in stories).

3.4.2. Performance Testing

During “Staging”, performance testing is conducted to identifying bottlenecks and any performance issues within the system. During performance testing, scripts that emulate real user behaviour are executed in quantity to place demands on the software as would be seen under normal and peak load conditions. Performance testing measures attributes such as response times, throughput rates, resource-utilization, and identifies the application’s breaking point.
One of the main reasons that performance testing is often neglected is because performance tests can take a long time to run and can be resource intensive. DevOps and its pipeline allow the execution of performance tests early, as well as running some performance tests in parallel with unit and integration tests. As functional tests are executed in a staging environment, performance tests for the whole system can be run in parallel. Accurate performance test results require the staging environment to be representative of the production environment, including database sizes as well as system configuration.

3.4.3. Security Testing

There is a type of DevOps called DevSecOps, where security testing is no longer considered an afterthought but is an integral part of the DevOps pipeline. Security testing is specialized testing, and therefore a partnership with security experts is needed to perform such types of testing in the pipeline. In DevSecOps, security tools are identified and integrated into the DevOps toolchain and the results are made visible to the whole team. This type of testing allows for early and continuous security testing to help identify any security problems early.

3.5. Testing During Deployment

During the “Deployment” phase, testing is conducted in the target environment to verify that the deployment has completed correctly and that the software is fit for use in that environment. Some of this testing is a re-execution of earlier functional testing, but combinatorial testing may also be included. Combinatorial testing verifies that non-interacting (independent) parameters or conditions are tested for unexpected interactions. These may include environment conditions (O/S, browser-type, etc.). A more complete discussion of combinatorial testing can be found in the EoST syllabus [EoST].

Testing for deployment is often automated because the target system and the software stability are known. Maintenance on the automation scripts should be relatively low as compared to functionally testing a new build.

3.6. Exploratory Testing

Throughout DevOps testing, exploratory testing is used. Exploratory testing combines testing with learning about the software, and is dynamic in its approach, allowing the tester to vary the next test based on what was found with the previous test. This

testing can be thought of as the tester “exploring” their way through the software and following interesting paths.

Exploratory testing may be used as a formal technique, with a guiding charter and a time-boxed session, or may be informal where users “try out” the software to see if it is meeting their expectations. Although exploratory testing does not have any formal coverage metrics and sometimes lacks reproducibility, it still yields valuable results with little preparation or overhead. Lessons learned during these sessions can also be used to improve and expand the automated tests.

3.7. Continuous Testing in the DevOps Pipeline

The primary rule with testing in a DevOps environment is that the testing must be continuous and the outcome from the testing determines if the software is ready to move to the next phase. Defects are found early and addressed prior to the software being re-introduced to the build. By attending to quality issues as soon as possible, a DevOps environment avoids building a quality debt that becomes more and more difficult to address as the solution becomes bigger and more people are involved. Developing and testing in small increments allows the software to progress steadily and for a firm and reliable base to be built.

4. Automation – 120 mins.

Keywords

configuration management, DevOps toolchain, Infrastructure as Code

Learning Objectives for Automation

4.1 Automation in DevOps

LO-4.1.a (K2) Explain the purpose of test automation in DevOps

4.2 Test Automation

LO-4.2.a (K2) Explain the levels at which test automation should be implemented in a DevOps environment

4.3 Infrastructure as Code

LO-4.3.a (K2) Explain how Infrastructure as Code works in a DevOps environment

4.4 Configuration Management

LO-4.4.a (K1) Recall the meaning of configuration management

LO-4.4.b (K2) Explain how configuration management is used when creating an Infrastructure as Code

LO-4.4.c (K1) Recall the types of assets that are usually tracked with configuration management

4.5 DevOps Toolchains

LO-4.5.a (K1) Recall the goal of DevOps and the use of toolchains

LO-4.5.b (K2) Summarize the tools that can be used in the DevOps toolchain

4.1. Automation in DevOps

Test automation is a method used to conduct testing by using automated test scripts rather than manual test cases. The test scripts are small software programs written in a scripting or programming language that accomplish the goals of a test by controlling the inputs to a software module, and verifying that the results match the expectations. In its simplest form, a script mimics the user interaction with the System Under Test (SUT) and is programmed to report any variances from the expected behavior.

