Foundations of DevOps Learning Outcomes

International Consortium for Agile

LICENSING INFORMATION

SPECIAL THANKS

ICAgile would like to thank the contributors to the Foundations of DevOps Learning Outcomes:
Bryon Brewer • Colin Garlick • Dominica DeGrandis •
Gene Gotimer • Tim Guay • Chris Knotts

CONTENTS

HOW TO READ THIS DOCUMENT

This document outlines the Learning Outcomes that must be addressed by accredited training organizations intending to offer ICAgile's Foundations of DevOps certification.

Each LO follows a particular pattern, described below.

0.0.0. Learning Outcome Name

Additional Context, describing why this Learning Outcome is important or what it is intended to impart.

The Learning Outcome purpose, further describing what is expected to be imparted on the learner (e.g. a key point, framework, model, approach, technique, or skill).

LEARNING OUTCOMES

1. THE CASE FOR DEVOPS

1.1. HISTORY OF DEVOPS

1.1.1. Origins of DevOps

DevOps is the extension of Agile principles beyond the delivery of software by including the Operations team and everyone else involved in the software delivery in the process from the beginning and addressing operational and other concerns as an integral part of the development cycle. Patrick Debois suggests, "Operations becomes a valued member of the traditional Agile process with an equal voice." This is a departure from early thinking where development and operations evolved their tools and methods independently of one another.

Distinguish DevOps from traditional approaches, make clear what was and was not intended with DevOps and anchor the ideas of DevOps in Agile principles.

1.1.2. DevOps Business Value / Benefits

DevOps delivers value quicker by reducing risk. DevOps focuses on systems thinking, frequent feedback loops and continuous improvement to bring software to production and allow deployment decisions to be business-driven, not tool or technology driven.

Explain the benefits and value of DevOps in order to set the organization's expectations and to gauge progress throughout adoption.

1.2. MINDSET AND PRINCIPLES

1.2.1. DevOps Principles

Many people come to DevOps looking for the practices that will make them "DevOps". But adopting DevOps is not just adding certain steps or particular people to the team, it is a shift in mindset based on core principles, the most important of which is that everyone involved in the delivery of software is involved throughout the application lifecycle. Much like the Agile Principles, the DevOps principles describe criteria that allow you to make intelligent decisions about how your organization will approach software delivery. The scope and breadth of DevOps is sometimes summed up as C.A.L.M.S. -- Culture, Automation, Lean, Metrics, Sharing.

Many people come to DevOps looking for the practices that will make them "DevOps". But adopting DevOps is not just adding certain steps or particular people to the team, it is a shift in mindset based on core principles, the most important of which is that everyone involved in the delivery of software is involved throughout the application lifecycle. Much like the Agile Principles, the DevOps principles describe criteria that allow you to make intelligent decisions about how your organization will approach software delivery. The scope and breadth of

DevOps is sometimes summed up as C.A.L.M.S. -- Culture, Automation, Lean, Metrics, Sharing.

1.2.2. Systems Thinking

According to W. Edward Deming, a system is “two or more parts that work together to accomplish a shared aim.” With systems thinking everyone gains, not part of the system at the expense of another. In a DevOps culture, this is fueled by communication and collaboration among the business, development and operations teams. This involves collaborating on trade-off decisions, and a consistent multi-directional flow of discussions and information.

Show the importance of adopting the culture of systems thinking across all parts of the organization and value stream (avoid optimizing only locally.)

1.2.3. Definition of Done

One of the keys to successfully adopting DevOps is to explicitly define what “done” means for their organization. Optimally “done” means delivered, completely tested, deployed in production and fully monitored. However, each organization and even each team will have to define what those terms mean for them, and even if they can get that far in their particular situation.

Explain that DevOps will vary from organization to organization, from team to team and that it may even change over time. It is not a “one-size-fits-all” process.

1.2.4. Reduce Risk

Software is never perfect. Our goal is to make it easy enough, fast enough, secure enough, safe enough. As we make the software “better,” we need to understand that we are reducing risks in use, security, deployment, etc. Considering risk may help us make intelligent decisions about where to invest effort in automating the integration and deployment of our software.

Explain how improving the software and the development process is about reducing risk involved in the development, deployment, operations and use of software.

