**Welcome to Introduction to DevOps**

DevOps, a culture of collaboration between Development and Operations teams. Through this collaboration, teams are able to more reliably and rapidly release high quality software to their users. What will learn:

* The stages that changes to software go through to get from a developer’s computer to its users
* The role and responsibilities of a traditional Operations team
* An overview of modern infrastructure management
* The difference between a DevOps culture and that of a traditional organization
* The purposes of key practices such as CI/CD, monitoring, and containerization
* The role of scalability, observability, and resiliency in a modern software system

**Infrastructure Management**

*Operations team is setting up a server in the server warehouse.*

In the modern age of web applications, software is stored and executed on **servers**. Servers are computers that run software that can be accessed by another device (also known as the **client**), often via the internet. Servers respond to requests with website code, images, and other content which are rendered by the client (typically a web browser).

Servers require quite a bit of maintenance in order to be used by developers. They need to be purchased, have operating systems and various software installed on them, configured to handle network requests, and much more. Typically, developers don’t manage the servers — so who does?

Traditionally, an Operations team is responsible for managing a company’s servers. More broadly, the Operations team manages all of an application’s **infrastructure**. Infrastructure is the full set of resources that support the development, testing, and deployment of applications. Infrastructure consists of:

* Hardware components such as servers, routers, switches and cables
* Software components such as operating systems, version control systems, and applications

There are dozens of tasks that fall under the responsibility of the Operations team, including:

* Installing and replacing (a.k.a “provisioning”) physical components such as servers, switches, and hard drives
* Performing software/firmware upgrades such as security patches
* Configuring infrastructure such as firewalls, user access, ports
* Monitoring network health and alerting personnel when issues arise

**Version Control Systems**

*Gabriel is excited to get their code onto their company’s servers. They sneak into the server room and pull out their USB stick. Right as they are about to plug it into the server, Lars stops him and says, “Whoa there Gabriel! That’s not how we do things around here. You should check out our version control system.”*

**Version control systems** (such as [Github](https://www.codecademy.com/learn/learn-git" \t "_blank)) are tools designed to manage different versions of a file or project. They track every change that is made to a file while saving all previous versions of the file. Some of the data that is tracked by a version control system includes:

* changed files
* new or deleted files
* renamed or moved files
* the author and date of the change

With version control (a.k.a. “source control”), the risk and impact of bugs are reduced. When a new version of the software has issues, it can be compared to previous “stable” versions to identify the error. If necessary, the software can be “rolled back” to previous versions until a fix is implemented. With the author and date of each change stored, development teams can quickly identify who has the most information about a breaking change.

Version control systems (VCS) change how teams work together. Common operations of version control, such as **branching** and **merging** enable development teams to collaborate more effectively.

* **Branching** is the process of creating a copy of the source code (the “trunk”). Developers can work on their own branches without changing the source code that real users and other developers depend on.
* **Merging** is the process of combining the changes in one branch with another. This occurs when the differences between the two branches are ready to be reconciled. When conflicts between branches arise during a merge, version control systems can assist in resolving them.

Lastly, version control systems are able to synchronize with project management tools. For example, when new code is added, an engineering manager can be alerted to review the changes.

**Instructions**

Take a look at this diagram of a typical version control system. Through the center of the diagram runs the “trunk”, the source code that all new branches are created from. As each new feature is completed, they must be merged. Sometimes, conflicts occur. Version control systems make resolving these conflicts easier.

Once code has been “checked-in” to a version control system, how does a developer know that their new code is ready to be deployed? In the next exercise, we will learn how teams maintain the quality of new code via testing.

**Testing**

*Monica is Gabriel’s engineering manager. Gabriel has checked their code into the version control system and wants users to access the new feature as soon as possible. Monica knows better. Changes need to be tested before being released to users.*

**Testing** is an essential component of the deployment process. Testing ensures new features integrate with existing features, work smoothly within the existing infrastructure, and satisfy the product and design requirements.

Different types of tests exist that are used in the various stages of deployment. Four types of tests that are often used are:

* **Unit test** — evaluates the smallest possible unit of testable code, such as a single function.
* **Integration test** — evaluates how the units of a particular program work with one another.
* **Acceptance test** — evaluates whether the user experience aligns with the business requirements of the software.
* **End-to-end test** — evaluates the application’s behavior using production-like infrastructure that includes networking, databases, and calls to external APIs.