Automation plays a significant role in DevOps and DevOps testing. The Agile movement embraced automation of unit testing, acceptance testing, and continuous integration. DevOps ties these with automation of the deployment process, eliminating the boundary between developer-side automation and operation-side automation.

Continuous testing was explained earlier as one of the main activities of DevOps. Continuous testing would not be possible without automation; it would not be an efficient process if a large number of test cases were run manually in the DevOps pipeline. Automation in the DevOps pipeline includes test automation, but also extends to other areas.

4.2. Test Automation

Test automation occurs at multiple levels in the DevOps pipeline. Automation begins with automating the unit tests which are targeted at testing individual units or modules of code. Unit testing helps to ensure that all code has associated tests and no issues are introduced with each release of the code. As discussed in Section 3.3, this testing can be approached in several ways, but to be truly automated, tools are required to build and support the unit test framework that then supports the creation and execution of these tests.

Test automation continues with the various checkpoints in the DevOps pipeline. Build Acceptance Tests (BATs) or Build Verification Tests (BVTs) are commonly used to determine that a build has the minimal required functionality. These tend to be positive path tests and serve as a gate between the build and the deployment of the build for further testing. The BVT is often used to determine if the build can be deployed into the test environment in readiness for manual testing. The purpose of the test is to ensure that the code being deployed for testing is not broken to the extent that further testing will be blocked. Ensuring a good BVT set is always run will help to streamline the entire process and ensure that testing can proceed effectively.

After the build is deployed into the test environment, a full suite of test automation can be run against the released code. This suite of automated tests can be built through any framework and should be triggered as the final step in the deployment process. Building maintainable test automation is covered in other syllabi such as the EoST syllabus [EoST]. It is important to remember that test automation in the DevOps environment is critical to the success of the process. Any test automation must be crafted to be maintainable, as there is an expectation that it will be actively used until the software under test reaches end-of-life.

4.3. Infrastructure as Code

Historically (and still in many organizations today), configuring an environment (test or staging) to run software was a manual process. A person (possibly a BA or tester or both) would identify the specifications for the environment and hand it to the Ops team. Someone on the Ops team would purchase the necessary hardware, install the operating system, prerequisite applications and libraries, then the product itself. Then someone might tune the parameters of the server, define the settings of the product, and create any automated jobs required to maintain the environment and hope that it mimics the production environment. This process could take a long time, be prone to errors, and was not easily repeatable.

In DevOps, CI/CD requires the automatic deployment of configuration changes to the test and/or production environments but also expects that these target environments are launched automatically as needed. CI/CD also requires an automated way to push the latest build to these environments and, eventually, to the customer's environment. "Infrastructure as Code" includes the process and technology needed to provision and manage environments (physical and/or virtual) through scripts.

Treating infrastructure as code is a key element of DevOps. It benefits both the development and the operations teams. Infrastructure as Code allows operations teams to get involved in the development process from the beginning, and developers can gain a better understanding of the supporting infrastructure because they are involved in specifying and understanding configurations for servers, networking, storage, etc.

4.4. Configuration Management

One of the main tenets of DevOps is to deliver software into the hands of the users in an efficient, repeatable and reliable manner. Configuration management has a very important role to play in this process to ensure the right software is delivered to the right people at the right time.

Configuration management is the method by which functional and physical characteristics of a configuration item are identified and documented. Configuration management is also usually extended to controlling, documenting and tracking changes to the item. This configuration item could be a module of code, a set of parameters required to configure a system or a particular set of instructions for hardware setup. Configuration management is critical for knowing which version of each piece of a system is required, how it is created and where it is stored.

As part of the Infrastructure as Code implementation, configuration parameters for the system are captured and stored in the code. The instructions to apply the parameters are executed each time an environment is created, thus ensuring that the configuration is consistent across each deployment. By centralizing the storage of these parameters and tightly controlling any changes, the true configuration is kept in a reusable form and environment configuration can occur quickly and reliably.