1.2.5. Small, Frequent Releases

Small, frequent releases are both a goal and a result of continuous delivery. Increasing the frequency of releases lowers the scope and the risk of releases at the potential cost of the "end game" activities necessary to push software into a production environment. To achieve the goals with minimum cost, the releases have to be quick and easy, generally aided by substantial automation. Because the releases are quick and easy, they tend to be performed more frequently. The more frequent they are performed, the more practiced the team is at them and the more feedback they provide, in turn making them easier for the future.

Explain the value and impact of small, frequent releases in building a continuous delivery practice and a DevOps culture. Begin to emphasize the importance of automation as well.

1.2.6. Continuous Improvement (Kaizen)

Rapid delivery and release cycles provide natural break points where the team can review its processes, plans and products to improve them. Making lots of small changes avoids the shock and effort of having to make sweeping changes all at once, not to mention reducing people's natural reluctance to change.

Explain the importance of continuously incorporating change in the processes and practices of a DevOps organization. Provides guidance on establishing feedback loops and metrics against which to gauge success.

1.3. CULTURAL CHALLENGES

1.3.1. Essential Conflict

There exists an essential conflict between development and operations that must be overcome through collaboration, processes and tools; that is that development produces changes that must be deployed to production to be utilized, and operations wants to avoid change to ensure stable conditions for operating systems. These differences in motivation can be supported by policy, metrics and incentives.

Identify and examine factors that will hinder implementation and discuss techniques to overcome them.

1.3.2. Teams

The makeup of the project teams can have a profound impact on the success of DevOps. Permanent cross-functional teams made up entirely of jacks-of-all-trades with a strong understanding of the business and project goals are optimal. But most teams will have a mix of senior and junior people, each with areas of knowledge. The team as a whole will overall strengths in some areas and weaknesses in others. Team members will join and leave over time.

Explain the value of long-term, cross-functional teams and discuss how to handle individual abilities and weaknesses of team members.

1.3.3. Organizational Structure

Organizational structure can be used to eliminate the rift between development and operations created by rewarding some for innovation and change and rewarding other for stability, uptimes and performance (aka Service Availability). Short of creating one group encompassing both roles, there are a number of options for combining teams including inserting operations into a development team, or developers into an operations team, or (preferred) creating cross-functional teams with representation by both disciplines.

Describe how challenges arise from separation of development and operations, how organizational structure influences technology decisions (such as Conway's Law) and discuss solutions to address these challenges.

1.3.4. Confidence in Automation

"Streamlining operations is heavily reliant on end-to-end automation" (Huttermann 2012). As the deployment pipeline matures, more and more depends on automated testing, automated deployment and other

automated processes. If these automated processes begin to experience problems, confidence in the entire process will suffer and the DevOps model may begin to falter.

Explain the confidence aspect of automation in a delivery pipeline.

1.3.5. Distributed Teams

Teams that are geographically distributed, whether in separate rooms or separate time zones, can present some challenges to developing working continuous integration and continuous delivery processes. The most effective approaches rely on a shared version control system and access to the continuous integration system and feedback systems for the entire team. And adherence to the disciplines of continuous integration and continuous delivery can be much more important with remote teams.

Show how CI/CD practices work for distributed teams, emphasizing that the philosophy stays the same even where execution needs to be handled differently.

2. CONFIGURATION MANAGEMENT

2.1. VERSION CONTROL

2.1.1. Commit Everything

At any point in time you should be able to recreate the entire system from source code to tests to database scripts to build and deployment scripts to configuration files to specification documents. New programmers can go to a single location to get everything they need to set up a development environment. A business analyst can set up a demo environment to show to another team in the company or a prospective client. An operations group can set up an environment on new hardware to evaluate the impact on performance.

Describe the various assets that are committed into version control, including configuration files, installation scripts, data sets, any dependent software that gets installed as part of the process and documentation that supports it.

2.1.2. Working on Mainline

Working on the main line, or trunk, goes hand-in-hand with frequent commits. If developers spend their time working in independent branches of the source code, then they are not frequently integrating their code. They cannot see when they are introducing conflicts or using outdated versions of other developers' code. The same applies to other team members and their work. If they don't commit frequently, not everyone gets a chance to see how their code integrates with others. This applies to development of the application software as well as the automation that installs and maintains the software.

