Review Exercises

Official Cert Guide

Advance your IT career with hands-on learning

Cisco Certified DevNet Professional

DEVCOR 350-901

Jason Davis
Hazim Dahir
Stuart Clark
Quinn Snyder

ciscopress.com

Special Offers

Save 70% on Complete Video Course

To enhance your preparation, Cisco Press also sells Complete Video Courses for both streaming and download. Complete Video Courses provide you with hours of expert-level instruction mapped directly to exam objectives.

Save 80% on Premium Edition eBook and Practice Test

The *Cisco Certified DevNet Professional DEVCOR 350-901 Official Cert Guide Premium Edition eBook and Practice Test* provides three eBook files (PDF, EPUB, and MOBI/Kindle) to read on your preferred device and an enhanced edition of the Pearson Test Prep practice test software. You also receive two additional practice exams with links for every question mapped to the PDF eBook.

See the card insert in the back of the book for your Pearson Test Prep activation code and special offers.

the applications that directly impact anyone who uses those applications or services. These developers would work within their defined teams to prioritize the tasks that were to be accomplished within the window of time in which the work was slated. When the work was completed and tested, the resulting work product was shipped to the operations team to deploy and run.

Where development left off, operations picked up. This team (or set of teams, depending on the application and infrastructure) was responsible for keeping the application or system functional and responsive for the end customers. This team was also thought of as the front-line support for any issues that manifested, either through the deployment or through operation of the application/system at scale (both in network and in load) that wasn't feasible during testing in the development lifecycle.

You can easily see how these two teams could quickly come in conflict with one another: operations teams blaming developers as being "lazy" in their code by making assumptions or not testing the full suite of features/options/functionality and developers blaming operations for being resistant to change and always saying "no" to updates or new deployments to support the application/system. This conflict, which is very unproductive for cohesive culture within an organization, also creates problems for end users of the application or system because features are slow to be rolled out and adopted against the codebase, resulting in a less than satisfactory experience.

While the root causes for the distrust vary by organization and team, much of the friction is caused by the seemingly opposing ideas of the two teams: development is supposed to add new features, whereas operations is tasked with keeping the application or system running at all times. New features or bug fixes merged into the production codebase could have unintended knock-on effects that could require operations to troubleshoot an issue beyond their control, while development must respond to the needs of the end users to ensure a quality experience, attract new users, or prevent users from abandoning the service for something else.

You could additionally make an argument that some of the issue stems from trust between the teams. Neither team believes that the other is operating in good faith and is only focused on the metrics that they are judged on, rather than working in tandem to ensure that the ultimate metric is achieved, supporting end users throughout their journey within the application or service.

The Journey to DevOps

In 2009, the term **DevOps** was first coined by Patrick Debois. The portmanteau of *development* and *operations* was followed by the term *days* as the name of the conference that he arranged in October that year. The use of the term rocketed from there and has been used in a variety of contexts ever since. However, over the years the term has been conflated to mean a variety of things, some of which are encompassed in DevOps, some of which are tangential to, and some that are simply used incorrectly (whether intentional or not).

DevOps is rooted in the principle that while both development and operations have different roles to fill within the delivery of applications and services to end users, they are both there to enable a business to perform its end function for those customers and that the conflict that arises between these two functions creates a friction point that hinders the end goal. To efficiently create and deliver an application or service, the two teams must ensure a strong working relationship and have empathy for the critical roles that each plays in enabling the business; in short, "developers that have operational mindsets and operations teams that think like developers."

In addition to the cross-functional nature of the teams, there are several generally accepted practices for DevOps teams to implement within their environment.

