**Introduction to Infrastructure Management**

*Summer launches her web browser, enters the URL for her company’s business analytics application, and voilà — immediately, her browser is filled with beautiful charts, graphs, and statistics loaded from millions of data points.*

*Meanwhile, Kai navigates to an e-commerce application and struggles to purchase a pair of shoes. Half of the product images in his search results fail to load, and when he finally manages to add an item and click the checkout button, nothing happens.*

What could cause such drastically different experiences when using web applications? Often, the answer lies in how businesses manage their infrastructure.

Recall that infrastructure is the set of hardware and software components used to develop, test, and deploy applications. Managing this infrastructure involves quite a number of tasks.

* Hardware needs to be installed and maintained.
* Thought must be given to power and cooling.
* Networks and databases need to be configured.
* Hardware failures and cyber-attacks are serious concerns with any infrastructure.

Dealing with all of these things is a tall task — even for seasoned professionals! While there are various types of infrastructure and methodologies for managing it, with DevOps, certain practices have become the norm.

In this lesson, we’ll explore the ways that infrastructure has traditionally been managed, as well as DevOps practices that address the common issues with traditional infrastructure management. The main topics we will cover are:

* Scaling Infrastructure
* In-house Infrastructure
* Virtualization
* Containerization
* Infrastructure as Code
* Orchestration
* Cloud Infrastructure

Before we dive in, let’s get a feel for the importance of properly managing infrastructure.

**Instructions**

1. Take a look at the web application here. Every user of the application is represented by a colored ball that bounces around the screen. Pay attention to the speed of the balls and the overall experience of the app in its default state.
2. Play with the slider included in the app. It simulates adding more users to the application. What is the experience as more users are added? Does the application maintain the same performance?

In this exercise, we saw the symptoms of an application with poorly managed infrastructure. Adding more users caused the speed of the application to slow dramatically. In the next exercise, we’ll explore a key concept related to combating these kinds of experiences — **scalability**.

**Scaling Infrastructure**

**Scalability** is a system’s ability to add resources to keep up with growing demand. When more more users begin using an application, infrastructure with great scalability will handle it without interrupting services. An infrastructure with poor scalability will likely cause slowdowns or disruptions.

##### **Vertical & Horizontal Scaling**

In practice, scalability can be achieved either through vertical scaling or horizontal scaling. **Vertical scaling** means adding computing resources, such as increasing network speeds, storage, or RAM. **Horizontal scaling** means adding more servers (or “nodes”) that each run the application. A tool called a “load balancer” then distributes the work across the many servers.

Vertical scaling is relatively simple and affordable, as it only involves upgrading a machine. However, there is some downtime required to perform the upgrade.

Horizontal scaling has the benefit of not requiring any downtime for existing servers. This benefit is the main reason why this is the scaling option chosen by most DevOps teams. That being said, it is slightly more complex to manage the many servers, and it is more expensive than upgrading existing machines.

##### **The Price of Scaling**

If scalability is so important, why not just run an application on a million powerful servers? Surely this would be enough to keep up with plenty of growth? Probably, but companies still need to consider the cost of their infrastructure. Scaling is about finding that sweet spot — enough to perform well but not so much that money is wasted.

This leads to another important goal when it comes to infrastructure — **elasticity**. Whereas scalability only deals with increases in resources, elasticity is the ability to automatically add or subtract resources to accommodate fluctuating demand. Elasticity is especially important when using pay-per-use infrastructure services since resources can be returned, and money can be saved, when demand shrinks.

### Instructions

Here we have the same bouncing ball application from the previous exercise. Each time a user is added to the application (and a new ball appears), an additional server is used. Play with the slider to simulate adding more users.

How does the experience differ from the last version of our app? Does it offer better performance? And which type of scaling does this application use?

DevOps uses several techniques to achieve both elasticity and scalability. Automation, cloud-based infrastructure, and [microservices](https://www.codecademy.com/content-items/c2a72d7debd1ed82435cac67da8932b1#:~:text=as%20microservices%20architecture.-,Microservices%20Architecture,-Microservices%20architecture%20refers) are some of these practices. In the next exercise, we’ll explore an important technology that leads to more scalable infrastructure.

Material: <https://www.codecademy.com/learn/introduction-to-dev-ops/modules/introduction-to-dev-ops-culture-and-processes/cheatsheet>

**Virtualization**

Let’s imagine we have a very simple blog application. It doesn’t require much of our server’s memory or processing power. However, it is getting a ton of traffic, which is great! We need to scale the application to keep up with growing demand.