Explain the mainline concept and provide guidelines on how to test and integrate source code and DevOps code as part of a mainline development model.

2.2. MANAGING CONFIGURATION

2.2.1. Application Configuration

Proper management of configuration information is crucial to allowing environments to be deployed quickly. When an environment needs to be manually configured after deployment it takes time and introduces unnecessary risk. On the other hand, when configuration information is built into binaries or packaged with binaries, those binaries must be recompiled or repackaged for each environment. By automatically configuring software at deploy time or at run time, the same binaries can be deployed to each environment and the system is ready for use when it comes up.

Explain proper configuration techniques to support automated deploys, building binaries once and supporting multiple environments.

2.2.2. Feature Toggles

Feature toggles are sometimes known as feature bits or feature flags, and they describe a technique that can be used to allow new functionality or algorithms to be deployed incrementally. They also allow the deployment of the feature to be decouple from making the feature being enabled.

Explain the option of using feature toggles to incrementally deliver partially-finished features, and how feature toggles can be used to coordinate the deployment of a feature separately from making the feature available to end users.

2.2.3. Configuration Management Tools

No tools have been more critical to the adoption of DevOps than the configuration management tools that enable automated deployments and configuration. Tools such as Puppet, Chef and Ansible are so critical to successful DevOps adoption that in is not uncommon for organizations to assume that simply using one of these tools "makes them DevOps."

Describe the role of configuration management tools, discuss their declarative nature and explain how they enable automated deployments and configuration.

2.2.4. Third-Party Components

The purpose of version control is to be able to reproduce a configuration at any point in time. Libraries and other components that are not part of your source code need to be tracked so that a build isn't recreated with the wrong versions of third-party components. This is also critical for new team members to be able to get everything they need to build the software when they first join the project.

Explain the importance of keeping everything needed to build the software in version control. It also stresses the importance of reducing the number of external dependencies in a build system.

3. CONTINUOUS INTEGRATION

3.1. PRINCIPLES OF CONTINUOUS INTEGRATION

3.1.1. Overview of Continuous Integration

Continuous Integration is core to the DevOps philosophy. The goal of continuous integration is that software is in a working state at all times. Code changes from the developers on the team are frequently integrated and automated tests are run to demonstrate that the code still works after integration. Bugs and problems are caught quickly and fixed immediately.

Explain the principles of continuous integration and why they are important. It must strongly emphasize the importance of automated testing as part of the build. If you do not have automated tests, you are not doing continuous integration.

3.2. PRACTICES OF CONTINUOUS INTEGRATION

3.2.1. Commit Code Frequently

Frequent commits are central to both DevOps and to Continuous Integration. By team members committing their work frequently, teams can integrate changes frequently and in small doses, before the integration effort becomes a huge, unpredictable endeavor. Risk is reduced, tests can be run more frequently and everyone always has access to the most up-to-date version.

Explain the importance of frequent commits and integration by all developers.

3.2.2. Write Automated Developer Tests

Automated developer tests are tests that are fast and can be run at least after every commit, so they are sometimes called "commit tests". They do not have to be solely unit tests, but they should not require external resources that might not be accessible at all times. These types of tests are sometimes called "component tests", in that while they test individual pieces of code they are not completely isolated and are therefore not technically unit tests. A failed test means a failed build. This is often referred to as making the build "self-testing".

Explain the criteria for writing good developer tests for CI.

3.2.3. Prioritize Fixing the Build

A broken build may be a failed compilation, unsuccessful packaging, failed tests, or static analysis findings that move beyond an acceptable threshold. Since the goal is to always have working software, the team must prioritize fixing a broken build. Since whatever broke the build likely was just done and committed, it is probably fresh on the mind of whoever made the ill-fated change or changes. That makes it a lot easier to fix or back out than it would be to fix something done yesterday or last week. Also, all the other developers on the team are held up from committing, so fixing the build removes a roadblock for them as well.

Discuss the impact of broken builds and approaches to fix rapidly, including rolling forward and time-boxing the fix before rolling back the changes. Provide concrete examples of the pitfalls of temporarily disabling tests and code checks to “fix” a build. The build should not be fixed at the expense of thorough and complete automated tests.