- An ability to automate the infrastructure lifecycle process
- A version control system (VCS) or some other source code management (SCM) platform
- An automated build process, including testing and security scanning
- An automated delivery or deployment process, with an agreed-upon methodology for deploying the code into production prior to full rollout
- Observability and instrumentation to provide required metrics back to measure against key performance indicators
- A real-time communications system that can link human interactions with feedback from observability instrumentation

A Cultural Shift

What often gets lost in any discussion of DevOps is that an organization does not simply "buy DevOps." It is not the tools, the process, or even a job role at its core; it's a cultural change and standard method of operation that needs to be adopted within an organization. This requires an organization that embraces change and failure, is open and transparent in communication (and systems, code, and metrics), and most importantly, creates and fosters trust between organizations. Only through embracing this culture can an organization decide on the actual implementation of how DevOps will look internally, from roles and responsibilities, tooling, and operational processes. These processes work toward these goals of embracing a culture of change and agility, ensuring sources of truth for all code that all have visibility to, providing testing and validation of code prior to release, and enabling automated infrastructure changes, allowing teams to reduce errors and focus on higher-level problems associated with enabling the business. However, the tooling, team organizational and reporting structures, and even the methodologies used to release the software vary drastically from organization to organization (and sometimes between business entities within an organization). This is the flexibility (and some would say challenge) with implementing DevOps; there's no single way to perform these functions, and it can take many small changes or iterations to settle on the exact tooling and procedures for an organization or entity. As DevOps has roots in agile software development methodology, some would say this is a feature rather than a bug.

The Emergence of the Site Reliability Engineer(ing)

The concept of **site reliability engineering (SRE)** predates the creation of the term *DevOps*. Originally a concept of the Google engineering teams, the idea of the SRE was to ensure the reliability and uptime of the systems used within Google. This original team of SREs was tasked with spending approximately 50 percent of their time on operations tasks, with the other half of their time focused on product development and management duties. This approach provided a natural visibility into both the creation and design of the applications, as well as the view of the requirements of running code on large-scale production systems.

Modern-day SREs generally have backgrounds in software design and engineering, systems engineering, system administration, and network design and engineering (or some permutation of the listed competencies). These individuals act as a conduit, assisting in bridging the gaps between operations and development, while also providing leadership and vision beyond the sprint, anticipating and remediating potential issues before they appear to ensure utmost reliability. In its purest form, SRE can be practiced and performed by anyone; it needn't be a specific job title.

The distinction here, while somewhat pedantic, is important when referring to the differences in organizational concepts and processes. DevOps principles work on breaking down the walls that exist between development and operations through culture and a blending of teams and responsibilities (though everyone has their own unique strengths and weaknesses), whereas SRE looks to solve the silo problem through adding additional people with the responsibility of translation to the process. Neither solution is right or wrong, but is a by-product of the organization and business unit to which these principles are applied, the current talent of the contributing teams, and the ability to upskill and learn new processes. To use the common answer within IT, "it depends."

SRE Responsibilities and Tenets

Much like DevOps, there is no firm consensus on what exactly is in and out of scope for an SRE. The responsibilities often vary in scale and scope depending on the organization and the product or service they deliver. Additionally, SREs may be scope-limited based on their underlying knowledge or the needs of the business, creating specific SREs focused on infrastructure or networks, observability and monitoring, or focused on a specific product or service offered by the organization. These classifications and delineations can be mixed and matched to ensure the most positive outcome to the services and products being delivered to the business.

Despite there being no written rules about SRE, some generally accepted practices are mentioned most often within the definition of SRE:

- The lifecycle of the infrastructure is automated, especially anything that introduces "toil" into the build-and-deploy process.
- A rigid focus is placed on reliability above and beyond what would be considered acceptable through definition of SLAs/SLOs. This could be achieved through systems, network, or application design, or a combination of the three to achieve the desired reliability, latency, or efficiency specs that exceed objectives.
- Anything implemented as part of the infrastructure lifecycle process should include instrumentation and observability. This also should follow the same principle as reliability, in that the observability should provide information that isn't known to be needed ahead of time (that is, being ahead of the curve).