Failures during testing help developers know that they need to update their code or increase their infrastructure resources. Success during testing gives a developer confidence that their project is in a releasable state.

For example, a developer’s workflow might incorporate testing like so:

1. Develop a new feature locally
2. Add new code to the version control system
3. The code change has tests run against it (unit, integration, sometimes acceptance or end-to-end)
4. If there are any failures, the developer will work on fixes
5. Repeat steps 1-4 until all tests pass.
6. The change is allowed to be merged.

Testing is executed throughout the various stages of the deployment process — but what exactly are those stages?

**Deployment Environments**

*Gabriel’s new feature went from their own computer, into the version control system, passed tests, and was merged into the source code. Now, it’s time to “deploy to production”! But what exactly does that mean?*

The **production environment** refers to the infrastructure that supports the complete application used by real users. This infrastructure consisted of hardware and software components scaled for real-world usage.

More broadly, an **environment** is the subset of infrastructure resources used to execute a program under specific constraints.

Along the way to the production environment, software often moves through a series of intermediate environments. Each intermediate environment allows developers to rigorously test new software without impacting production infrastructure.

Though the names of environments may differ from company to company, a common set of environments includes:

* The **local development environment** — where software is first written and tested, typically on a developer’s own computer.
* The **integration environment** — where software changes are merged using a version control system.
* The **quality assurance (QA) / testing environment** — where tests are executed to ensure the functionality and usability of each new feature.
* The **staging environment** — where the software can be performance tested in a production-like environment, but before real users are involved.
* The **production environment** — where software is accessible by real users!

These environments do not strictly represent a linear path from a developer’s computer to production. Instead, each of these environments can be viewed as a space that developers can use throughout the entire deployment process.

**Instructions**

In the image for this exercise, we can see three deployment environments: development, staging, and production. Once a new feature is created in the development environment, it is tested in intermediate environments, and finally released to users!

Using a staging environment is quite similar to using focus groups to test new products. It’s important for a company to test how a new feature will behave in real-world scenarios, without running the risk of releasing a potentially faulty product to all of its users.

*Want to learn more? Check out our full article on* [*environments*](https://www.codecademy.com/article/environments)*.*

The movement of software in and out of intermediate environments is a common source of bugs, particularly when these migrations are performed manually. In the next lesson and throughout this course, you will learn about how DevOps seeks to mitigate these issues through automation. Before we get there, continue to the final exercise to review what we have learned about deployment.

**Review**

*Gabriel’s new feature has finally been deployed to production! Users around the world are accessing the new feature via servers, networks, and other infrastructure. Their user experience is as good as ever thanks to extensive testing and version control.*

In this lesson, we learned about **deployment** — the process of making a piece of software available to its users. Deployment is a long journey that starts with a developer moving their code into a **version control system**, through a **staging environment**, and ends with them deploying their code to a **production environment**. Along the way, **unit**, **integration**, **end-to-end**, and **acceptance tests** are conducted.

This deployment process ensures that new code is shipped quickly, reliably, and with high quality. Members of the Development team and Operations team own various pieces of this process. In the next lesson on DevOps culture, you will learn how the boundaries between these two teams begin to blur in order to create a more open, and efficient, team dynamic.

**DevOps is a culture** that shifts the way that Development and Operations team members work together. DevOps culture aims to foster trust, collaboration, problem resolution, and continuous improvement across the entire team. So how can Ariadne’s team adopt a DevOps culture to increase collaboration and reduce barriers between these departments?

Teams use a variety of practices and tools in order to foster a DevOps culture. You may have even heard of some of these practices, such as **automation** or **blameless retrospectives**. Your team may already use some of these tools, such as **version control systems**. On their own, these practices and tools can certainly make improvements. When used within a DevOps culture, a team can be transformed.

**Dev vs. Ops**

A traditional software company often has a separate Development and Operations team.

* The **Development team** writes an application’s features.
* The **Operation team** creates and maintains the infrastructure that the application runs on.

In this arrangement, the Development team sends its code to the Operations team who deploys it on the infrastructure.

While each team can have clearly defined responsibilities, there is an inherent conflict between the two teams. Developers want to produce new functionality as fast as possible. Meanwhile, Operations members want the infrastructure to be stable and reliable. Unfortunately, new changes are the biggest threat to the stability of a system.