3.2.4. Continuous Feedback

The point of continuous integration is to have software that is always in a working state. However, without feedback, there is no way to know if that is true. You need to see that a build worked, the tests ran successfully, the static analysis did not find any critical errors or too many less critical ones, etc. And on top of that, you need to act on the feedback. When a build breaks, you need find out and to fix it right away. Getting the software back to a working state is the team’s priority.

Explain the value and purpose of continuous feedback.

3.3. QUALITY ASSURANCE

3.3.1. Development Standards

Development standards can involve stylistic decisions (e.g., tabs vs. spaces, naming conventions, standard comment blocks), architectural decisions (e.g., MVC pattern, no singletons, REST interfaces), or what tools or components to use (e.g., no Open Source, only Open Source, no GPL, only GPL). What is important is that the team agrees on them and that they are documented so that new (and current) team members can find them and review/revisit them periodically.

Explain the importance of having documented development standards.

3.3.2. Static Analysis

Static analysis tools can check source code for coding style, naming conventions, confusing or dangerous code constructs, unused code or variables, potential security flaws and even design issues. They take care of a lot of the mindless, rote checking that would normally be done by peers in a code review, and they can take very little effort to install and maintain. The tools collect metrics and can produce dashboards to make it easy to see trends in quality.

Explain the value of automated static analysis tools for continuous integration.

3.3.3. Test Automation

Automating tests is often a major shift for organization that are adopting DevOps approaches. Even though the delivery pipeline relies on automated testing, the benefits and critical nature of automation are not always clearly understood. Testers who can write automated tests become critical enablers of the delivery pipeline. The costs in time, effort and money are frequently underestimated. And the expectations by management are not always realistic.

Show how the delivery pipeline is enabled by automated testing and discuss strategies for adopting automated testing as a focus for testers.

3.3.4. Types of Tests

Many types and levels of testing can be automated in DevOps organizations. It is important to be aware of these types as well as their benefits, their unique considerations for automation and how they each fit in to the delivery pipeline.

Explain the impact each level of testing has on quality and trade-off considerations related to automation and overall time and effort.

3.3.5. Managing Defects

Ideally, bugs are fixed as soon as they are found, preferably before the software leaves the development environment. But even the most disciplined development team will find that some bugs inevitably creep through to later stages, even into production. And teams that are supporting legacy software may have defect backlogs from the beginning. There are a number of approaches to dealing with the defect backlog, from dropping everything immediately to fix the bug, to treating it like a feature and prioritizing it similarly, to accepting the risk and consequences and living with the defect as-is.

Explain some of the approaches for dealing with defects that are found late in or after the development cycle, and how to handle that in a continuous delivery model.

4. CONTINUOUS DELIVERY

4.1. DEFINITION OF CONTINUOUS DELIVERY

4.1.1. Overview of Continuous Delivery

Continuous Delivery could be considered the logical extrapolation of continuous integration out of development. Whereas the goal of continuous integration is having working software at all times, the goal of continuous delivery is having software that is deployable to production at all times. In continuous deployment, each commit that passed the automated tests and checks is pushed into production. It is the logical extreme of continuous delivery; not only is every change a candidate for release, each change is automatically released, without manual intervention. While Continuous Delivery is critical in a DevOps organization, the term DevOps refers to the entire culture of collaboration and delivery.

Define continuous delivery and explain how it differs from continuous integration, continuous deployment and DevOps. Also include a discussion about the problems and risks involved in releasing software.

4.2. PRINCIPLES OF CONTINUOUS DELIVERY

4.2.1. Repeatable, Reliable Process for Releasing Software

To deploy your software, you need to do three things: provision the environment for the software to run in, install the software and configure the software. If these three steps can be automated, you can have push-button deploy. That makes software delivery very easy, which is the goal of continuous delivery: make the decision to release software a business decision, not a technical one.

Explain the value of a repeatable, reliable process for software delivery, which is the basis of the other principles of continuous delivery.

4.2.2. Automate Almost Everything

No matter how careful a person is, if they repeat a process manually enough times, they will make a mistake. Investing the time to automate a process will not only make the process more reliable and repeatable, but you can often save time in the long run. Also, once a process is automated, you'll find that you have fewer reasons not to exercise it, which means it becomes even more useful. Any process that does not require a human decision to be made is a good candidate for automation.

Explain the value of automating processes and the approach to bring automation into an environment.