Version control is a part of configuration management. Each item tracked by configuration management is given a version. This version is incremented every time there is a change to the item. In this way it is easy to see what has changed from version to version and rollback is possible if a new version was not implemented correctly. For code, the version is incremented each time the code is released for a build. An entire build of the software is also versioned, and the individual versions of each module are recorded along with the version of the composite build. For configuration parameters, such as how much disk space to allow, versioning is done in the same way. Anytime something is changed in a particular parameter, the version

is updated. These individual parameters roll up together to make a version for the entire configuration.

Examples of managed assets generated throughout the pipeline include source code, build scripts, installation scripts, configuration files, infrastructure (software and hardware configurations known as Infrastructure as Code), test cases and test scripts. These assets are created and maintained by different people in the team and should be accessible by the different teams at any time. All these assets needs to be version controlled.

4.5. DevOps Toolchains

The ultimate goal of DevOps is to streamline development through to delivery. DevOps does not necessarily require tools and is not defined by its tools. However, for most organizations, the tools play an important role in automating tasks and ensuring processes run as efficiently as possible.

A DevOps toolchain is a combination of tools that help in development, integration, testing, deployment, delivery, and management throughout the software development lifecycle. The DevOps toolchain enables teams to move smoothly through the DevOps pipeline. It should include tools from different categories such as:

- Project planning and management tools
- Requirements engineering tools
- Configuration management and provisioning tools
- Source code scanning (for quality and security) tools
- Build automation and continuous integration tools
- Functional and non-functional testing tools
- Deployment tools
- Release orchestration tools
- Application and infrastructure monitoring and performance tools

It is critical to design a DevOps toolchain that is adaptable to accommodate changes in team preference, application architecture, quality processes, and other technology shifts. In order to maintain an effective DevOps pipeline, DevOps teams must choose the right tools to build their toolchain.

5. Management – 160 mins.

Keywords

none

Learning Objectives for Management

5.1 Cultural Changes

LO-5.1.a (K2) Explain why a team may be averse to collaboration and how this can be overcome

LO-5.1.b (K2) Explain the roles of specialists vs. generalists in a DevOps implementation

5.2 Management Challenges

LO-5.2.a (K2) Summarize the challenges management faces in a DevOps environment

LO-5.2.b (K3) For a given scenario, identify the key challenges and select the best approach to deal successfully with the challenge

5.3 Barriers to DevOps Adoption

LO-5.3.a (K2) Summarize the barriers to the adoption of DevOps

5.4 Success Factors

LO-5.4.a (K2) Summarize the success factors in a DevOps implementation

LO-5.4.b (K1) Recall why goals must be clearly stated

5.1. Cultural Changes

DevOps is not only a process change, it is also a cultural change. Traditional ways of working, established responsibilities and roles, and even job satisfaction for individuals all undergo a metamorphosis when DevOps is implemented. Each of these must be understood and managed in order for the transformation and continued operation to be successful.

5.1.1. Eliminating Silos

Silos develop in organizations, whether by necessity or tradition. This involves a concentration of expertise and a feeling of ownership within an area of the business. Traditionally, developers develop, testers test and operators keep the system running in production. Understanding all the tasks performed across these roles is critical for the successful implementation of DevOps. Not having a full understanding, or worse,

assuming tasks are not needed, will inhibit the implementation and will also alienate important stakeholders.

Successful elimination of silos first requires understanding the purposes and responsibilities of each siloed area and sharing that understanding across stakeholders. This will allow for an accurate assessment of what can be effectively automated, which activities may be redundant, what must be kept and what can be eliminated. For example, it may be determined that the development team writes a set of deployment instructions, but the operating team writes their own instructions. This is a possible area for automation, but also an opportunity to look at why the teams are developing individual sets of instructions. Are they different? Automating from the developers' instructions may be the wrong thing to do. It will be more efficient for the operators to be involved during development to understand how the software will be deployed.