We want to avoid any downtime to our servers so we won’t scale vertically. Scaling horizontally will enable us to handle the increase in traffic without interruptions!

Horizontal scaling should work well for performance, but it isn’t very efficient. Servers are expensive, and our application uses only a small portion of each server’s physical resources. This will result in a lot of unused physical capacity.

To solve this problem, we could turn to **virtualization**. Virtualization technology allows many virtual machines (VMs) to run on one physical computer. Each virtual machine is a distinct environment with its own operating system, dependencies, and users.

Virtualization would reduce the number of servers needed to run many instances of our blog application. Each server can be utilized closer to its full capacity by being split into several VMs. This allows us to horizontally scale even more efficiently based on demand.

In addition to more efficient resource utilization, virtualization brings convenience as well. Virtualization management tools simplify the task of creating virtual machines. Using these tools is more efficient than installing and managing pieces of hardware by hand.

**Instructions**

Take a look at the image provided which shows horizontal scaling with virtualization in action. *Without* virtualization each server can only run one process, resulting in a lot of unused server resources. *With* virtualization, a single server can run multiple processes and achieve a higher usage percentage.

In the next lesson, we’ll learn about a successor of virtualization that allows for even more efficient scaling.

**Containerization**

Dakota has tested all of the features of her application in her local development environment. She is proud to push the app to their team’s testing environment for QA to look at. Within hours, QA is reporting that none of the pages are loading. She looks at their server logs and sees cryptic error messages coming from a specific Linux package that is out of date. “Hmmm”, she wonders. “Why didn’t I see this in my local environment?”

This type of story is all too common in software development. An application or feature works in one environment but not another. Often, this is caused by differences in **dependencies** – external files and programs that are not a part of the application but are used by it.

DevOps relies upon **containerization** to solve this issue. Containerization is a form of virtualization in which users create virtual environments called **containers**. Like virtual machines (VMs), containers include instances of applications as well as their dependencies. This makes them a convenient solution to help applications behave consistently when moving through the deployment pipeline.

But couldn’t we just use VMs to also solve this problem?

Unlike VMs, containers do not include their own operating system. Instead, they share the operating system of the host machine. The lack of their own operating system makes containers smaller and faster to spin up than VMs. It would be inconvenient to re-create a virtual machine whenever changes to an application or its dependencies are made. Containers, on the other hand, can be started in only seconds.

Additionally, since containers do not need their own operating system, they use less physical resources than virtual machines. This allows many more containers to be run than VMs, leading to even more efficient scaling.

The technology for containerization has been around for decades. However, it did not become widespread until 2013 with the release of [Docker](https://www.docker.com/). Docker provided a simple interface for developers to create and run containers. Today, there are a handful of containerization tools that are used in addition to Docker.

### Instructions

1. Take a look at the GIF for this exercise labeled “Without Containerization”. We see an application moving through different environments in a deployment pipeline. Each environment has a virtual machine that runs our application. However, each step needs to be paused for dependencies to be upgraded.
2. Now, take a look at the GIF labeled “With Containerization”. Our application can move seamlessly between the different stages in our pipeline. It takes only seconds to deploy the container, and we are guaranteed to have a consistent environment in which it runs.

In the next lesson, we’ll look at how virtualization and containerization enable automation of time-consuming practices.

**Orchestration**

Juan has been in the software business for a long time. He oversees application deployment for a prestigious banking institution. The team is about to roll out a major update to their application. It will require getting thousands of production servers to run the new version. He thinks back to the old days and laughs. He would have had to log in to each server, one by one, and redeploy the application with the changes. Today, there is a far better way…

Juan will likely turn to orchestration software to assist in deploying the update to their application. According to [RedHat](https://www.redhat.com/en/topics/automation/what-is-orchestration), “**Orchestration** is the automated configuration, management, and coordination of infrastructure.”

Much like a conductor to an orchestra, orchestration tools direct many individual components on how to play their part in order to achieve consistency across the entire infrastructure system.

Tools like Docker give the ability to create and control individual containers. Orchestration software, such as [Kubernetes](https://kubernetes.io/), controls many containers working together in harmony. Once the desired infrastructure configuration for a system has been defined, Kubernetes makes sure that new containers are deployed based on that configuration. To do this, Kubernetes automatically performs tasks such as:

* Deploying containers across many servers
* Restarting failed containers
* Rolling out updates without any downtime
* Horizontal scaling of containerized applications

### Instructions

1. Take a look at the image for this exercise showing a GIF of Kubernetes in action. How does this GIF embody the concept of orchestration?