4.2.3. Keep Everything in Source Control

At any point in time you should be able to recreate the entire system from source code to tests to database scripts to build and deployment scripts to configuration files to specification documents. New programmers can go to a single location to get everything they need to set up a development environment. A business analyst can set up a demo environment to show to another team in the company or a prospective client. An operations group can set up an environment on new hardware to evaluate the impact on performance.

Explain the importance of being able to create or recreate an environment for building, testing, or using the system at any point in time.

4.2.4. If it Hurts, Do it More Frequently, and Bring the Pain Forward

This principle is can be thought of as a guide for identifying where change is needed, choosing the areas for highest return on investment, and the parts of the process that need the most practice. Whether the pain is risk-induced or effort-related, the most painful parts of the process are usually the parts that need the most attention. If you can automate, simplify and/or practice those parts, then the next more painful areas become the next targets for improvement.

Demonstrate the principle of prioritizing change based upon ROI. Emphasize automation benefits, process simplification, risk avoidance, and targeting processes requiring large LOE investments."

4.2.5. Build Quality In

"Build quality in" was lean-pioneer W. Edward Deming's motto. Defects are easier to catch and easier to fix the earlier you catch them. If you can build the software with quality in mind from the beginning, you don't have to try to shoehorn it in later and the result is usually much better. Testing must be part of the development phase, not a phase after development is done.

Demonstrate applicable lean principles, present cost studies for defect identification over the product lifecycle and enumerate techniques to build quality into the process to reduce cost and risk.

4.2.6. Done Means Released

Ideally, a feature is not done until it is released into production. Practically, done might have to mean released to a staging environment. But a feature cannot be considered done just because it has been coded. It must tested, deploy automated and the business has to have reviewed it before it is ready for production. Otherwise it isn't ready for the business to release it to production so it cannot be "done". Each team will need to define done for their process, and that definition may change over time.

Explain the need to define done for the specific situation and environment.

4.2.7. Everybody is Responsible for the Delivery Process

Core to DevOps is the concept of shared responsibility. Delivery issues can't be "their problem," the entire team must be working to fix them. Work can't be thrown over the wall from one team to another, the whole team must be working together from the beginning. Everyone needs to help remove the barriers between the groups involved in development and operations and work together towards the common goal of delivering software. That means everyone will need to collaborate and to be able to see the status of the software at all times.

Explain this central concept of collaboration and shared responsibility.

4.2.8. Continuous Improvement

Just as your software will continue to be improved, so should your software delivery process continue to improve and mature. The team should discuss the process regularly and work on retaining what is working and to improve or replace what isn't. Holding and acting on retrospectives is key to improving and maturing the process.

Explain the importance of regular reflection and re-evaluation of the delivery process by the entire delivery team.

4.3. PRACTICES OF CONTINUOUS DELIVERY

4.3.1. Build Binaries Only Once

An important practice in continuous delivery is building your binaries at only one point in the development process. Each time source code is compiled it takes time and introduces a chance for a difference to creep in. By ensuring the exact same binaries are used from stage to stage in the pipeline, not only are you keeping the process efficient but you also ensure that latter stages are using a binary that is known to be good since it has been tested and verified in previous stages of the pipeline.

Explain the importance of building binaries only once in the deployment pipeline.

4.3.2. Same Deploy Process Everywhere

One key to making sure that the deploy process is repeatable and reliable is to use it repeatedly and then test to show that it behaves the same way each time. If you can do this in multiple environments, you gain confidence that doing it in production will work the same way. Also, your return on investment in maturing (and hopefully automating) the deploy process grows as it is used more often. On *the other hand, if you use different deploy processes for different environments, you have to spread your efforts across multiple processes to mature them.*

Explain the practice of using the same deploy process in all environments and to offer methods to accomplish this.

4.3.3. Smoke Test Your Deployment

Simply deploying the software isn't enough. You have to make sure the deploy worked. A smoke test is run after deploying to make sure the software is up and running and that basic functions work. External services that the application depends on, such as a database or authentication service, could also be checked.

Explain what a server validation smoke test is and why it should be used.

4.4. DEPLOYMENT PIPELINE

4.4.1. Definition of Deployment Pipeline

The deployment pipeline is the process of taking a code change from the developer and getting it deployed into production. Along the way, the software will pass through a number of stages, each one representing a higher level of confidence that the change is a viable candidate to be deployed into production. As more automation is brought into the process, the software can progress quicker and more reliably through the stages and the level of confidence grows. Continuous Delivery is often focused more on stages of assessment than on progression through static environments.