Out of these practices, you could create principles that SRE should strive for that are tangential to these tenets. Again, there is no common standard for these principles, but by combing through SRE requirements docs posted to job boards, you can see the varying responsibilities that these individuals can have, but you can draw a parallel with these responsibilities against these core tenets. Some SREs specifically focus on things like capacity planning and infrastructure design, which align with the second and third points, because planning and

design can't occur without proper understanding of the current environment's load, latency, and growth in usage over time (which requires observability tooling). Other SREs may be focused on the operational efficiencies gained through creation of CI/CD pipelines, which focuses on the energy, time, and frustration spent in moving apps from development to production in a secure manner (the "toil" mentioned previously) but doing so in a fashion that ensures reliability and uptime of the application and system. This function touches on all three principles because automated infrastructure and deployments are beneficial only when uptime is compromised and any resulting issue with the application/system is identified through detailed observability at all layers.

SRE vs. DevOps

As you look over the practices and principles of SRE, you can easily see the alignment and overlap that they have with DevOps and some of the responsibilities that fall out of that culture. This overlap is not accidental because both ideas came from similar lineage within the evolutionary path of the SDLC. However, the ways in which they achieve the overall goals differ in their approach.

Recall that DevOps is a change in culture, breaking down the traditional walls that existed between development and operations through empathy and cross-functional teams that span both sides of the application/system experience. The idea is that through cross-pollination of all members of the team, everyone views all aspects of delivering the outcome to the business with a similar lens, leading to better outcomes for all. The tooling and process that follow this organizational shift are ancillary to the overall shift.

SRE, on the other hand, does not require the outright blending of development and operations. Rather, it achieves this goal through specific individuals or teams that are tasked with overseeing the integration work needed to achieve (and exceed) the outcomes required by the business for the service or application. While culture must shift in support and recognition of the work that SRE is tasked with performing, it may not necessarily be as drastic a shift as that within DevOps.

Finally, to put a simplistic point on this, Google engineers, in their popular SRE books, state that "SRE implements DevOps," further confusing everything and providing the perspective that everything is made up and nothing really matters (at least in title). The important takeaway is that empathy and understanding between development and operations are critical to being able to deploy and maintain services at scale while ensuring changes and updates are added in a timely fashion to the codebase.

Continuous Integration/Continuous Delivery (Deployment)

In this chapter alone, **continuous integration/continuous delivery** (or deployment), also known as CI/CD, has been mentioned on several occasions and is a key piece to implement regardless of whether SRE or DevOps methodology is being followed. While *CI* and *CD* are often used together, *CI/CD* is a combination of two concepts, driven together to create an automated outcome for both development and operations teams. The "pipeline," the process that software goes through as it is put through the CI/CD process, is also only a function or sidecar to a larger **version control system (VCS)**. This VCS provides some integration with the "runner" that will perform the actual integration delivery/deployment process based on the configuration applied to a specific repository. This process runs whenever an update is added to the repository (either in the "main" or a feature branch), and the process can be completely customized to fit the requirements of the repository of code that is being worked on.

The choice of VCS and CI/CD provider is driven mostly by the organization, collaboration and security requirements, and third-party integrations with other functions (such as security and vulnerability scanning). Typical VCS include git, svn, hg, or p4 (though git is most prevalent), any of which could be hosted on-premises or through an SaaS offering. After the VCS system is selected, the desired CI/CD platform can be chosen. Some of them, such as Jenkins, are external applications that require integration into the chosen repositories, whereas others, such as GitLab-CI, integrate directly with GitLab to provide a single portal for both source code and pipelines. For the scope of this section, the choice of VCS and CI provider is largely inconsequential to the overall understanding of the functions they provide, and this section focuses on those functions.

NOTE Specific implementations of a VCS and technology (such as a git-based implementation with GitHub, GitLab, or BitBucket) are referred to as a **source code manager (SCM)**. Although *VCS* and *SCM* are sometimes used interchangeably, these two terms refer to two separate concepts.

As mentioned earlier, the full pipeline provided by a CI/CD system is two discrete functions, integration and delivery/deployment, which are interrelated but serve two purposes and divisions in labor. The following sections describe this division and the functions that occur within the two subsystems in more detail.