**Orchestration**

Juan has been in the software business for a long time. He oversees application deployment for a prestigious banking institution. The team is about to roll out a major update to their application. It will require getting thousands of production servers to run the new version. He thinks back to the old days and laughs. He would have had to log in to each server, one by one, and redeploy the application with the changes. Today, there is a far better way…

Juan will likely turn to orchestration software to assist in deploying the update to their application. According to [RedHat](https://www.redhat.com/en/topics/automation/what-is-orchestration), “**Orchestration** is the automated configuration, management, and coordination of infrastructure.”

Much like a conductor to an orchestra, orchestration tools direct many individual components on how to play their part in order to achieve consistency across the entire infrastructure system.

Tools like Docker give the ability to create and control individual containers. Orchestration software, such as [Kubernetes](https://kubernetes.io/), controls many containers working together in harmony. Once the desired infrastructure configuration for a system has been defined, Kubernetes makes sure that new containers are deployed based on that configuration. To do this, Kubernetes automatically performs tasks such as:

* Deploying containers across many servers
* Restarting failed containers
* Rolling out updates without any downtime
* Horizontal scaling of containerized applications

### Instructions

1. Take a look at the image for this exercise showing a GIF of Kubernetes in action. How does this GIF embody the concept of orchestration?

You may be wondering, how can an orchestration tool like Kubernetes automatically configure new containers? In the next lesson, we’ll look at a couple of common principles that orchestration tools are built on, starting with **Infrastructure as Code** (IaC).

**Infrastructure as Code**

Orchestration tools rely on a core principle of DevOps: **Infrastructure as Code**. Infrastructure as Code (IaC) is the act of defining infrastructure properties in configuration files. These configuration files are then stored and tracked in version control systems.

Infrastructure configuration used to be performed manually. Individual team members would install operating systems, upgrade dependencies, and configure networks. This process was both time-consuming and error-prone. Over time, configurations across servers would become inconsistent due to human error. This build-up of errors is known as “configuration drift”.

IaC solves this problem by relying on configuration files as the source of truth for infrastructure state. Since configuration files are machine readable, it enables easier automation. It is also simple to roll back changes if needed since these files are stored in version control systems.

IaC/orchestration tools also rely on the idea of *immutable infrastructure*. Immutable infrastructure means that servers are not changed once they are created. Instead, when infrastructure configurations change, new servers are created and old ones are destroyed. Immutable infrastructure and the use of configuration files guarantees that all servers are created in the same manner and eliminates the risk of configuration drift.

Immutable infrastructure would not have been practical in the age of physical servers, since the cost and time to set up physical servers is high. Operations teams had to rely on updating servers once they were deployed. Virtual machines, however, can be destroyed and created in minutes with little cost. With containers, it might only take seconds.

IaC and immutable infrastructure ultimately lead to lower business costs and a better user experience. These principles demonstrate how DevOps applies development principles to operations.

**Instructions**

Take a look at the image to the right, showing a GIF of how IaC tools can create new infrastructure resources. How does this GIF embody the concepts of Infrastructure as Code and immutable infrastructure?

In exercises thus far, we’ve learned a lot about techniques for configuring infrastructure. The next two exercises will focus on two strategies for physically managing infrastructure – **in-house** and **cloud-based** infrastructure.

**In-House Infrastructure**

Historically, businesses owned and managed infrastructure on company premises with their staff. This is known as **traditional-** or **in-house infrastructure**. With traditional infrastructure, the company acquires, configures, and maintains physical infrastructure components themselves. These components include servers, power supplies, and cooling.

Due to the complexity of infrastructure components and how they interact, this can be a tall order. Some of the problems that companies run into when managing their infrastructure include:

* Hardware components such as power supplies, hard drives, and RAM fail over time.
* Malicious users attempt to disrupt web services and steal sensitive data.
* Software becomes outdated, requiring consistent patches and upgrades.
* Scaling up infrastructure as demands grow.

Despite the many challenges that traditional infrastructure management faces, it offers unparalleled customization over its resources. While more modern solutions exist, many companies still use traditional infrastructure.

**Instructions**

Can you think of any reasons why a modern business might still rely on traditional infrastructure?

Getting ahead of the problems of in-house infrastructure comes with heavy costs: staff hours, equipment purchases, and power. In the next exercise, we’ll take a look at how a cloud-based infrastructure mitigates many of these challenges.