This difference in goals gets in the way of software development. Specifically, a few issues arise when an Operations team is separate from a Development team:

* Development and Operations teams own different environments in the deployment process. Differences between environments can lead to bugs that are difficult to resolve.
* Handoffs between teams take time
* Information is siloed, meaning decisions are often made without consideration of the other team.

In a traditional team structure, like the one shown in the image, Development and Operations teams work separately, resulting in frequent handoffs, delayed deployments, and more bugs.

DevOps aims to end the divide between the Development and Operations teams, changing the production dynamic for the better. In the next exercise, we’ll take a look at how teams that embrace DevOps culture combine the responsibilities of traditional Development and Operations teams.

**Dev + Ops**

Teams that foster DevOps culture seek to be highly communicative, sharing knowledge and experience across team members. By integrating Development and Operations teams, we can resolve many of the issues that arise due to conflicting goals. As a result, we have:

* Faster development and deployment cycles due to fewer handoffs & shared knowledge
* Environment consistency from Development to Staging to Production
* Improvement of operations activities by applying dev best-practices like version control

A typical engineering team within a DevOps culture may include various engineers, quality assurance (QA) testers, security operations, and information technology (IT) specialists. Rather than siloing information, these team members can share responsibilities, align on team objectives, and make decisions together.

Through this collaboration, teams can produce better software in less time and more reliably than ever before.

**Combining Development and Operations team members onto one team provides a number of benefits — but what challenges might teams face when making this transition?**

Each team will face their own challenges when trying to combine Development and Operations. Here are just a few examples:

* When teams are used to having complete ownership over responsibilities, they may resist sharing them.
* Adopting DevOps practices may take time to get used to. Later in this course, we’ll learn about some of the practices that teams make use of.
* Establishing agreed-upon processes can be challenging when team members have different experiences. Rather than having these differences drag a team down, teams with a positive DevOps culture should utilize these differences to make more informed decisions

Tearing down boundaries between Development and Operations is just one way that teams within a DevOps culture like to throw away the old in favor of the new.

**DevOps Culture**

Development and Operations is the way to go. Excited to adopt DevOps, Ariadne gets to work learning about the key components of a positive DevOps culture.

The culture of DevOps is the most critical factor to its success. Collaboration cannot occur from only applying a set of practices and tools. It requires a culture in which collaboration can thrive.

The central pillars of a DevOps culture include (click on each item to learn more!):

**Systems-level thinking**

*Systems-level thinking means thinking about the whole production system, rather than a single department. Doing so allows teams to identify* ***bottlenecks****.*

**Continuous experimentation and learning**

*Teams that embrace continuous experimentation and learning encourage rapid development of new features and accept failure as a learning opportunity.*

**Feedback loops**

*Feedback loops allow teams to draw information from each part of the system. As they gain insight into their systems, processes can be improved and optimized.*

To have a successful DevOps culture within an organization, these components must be implemented at both a team and individual level. Only with a positive culture can the practices and tools be utilized to their fullest.

who are often busy or have a backlog of tasks to handle. Ariadne sees that having only one set of team-members be able to control deployments is a bottleneck that slows down production. Which of the three pillars of DevOps culture is Ariadne practicing?

Ariadne is practicing systems-level thinking!

The central pillars of DevOps culture include systems-level thinking, continuous experimentation and learning, and feedback loops. Over the next few exercises, we’ll take a closer look at each of these pillars, starting with systems-level thinking.

**Systems-Level Thinking**

Within a traditional team, team members often focus on their own tasks rather than the big picture. Developers might concentrate on the code without considering infrastructure or testing needs. The Operations team might only consider infrastructure needs without considering impacts on functionality.

DevOps seeks to have each team member consider all aspects of the development process. This practice is known as **systems-level thinking**. Though this may sound daunting, the shared-responsibility team structure of DevOps makes this possible. When a team has developers, IT specialists, QA testers, and security experts, information is shared and decisions are made as a team. Meanwhile, individuals are given the opportunity to grow and gain knowledge in new domains.

One important outcome of systems-level thinking is the identification and resolution of **bottlenecks**. A bottleneck is a system’s slowest point, resulting in the slowing down of the entire process. A Development team with a systems-level view can more easily see where these slowdowns occur, resolve systemic issues, and optimize processes.

**Instructions**

Resolving bottlenecks requires a team to acknowledge the issues within the system. Teams practicing DevOps view these shortcomings as opportunities for continuous improvement rather than as a concerning failure. In the next exercise, we’ll look at how learning from mistakes and experimentation are essential to DevOps.