Explain what the deployment pipeline is and why it is important to develop one. Also, the deployment pipeline is itself a product that must be maintained.

4.4.2. Commit Stage

The first phase of the deployment pipeline begins when the code is committed. The code is compiled, unit tests and other quick commit tests are run, and usually static analysis takes place. This phase must be quick to inform the developer that the recent changes have not caused obvious problems. We would like as much feedback as we can get but the developer is stopped waiting for this feedback, so we can't afford to wait for comprehensive tests. Instead we want to get a quick level of confidence that these changes represent a viable production candidate, and that the time and effort of running further tests and checks is warranted.

Explain the purpose and the value of the commit stage as the beginning of the deployment pipeline in catching developer errors.

4.4.3. Automated Acceptance Stage

At this stage in the deployment pipeline, functional tests and regression tests are run to verify that the code works and meets at least a minimum level of functionality. In general, if tests fail at this point either an unintended change in behavior has been introduced, or a feature that was thought to be complete is not. The team is generally not waiting for this stage to pass before continuing *work on other features, but will still make it a priority to resolve any problems that are found during this stage.*

Describe the purpose and value of the automated acceptance stage to determine if the software has functionality that should be considered for release.

4.4.4. Manual Testing

The goal of progressing through the deployment pipeline is to gain confidence that the software is a viable production candidate. In many systems, some form of manual or exploratory testing will be required before that level of confidence can be reached. Even if the rest of the deployment pipeline is completely automated, a manual testing stage can be acceptable. It is important that it happen late enough in the deployment pipeline that team members are not wasting their time testing software that will ultimately be rejected as a viable production candidate through automated checks.

Explain that manual testing is a valid stage in the deployment pipeline and to discuss when to perform it.

4.4.5. Non-functional Testing

Load testing, stress testing, performance testing, and security testing are all examples of non-functional testing. Almost all software will require one or more of these types of testing to be considered for production deployment. Many of these tests can and should be automated. In some cases, these types of tests will not be able to automatically determine whether the software should continue in the deployment pipeline, but rather will simply provide data to team members who will have to make a judgment call.

Describe the types of non-functional test stages that might be part of a deployment pipeline and discuss how subjective decisions can be handled. It should also discuss techniques for automating the data gathering as well as the decision-making for non-functional requirements. Describe the types of non-functional test stages that might be part of a deployment pipeline and discuss how subjective decisions can be handled. It should also discuss techniques for automating the data gathering as well as the decision-making for non-functional requirements.

4.4.6. Rolling Back a Release

One of the easiest ways to boost confidence in your ability to deploy software quickly is to have a fallback position; a way to get back to a prior release that is known to work. If a process is in place to back out changes, then the risks associated with finding a problem with or after a deploy is greatly mitigated. Back out strategies include A/B or Blue/Green releases, being able to redeploy an older version, or even keeping the old deploy directory around.

Explain the benefits of having an explicit rollback process to un-deploy software if needed. Other options for dealing with a failed deploy (e.g., fixing and deploying an even newer version) should also be discussed.

4.4.7. Pushing to Production

The deployment to production is ideally handled the same as a deployment to any other environment. However, there are often additional outside influences that must be considered in production. Time of day, length of outage and impact to transactions already in flight are some of the concerns that may not be addressed in other environments even is the deployment mechanism and automation is the same.

Explain the impact of deploying to live environments with live users. It should include a discussion of different concerns and techniques and how different solutions can be integrated into the automation to reduce production impact and risk.

4.4.8. Deployment Orchestration

Even simple software projects often involve dependencies between systems. Orchestration is the term given to coordinating the different dependencies, whether the pieces have to be deployed in a certain order, or information from one system is needed in order to deploy the other one, or just that information that is needed for the deploy but not known until after the deploy is underway. It could even be that deploying one system will affect but cannot be allowed to disrupt other systems.

Explain methods for orchestrating deployments and when orchestration might be needed.

