**Learning objectives**

By the end of this module, you're able to:

* Provision and configure target environment.
* Deploy to an environment securely using a service connection.
* Configure functional test automation and run availability tests.
* Setup test infrastructure.

**Provision and configure target environments**

The release pipeline deploys software to a target environment. But it isn't only the software that will be deployed with the release pipeline.

If you focus on Continuous Delivery, Infrastructure as Code and spinning up Infrastructure as part of your release pipeline is essential.

When we focus on the deployment of the infrastructure, we should first consider the differences between the target environments that we can deploy to:

* On-Premises servers.
* Cloud servers or Infrastructure as a Service (IaaS). For example, Virtual machines or networks.
* Platform as a Service (PaaS) and Functions as a Service (FaaS). For example, Azure SQL Database in both PaaS and serverless options.
* Clusters.
* Service Connections.

Let us dive a bit further into these different target environments and connections.

**On-premises servers**

In most cases, when you deploy to an on-premises server, the hardware and the operating system are already in place. The server is already there and ready.

In some cases, empty, but most of the time not. In this case, the release pipeline can only focus on deploying the application.

You might want to start or stop a virtual machine (Hyper-V or VMware).

The scripts you use to start or stop the on-premises servers should be part of your source control and delivered to your release pipeline as a build artifact.

Using a task in the release pipeline, you can run the script that starts or stops the servers.

To take it one step further and configure the server, you should look at technologies like PowerShell Desired State Configuration(DSC).

The product will maintain your server and keep it in a particular state. When the server changes its state, you can recover the changed configuration to the original configuration.

Integrating a tool like PowerShell DSC into the release pipeline is no different from any other task you add.

**Infrastructure as a service**

When you use the cloud as your target environment, things change slightly. Some organizations lift and shift from their on-premises servers to cloud servers.

Then your deployment works the same as an on-premises server. But when you use the cloud to provide you with Infrastructure as a Service (IaaS), you can use the power of the cloud to start and create servers when needed.

It's where Infrastructure as Code (IaC) starts playing a significant role.

Creating a script or template can make a server or other infrastructural components like a SQL server, a network, or an IP address.

By defining a template or using a command line and saving it in a script file, you can use that file in your release pipeline tasks to execute it on your target cloud.

The server (or another component) will be created as part of your pipeline. After that, you can run the steps to deploy the software.

Technologies like Azure Resource Manager are great for creating Infrastructure on demand.

**Platform as a Service**

When you move from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS), you'll get the infrastructure from the cloud you're running on.

For example: In Azure, you can create a Web application. The cloud arranges the server, the hardware, the network, the public IP address, the storage account, and even the web server.

The user only needs to take care of the web application on this Platform.

You only need to provide the templates instructing the cloud to create a WebApp. Functions as a Service (FaaS) or Serverless technologies are the same.

In Azure, it's called Azure Functions. You only deploy your application, and the cloud takes care of the rest. However, you must instruct the Platform (the cloud) to create a placeholder where your application can be hosted.

You can define this template in Azure Resource Manager. You can use the Azure CLI or command-line tools.

In all cases, the infrastructure is defined in a script file and lives alongside the application code in source control.

**Clusters**

Finally, you can deploy your software to a cluster. A cluster is a group of servers that host high-scale applications.

When you run an Infrastructure as a Service cluster, you must create and maintain the cluster. It means that you need to provide the templates to create a cluster.

You must also ensure you roll out updates, bug fixes, and patches to your cluster. It's comparable with Infrastructure as a Service.

When you use a hosted cluster, you should consider it a Platform as a Service. You instruct the cloud to create the cluster, and you deploy your software to the cluster.

When you run a container cluster, you can use the container cluster technologies like AKS.

**Service connections**

When a pipeline needs resource access, you must often create service connections.

**Summary**

Whatever the technology you choose to host your application, your Infrastructure's creation or configuration should be part of your release pipeline and source control repository.

Infrastructure as Code is a fundamental part of Continuous Delivery, allowing you to create servers and environments on demand.

**Configure automated integration and functional test automation**

Completed 100 XP

* 6 minutes

The first thing that comes to mind about Continuous Delivery is that everything needs to be automated.

Otherwise, you can't deploy multiple times a day. But how to deal with testing, then?

Many companies still have a broad suite of manual tests to be run before delivering to production. Somehow these tests need to run every time a new release is created.

Instead of automating all your manual tests into automated UI tests, you need to rethink your testing strategy.

Lisa Crispin describes in her book Agile Testing that you can divide your tests into multiple categories.

  
Source: [https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants](https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/)

We can make four quadrants where each side of the square defines our targets with our tests.

* Business facing - the tests are more functional and often executed by end users of the system or by specialized testers that know the problem domain well.
* Supporting the Team - it helps a development team get constant feedback on the product to find bugs quickly and deliver a quality build-in product.
* Technology facing - the tests are rather technical and non-meaningful to business people. They're typical tests written and executed by the developers in a development team.
* Critique Product - tests that validate a product's workings on its functional and non-functional requirements.

Now we can place different test types we see in the other quadrants. For example, we can put Unit tests, Component tests, and System or integration tests in the first quadrant.

We can place functional tests, Story tests, prototypes, and simulations in quadrant two. These tests support the team in delivering the correct functionality and are business-facing since they're more functional.

In quadrant three, we can place tests like exploratory, usability, acceptance, etc.

We place performance, load, security, and other non-functional requirements tests in quadrant four.