*Image (2022) retrieved from* [*Integrify*](https://www.integrify.com/bottleneck-software/)

**Learning from Failure**

*The team is called into their manager’s office once again and Thomas is nervous. His team has been losing people and is struggling to keep up with demand. The user response to the latest feature he spearheaded has not been positive. Everyone in the room is silent, waiting to see who their manager will be blaming this time. A few minutes of yelling and finger-pointing later, Thomas is packing his things, the latest member to be fired.*

Organizations that punish failure create a culture of fear that gets in the way of innovation and growth. DevOps seeks to change this by viewing failure as a natural part of everyday work. Rather than having mistakes be punished, DevOps culture stresses the importance of failure as a learning opportunity.

Failure itself is never something that teams seek out. However, once failure is normalized, teams can be more open to experimentation. Teams can try out ambitious solutions, fail quickly, learn from those mistakes, and then experiment again. This cycle enables teams to regularly make product and process improvements, both big and small.

One method DevOps uses to normalize failure and learn from experimentation is through **blameless retrospectives** (or “post-mortems”). These retrospectives are meetings held at the end of a sprint, project, or issue resolution. Here, team members discuss what went well and the areas where they can improve.

**Instructions**

What are some of the consequences of Thomas getting fired from his team after his failed product launch? If Thomas’ team were practicing DevOps culture, how might they have responded differently to the failed product launch?

Others may view Thomas’ failure, and subsequent removal from the team, as a warning sign that failure is not tolerated. They may be less likely to propose ambitious product features and growth at the company may stall out.

If his team had been following DevOps culture, they may have instead held a blameless retrospective to learn from the failure and continue to release ambitious features.

When practicing retrospectives, teams should use data to identify areas of improvement. In the next exercise, we’ll look at how DevOps teams incorporate metrics into their processes to inform their decision-making.

**Feedback Loops**

*Ariadne’s project manager Yuri regularly tracks the average time it takes for various tasks. She notices that since the start of the new year her developers are taking an unusually long time to respond to user-submitted bug reports. She digs deeper and discovers that the automation tool her team had been using for years to assign bug reports to developers has been shut down! She quickly pivots and finds a new tool for her team to use.*

In this situation, Yuri created a **feedback loop**. A feedback loop is created when a team identifies and tracks a key piece of data, or **metric**. Teams can then use that information to drive process improvements. A typical feedback loop may look like this:

* The system is monitored and data is collected.
* Data is analyzed and bottlenecks are identified.
* Solutions are created and implemented.
* New solutions are monitored again.

Choosing which metrics to track is perhaps the most important step in creating a feedback loop. Metrics that can’t provide meaningful insight into a system just distract from the real bottlenecks.

Focusing on metrics that affect the customer is a great place to start. Some of these include:

* Time to load a website page
* Time to resolve an issue/outage
* Time to release new features

When and where metrics are tracked is another important aspect to consider. A defect, or problem, becomes more expensive to fix as it moves along the development process. DevOps seeks to discover defects as early as possible, a strategy known as **shifting left**.

When we collect and utilize feedback loops, we are also embracing, rather than turning away from, the inefficiencies in our system. Feedback loops contribute directly to a culture of continuous learning.

**Instructions**

**What was the key metric that Yuri tracked to create her feedback loop?**

Answer

The metric that Yuri tracked was the time it took to respond to bug reports.

Well done! We’ve seen quite a few scenarios where a DevOps culture could have helped. We also learned the key practices teams use to foster a DevOps culture including systems-level thinking, an embrace of failure and learning, and feedback loops. Now, before you run off to change the way *your* organization operates, continue on to the next and final exercise to review what we’ve learned.

**Review**

DevOps culture centers around collaboration between team members across all domains of software development. Teams practicing DevOps can likely include developers, IT specialists, QA testers, and security experts. Rather than siloing information, teams can share knowledge and make informed decisions together, also known as **“building quality in”**.

In order to foster a DevOps culture, teams can make use of key DevOps practices:

* **Systems-level thinking** — thinking of the whole system to identify **bottlenecks**
* **Continuous experimentation and learning** — embracing failure through practices such as **blameless retrospectives**
* **Feedback loops** — using **metrics** and **shifting left** to drive process improvement

By fostering DevOps culture and following DevOps practices, teams can build and release software with greater speed, quality, and reliability than ever before. Throughout the rest of this course, you will learn about some of the specific tools that teams use to implement this culture and its practices.