Continuous Integration (CI)

Continuous integration is the cornerstone of a pipeline and generally the first item to be implemented when organizations need to accelerate the speed at which code moves into production, because without the integration of code, it cannot be delivered or deployed to the end systems. As with most concepts addressed so far, what comprises the act of integration depends on the organization and the level at which different stages can be successfully automated as part of the pipeline.

Key Topic

The need for CI arises with the nature of distributed development teams working across a variety of segments of the code concurrently, using the tools within the VCS (such as support for "git") to commit, branch, and merge the respective features and fixes into the mainline codebase. Rather than a single individual writing code and compiling it against "their machine" (which could have specific dependencies or one-off configurations applied to it that make the software compile and run as expected), the CI platform creates a single platform against which all code is linted (validated for proper syntax), tested against a variety of units or other tests (testing within the code itself, rather than at the application/functionality level), scanned against security vulnerabilities, and published as a complete software package (also known as an "artifact" of the CI process). These artifacts can be packaged as releases, ready to be shipped off to a staging or production environment through manual interaction with the finished package and the end systems on which it will be deployed.

The flexibility of the CI pipeline allows for complete customization of what tests are performed where and to what level. For example, as developers work on adding new features, they may create a branch within the repository to separate this work from the main branch. At each commit to the branch, the code could be put through a series of unit tests to ensure that the code itself works at a functional level based on desired inputs and outputs. These tests provide feedback at every commit being made and may be sufficient for work within

a specific feature. However, once the code is merged from the feature branch into the main code, it may undergo additional testing from vulnerability or security scans to ensure that the entire package as written does not contain a known vulnerability, which may not have been tested adequately in performing the same security analysis against the code for a single feature. This level of customization is entirely possible within the pipeline.

Continuous Delivery: One of the CDs

The latter half of the CI/CD pipeline is the method in which the resulting integration artifact makes it from the VCS release system out to the end users. Most sources online reference CD as a monolithic item that is always the same, but subtle differences exist, depending on whether the resultant pipeline uses continuous *delivery* or continuous *deployment*. Although the two have a similar outcome, the resulting integrated software from the VCS is deployed to end users; both ideas are covered in this section for completeness. In a real-world environment, only one method would be used per code repository, though both methods could be used within a single organization or business unit, depending on release methodologies in place.

The idea behind continuous delivery is that after the resultant software is tested and built, it is "shipped," or placed in a location in which it can be manually placed on the infrastructure, often in a Q/A or test environment. This allows for some level of human control over the environment, ensuring that the supporting infrastructure and required set of test users are ready prior to allowing the application or service to "go live." This ensures focused testing across the desired range of users and situations, allowing staggered or segmented release of the resultant product in a controlled manner, as defined by the organization's methodologies or processes. When the testing phase is completed and the key results, metrics, and performance indicators are reviewed and found to be satisfactory, the code is pushed to the production environment.

Continuous Deployment: The Other CD

Continuous deployment takes the concept of continuous delivery one step further. Rather than relying on manual intervention to move the software into test and then to production, continuous deployment automates the movement into production. While the process doesn't move every node/system to the new version of the application/service at once, it assumes that end users, rather than designated test or Q/A groups, are responsible for testing the service. As you can imagine, this testing requires an understanding and readiness of the supporting infrastructure, as well as levels of observability and instrumentation to be able to identify issues with the rollout as they appear (sometimes users are not aware of or are more tolerant of less than expected performance and may not report issues or be aware of how to report them). Teams responsible for monitoring the application, infrastructure, and overall performance must also be aware of how the new version is released into production.

Like both CI and continuous *delivery*, the way in which the deployment of the committed code occurs is entirely up to the way the pipeline is constructed. In general terms, only the "main" code branch is deployed directly to production because it includes all the different components that make up the full application/service. Feature branches are still worked on independently and validated through the defined CI process for nonmain branches. When the code from the feature branch is merged into the main branch, a separate CI process, followed by the deployment process, is executed, and the new code moves into production. If code is inadvertently committed to the main branch, the pipeline should be written well enough to ensure that it catches and prevents broken code from being automatically moved