5.1.2. Collaborative Work Environments

Just telling everyone to collaborate will not make it happen. Collaboration requires sharing knowledge and expertise, which may be perceived by the team as losing their importance to the organization. This is particularly true in an organization that has had workforce reductions in the past. Knowledge is often equated to power and value. Stakeholders may be resistant to collaboration, openly or covertly, because of fear. It is critical in a DevOps environment to remove this fear factor and ensure that everyone's contribution is valued. The team should succeed as a whole, not as individuals. This requires a large degree of trust and management reinforcement to allow people to retain their feeling of importance and responsibility to the organization.

Management can help effect this change by clarifying the expectations for individuals and rewarding collaboration rather than individual achievements. This will help to reduce fear and will encourage the necessary behavioral changes for a successful DevOps implementation.

5.1.3. Specialist vs Generalist

The software industry tends to value specialists over generalists, both in recognition and pay. Specialized skills are still needed in the DevOps environments, as well as deep knowledge in particular areas (such as performance and security testing). However, this has to be mixed with generalized knowledge of the entire process, from product conception to deployment to support. This may also require adjustment of reward systems in an organization where specialists have a higher level of recognition.

Building generalists with some specialist knowledge is often a good compromise. In a large organization, it may make sense to keep the specialist roles if the individuals can be shared across teams. This can help an organization ensure they are using consistent methods for automation and specialist testing across multiple product teams. Ideally, teams meet frequently to share best practices and lessons learned, to help build organizational knowledge and capability.

5.2. Management Challenges

There are a number of challenges management faces in a DevOps environment, particularly when converting to DevOps. The less agile the current lifecycle, the bigger the adjustment into DevOps. The following list highlights some of the biggest challenges that an organization is likely to face:

- Mindset changes – People can be uncomfortable and nervous with changing from distinct roles to collaboration, specialist to generalist, and moving to more dependence on automation. Management needs to ensure that people understand these are opportunities to learn, develop skills and modernize the approach to development through to deployment. It is important to remember that individual job satisfaction may change and the rewards will be different. A career software tester may get job satisfaction from finding and reporting defects rather than assisting in the resolution. The mindset change from "doing the job" to "releasing the product" can be difficult and must be facilitated by management support and reinforcement.
- Reward systems – It is important to adequately and fairly reward people for their long term contribution. Motivation is difficult to maintain and must be frequently revisited to ensure the right behaviors are encouraged. In a DevOps environment, rewards should focus on the goal of the team – which is to release quality products efficiently. Individuals will be varied in technical depth, but contribution to the team and the necessary skill mix must be considered in the rewards. Not everyone needs to be able to automate infrastructure deployment; someone with strong domain knowledge who can identify critical business processes and provide test data may be just as valuable to the team. It is important for the manager to not fall back into rewarding specialist skills over generalist contributions because both are needed in a successful DevOps environments.
- Who reports to whom and when – Reporting structures are often changed when DevOps is implemented. It is important to remember that people need to understand who they report to administratively. Reporting into a project team and working out assignments within the team can work effectively with motivated and self-managing individuals, but long term career planning, compensation, training and administrative tasks still need to belong to a management individual who has the knowledge and authority to provide the management support for individuals to succeed.
- Training – To help minimize discomfort and to allow career growth, adequate training must be provided so that individuals can improve their skills. In DevOps there is a strong emphasis on automation for testing and deployment. Training in these areas must be provided for individuals to participate effectively in the team and feel that they are contributing.
- Managing budgets – Regardless of the methodology, budgets are a reality in business. This includes budgets for tools, training, hiring, compensation and

environments. A test manager must understand the new cost areas, particularly tooling and environment costs, to ensure that adequate budget is allocated for a successful implementation of DevOps.

- Proper tooling – DevOps tools have been discussed in section 4.5, but there are important management challenges around procuring and implementing the proper tools. It is important to have valid technical input into proper tooling and to understand the expected lifespan of the tools and the return on investment (ROI), in order to make proper selections for the organization. New tools appear on the market frequently, but these may be untrialled and have limited support. Understanding the timespan for the use of the tool is important to ensure a safe, reliable, and feature-rich option is selected. New and shiny tools may be attractive and appear to have the best new capabilities, but they may not be the best choice for the needs of the organization.