5. OPERATIONS

5.1. MANAGING INFRASTRUCTURE

5.1.1. Virtualization

With virtualization, a computer system is simulated so that multiple virtual systems can be run on one physical system. Virtualization services usually provide mechanisms for running multiple operating systems, taking and restoring point-in-time snapshots and dynamically reconfiguring system resources such as memory or hard disks. But perhaps the most important feature of virtualization for DevOps is the ability to dynamically provision a virtual system in an automated fashion, which would be impossible, or at least cost-prohibitive, to do with physical systems.

Explain the capabilities and benefits of virtualization for continuous delivery processes and DevOps.

5.1.2. Cloud Computing

Cloud computing usually refers to a group of virtual resources that are pooled together to allow sharing. Clouds can be public, such as Amazon Web Services, or private in which the resources are only shared within an organization. Entire virtual systems can be cloud enabled, as can individual applications (e.g., Salesforce) or services (e.g., cloud databases or queues).

Explain the advantages and disadvantages of cloud computing, and to explain how they apply to continuous delivery and DevOps.

5.1.3. Containers

Lightweight container technologies are growing in popularity for enabling DevOps processes. They are generally smaller and faster than their virtual machine counterparts, better enabling rapid feedback cycles and allowing for better standardization across development, test and deployment environments.

Describe the concept of containers and discuss how they differ from virtual machines.

5.1.4. Infrastructure as Code

Using automation to instantiate individual systems using virtual machines or containers is only one part of dynamic provisioning. Networking, storage, and user management are also among the activities that must be considered when creating dynamic environments, and they are often hard to define in terms of a single system.

Explain how the concept of infrastructure as code extends beyond individual systems, and how automation can be used to manage and provision infrastructure.

5.1.5. Scaling

Scaling systems includes adding systems in clusters to increase throughput and performance, as well as being able to handle periodic spikes in load. It can include separating concerns logically, such as offloading a database server to a separate system. Your automated DevOps pipeline needs to be able to work across various scaling architectures to deploy your software into the various environments. As those environments are expanded and scaled up, the automation needs to be able to support that.

Explain how different scaling architectures may impact the automation and deployment pipeline. This includes discussing how to make flexible choices early to allow for future scaling needs.

5.1.6. Monitoring

Feedback is critical throughout the deployment pipeline, but it can't stop there. Even once the software is deployed to production, it must be monitored and feedback presented to the team so that everyone can determine the health and status of the software in a given environment at any time. System health must be monitored (e.g., is the system alive, how much hard drive space is available) as well as application health (e.g., is the application running, are users using it, is a particular feature generating errors). Proper monitoring and dashboarding the results are important for both proactive and reactive planning and decisions.

Explain the role of monitoring in different environments and the types of tools and techniques that can be used.

5.1.7. Continuity Planning

One of the most important and most difficult parts of operations is handling the situation when things go wrong. Hardware can fail, networks go down, external vendors can go offline, Mother Nature can take a data center out of service.

Planning for expected and even unexpected outages and interruptions can make a significant difference in the team's ability to deal with adversity. Continuity planning includes both disaster recovery and being able to gracefully handle expected events such as switch overs, data center changes and software and hardware upgrades.

Explain the importance of continuity planning and disaster recovery planning and introduce the concept of using your DevOps automated delivery pipeline to support automated the continuity process.

5.2. MANAGING DATABASES

5.2.1. Test Data

To run meaningful tests involving a data, some sort of test data is usually necessary. Good tests are fast, independent of order and other tests failing or succeeding and can be run multiple times with repeatable results. Different types of tests may, in fact, need different sets of data. There are several good techniques for achieving each of these requirements.

Explain techniques for dealing with database tests and test data.

5.2.2. Managing Change

While application code can usually be completely replaced during each deploy, application data most likely cannot. It must be preserved from release to release, and often modified or restructured. Data modifications and migrations can and should be automated, tested, versioned, and deployed just like code changes. And just like code changes, those data changes must allow for backing out if something goes wrong.

Explain techniques for making changes to data while minimizing risk and the benefits of automating data changes through scripts.

5.2.3. Reverting Data Changes

Undoing changes to data and/or to a database can be particularly risky. Often the changes cannot simply be removed. User data may have been added to the system between an update and a desired rollback. Other data, other database tables and even other applications can be dependent on the new data that you want to undo. There are a range of approaches for handling data and database changes to allow for reverting changes.

Explain approaches for reverting data changes with minimal disruption.