Looking at these quadrants, specific tests are easy to automate or automated by nature. These tests are in quadrants 1 and 4. Tests that are automatable but mostly not automated by nature are the tests in quadrant 2. Tests that are the hardest to automate are in quadrant 3.

We also see that the tests that can't be automated or are hard to automate are tests that can be executed in an earlier phase and not after release.

We call shift-left, where we move the testing process towards the development cycle.

We need to automate as many tests as possible and test them.

A few of the principles we can use are:

* Tests should be written at the lowest level possible.
* Write once, and run anywhere, including the production system.
* The product is designed for testability.
* Test code is product code; only reliable tests survive.
* Test ownership follows product ownership.

By testing at the lowest level possible, you'll find many tests that don't require infrastructure or applications to be deployed.

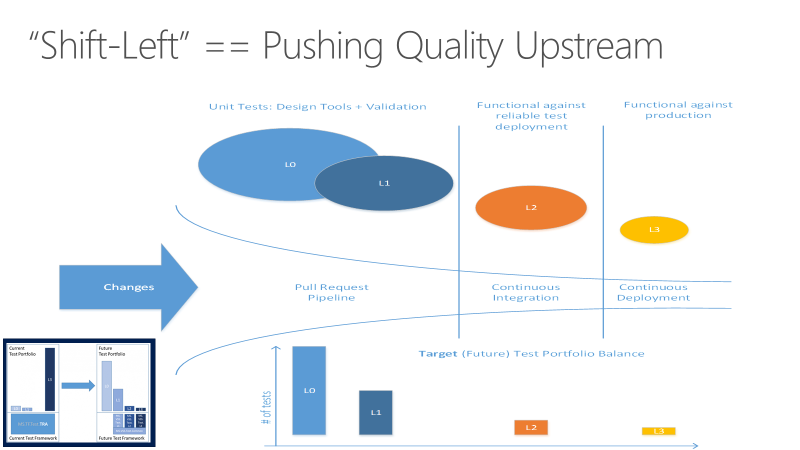
We can use the pipeline to execute the tests that need an app or infrastructure. We can run scripts or use specific test tools to perform tests within the pipeline.

On many occasions, you execute these external tools from the pipeline, like Owasp ZAP, SpecFlow, or Selenium.

**Understand Shift-left**

The goal for shifting left is to move quality upstream by performing tests early in the pipeline. It represents the phrase "fail fast, fail often" combining test and process improvements reduces the time it takes for tests to be run and the impact of failures later on.

The idea is to ensure that most of the testing is complete before merging a change into the main branch.

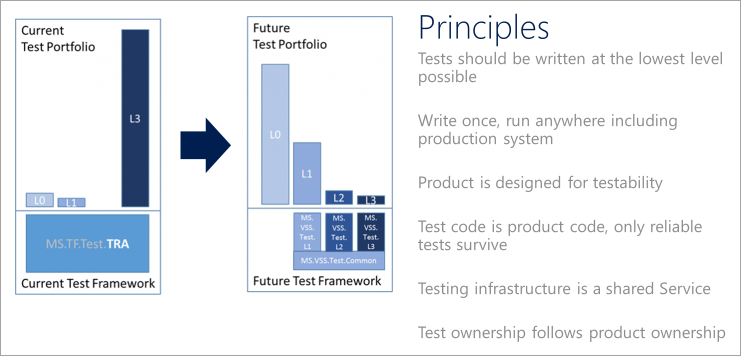


Many teams find their test takes too long to run during the development lifecycle.

As projects scale, the number and nature of tests will grow substantially, taking hours or days to run the complete test.

They get pushed further until they're run at the last possible moment, and the benefits intended to be gained from building those tests aren't realized until long after the code has been committed.

There are several essential principles that DevOps teams should adhere to in implementing any quality vision.



Other important characteristics to take into consideration:

* **Unit tests:** These tests need to be fast and reliable.
  + One team at Microsoft runs over 60,000 unit tests in parallel in less than 6 minutes, intending to get down to less than a minute.
* **Functional tests:** Must be independent.
* **Defining a test taxonomy** is an essential aspect of DevOps. The developers should understand the suitable types of tests in different scenarios.
  + **L0** tests are a broad class of fast in-memory unit tests. It's a test that depends on code in the assembly under test and nothing else.
  + **L1** tests might require assembly plus SQL or the file system.
  + **L2** tests are functional tests run against testable service deployments. It's a functional test category requiring a service deployment but may have critical service dependencies stubbed out.
  + **L3** tests are a restricted class of integration tests that run against production. They require a complete product deployment.

Check the case study in shifting left at Microsoft: [Shift left to make testing fast and reliable](https://learn.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable).

For more information, see:

* [Shift right to test in production](https://learn.microsoft.com/en-us/devops/deliver/shift-right-test-production).

You can use test functionality from a platform like Azure on other occasions. For example, Availability or Load Tests executed from within the cloud platform.

When you want to write your automated tests, choose the language that resembles the language from your code.

In most cases, the application developers should also write the test, so it makes sense to use the same language. For example, write tests for your .NET application in .NET and your Angular application in Angular.

The build and release agent can handle it to execute Unit Tests or other low-level tests that don't need a deployed application or infrastructure.

When you need to do tests with a UI or other specialized functionality, you need a Test agent to run the test and report the results. Installation of the test agent then needs to be done upfront or as part of the execution of your pipeline.