5.3. Barriers to DevOps Adoption

Despite all the benefits that DevOps can deliver, there are often significant barriers to adoption. Some of the most common barriers include the following:

- Perceived loss of control – DevOps requires collaboration and sharing information. To some, this is seen as a loss of control and loss of power. The best way to combat this is to ensure that the benefits are clearly demonstrated and the reward system encourages cross-team working and sharing.
- Territorial battles – Ownership, particularly of certain areas of the business, becomes less important and may actually be counter-productive to an effective DevOps implementation. When territories are defined and protected, knowledge is not shared and the implementation may be set up to fail. For example, if the operations staff does not want to share information that the team needs to automate deployments, the effort will be considerably more difficult.
- Misunderstandings and misconceptions – As discussed in Chapter 1, there are a number of misunderstandings, misconceptions and unrealistic expectations that surround DevOps. This can result in schedules that cannot be met, goals that are not feasible and general discouragement of the team. Upper management may brand the attempt as a failure purely because the expectations were never realistic.

To help avoid these barriers, it is vital to ensure that the stakeholders understand the following:

- What is DevOps?
- What will it produce?
- How long will it take?

The stakeholders must be taken on the full journey with the implementation team to help remove barriers and set realistic expectations.

5.4. Success Factors

Establishing visible goals will allow all stakeholders to view progress and assess the success of the effort. Of course, these goals must be realistic and clear or the likelihood of failure increases. Some common measures of success include:

- Successful handoffs – A product is handed off smoothly from development to testing to deployment phases of its lifecycle. Each handoff is measured for quality and that measurement is reported and reviewed for possible areas of improvement.
- Team engagement – In a successful DevOps implementation, the entire team is engaged. They have their roles, understand their success measures and work together to create the most efficient environment possible.
- Goal definition and fulfillment – Success can be measured only when clear goals are defined. Satisfaction can be obtained when goals are met and there is a clear achievement made by the team. This also means that when goals are fulfilled, a celebration should be made and the achievement acknowledged. Expecting success without celebrating it tends to demotivate people. A strong team will expect success, but will also celebrate when it occurs.

For a DevOps implementation to be considered successful, realistic goals must be stated, tracked and achieved. Specific goals may vary between organizations, particularly if DevOps is being implemented to correct previous failures, but it is important to ensure that schedules, budgets, training and effort are all adequate for achieving the stated goals.

6. References

6.1. ISO/IEC/IEEE Standards

- ISO/IEC/IEEE 12207:2017 Systems and Software Engineering - Software Life Cycle Processes
- ISO/IEC/IEEE 15288 Systems and Software Engineering – System Life Cycle Processes

6.2. AT*SQA Syllabi

[EoST] Essentials of Software Testing, 2020. AT*SQA. www.atsqa.org.

6.3. Trademarks

The following registered trademarks and service marks are used in this document:

- AT*SQA® is a registered trademark of the Advancement of Testing and Software Quality Assurance

6.4 References

[Beck04] Beck, Kent and Cynthia Andres. 2004. Extreme programming explained: embrace change. Addison-Wesley Professional.

[Debois11] Debois, Patrick. 2011. Devops: A software revolution in the making. Cutter IT Journal 24, 8 (2011), 3–5. Code: B4.

[Gregory19] Gregory, Janet and Lisa Crispin. 2019. Agile Testing Condensed: A Brief Introduction. Library and Archives Canada. ISBN-10: 199922051X.

[Kim12] Kim, Gene. 2012. The Three Ways: The Principles Underpinning DevOps. (2012). http://itrevolution.com/the-three- ways-principles- underpinning-devops/, accessed on Jul 2018.

[North19] North, Dan (31 Dec 2019). "BDD is like TDD if…". faster organisations, faster software. Dan North & Associates.

6.5. Other Sources of Information

[Kim16] Kim, Gene, Patrick Debois, John Willis, and Jez Humble. 2016. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press.

[North06] North, Dan. (March 2006). "Introducing BDD". Dan North. Retrieved 25 April 2019.

[Smart14] Smart, John Ferguson (2014). BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle. Manning Publications. ISBN 9781617291654.