### **Delivering Microservices**

In the traditional world of monolithic applications, it’s not uncommon for deployments to be a manual process often where you follow a step-by-step word document that explains how to upgrade to a new version of your software.

But, once you moved to microservices, a manual process is not going to work anymore, You’ve got far more things to deploy, and you also want to deploy much more frequently. And so microservices pushes us very strongly in the direction of automating our deployment.

If we have a reliable and repeatable deployment process for our microservices, that allows us to upgrade them frequently as soon as new features or bug fixes become available.

If fact, some teams uses microservices deploy multiple times a day. So, how can we achieve this?

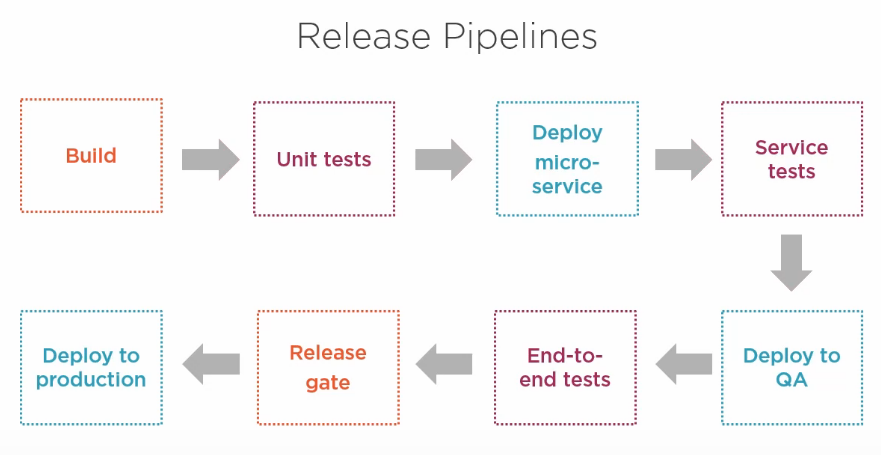
Well, a continuous Integration server can be used to automatically build our code whenever we check it in. And it’s also very common for that CI server to run unit tests as well.

The CI Server can also trigger a release pipeline. A release pipeline starts off where the CI build has finished, It builds the code and run the unit tests. But, next, we want the microservice to get deployed. May be to a virtual machine running or to a cloud deployment resources.

Once the Microservice is deployed, this is a great opportunity to run the service level integration tests, where we test a microservice in isolation. If it passes those test, the next step might be to deploy the microservice into a QA environment, where it can be tested as part of the entire system.

And this is a point where you’d run any automated end-to-end tests. Now at this point there’s often what’s called as a release gate. Just because our microservice has successfully passed all of its automated tests, doesn’t necessarily mean that we’re ready to push it straight to production.

There may be the need for some additional manual testing or some kind of risk assessment testing or customer sign off before, finally, we’re ready for this code to go to Production. Once you reach the state of being ready to push a microservice into production, it’s really important that you use exactly the same procedure that you used to deliver it to the QA or staging environment for pushing it to production.



**Deployment Environments**

A developer might have their own virtual machine that they want to run a microservice, which will allow them to debug their code locally.

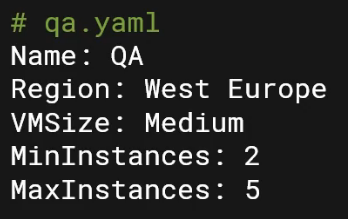
You might have a QA environment in which you’re going to run your end to end tests or perform some manual testing.

Sometimes you might have other environments dedicated to penetration testing or performance testing.

And of course you will have a production environment. And in some scenarios you’ll have a multiple production environments for few geographical regions.

We define each environment by using configuration file, and JSON or YAML are very popular choices here.

For example:



Configuration file like these are often used in conjunction with other deployments scripts and manifests to allow you to deploy a specific version of a microservice into a specific environment.

**Independent Upgrades**

We don’t want microservices to be tightly coupled in such a way that upgrading a single microservices requires all the others to be upgraded at the same time. And that means our automated process needs to be able to update just one microservice at a time, even if it can deploy them all at once.

So, its very important to think about how you’re going to upgrade an individual microservice? You might think that it’s a simple case of just stoping the old one and starting the new one, and that’s certainly a valid approach. However it does cause a small period of service unavailability. And depending on what that microservice is responsible for, that may or may not be acceptable. But there are number of other techniques that you can use to reduce downtime when you’re upgrading a service.

One is called **Blue Green Swap,** where you run the old and new version of the service simultaneously and then swap traffic one to other by means of load balancer. And that means you don’t have to have downtime while you’re waiting for the new version of the service to start up.

Another option, if you’ve multiple replicas of a service running, is to replace the instances one by one. So, for example, if you have got 3 instances of Version 1, all load balance, you might, one at a time, add new instances of version 2 and remove instances of version 1 until all are upgraded.

**Monitoring Microservices**

Your microservices might be running dozens of different processes across many different host machines, and you don’t want to have to individually connect to every worker node in your virtual machine cluster and examine individual log files for every microservice. Instead, what we want is a system where all of our monitoring, telemetry, and logs are available in a centralized place. Ideally we want a dashboard that gives us an instant view of the overall health of the system and the ability to dive in deeper if a problem is detected.

Let’s see what we should monitor:

1- We want to capture the host metrics. This include things like CPU and memory usage. By tracking these metrics, we can detect if we need to scale out to meet increased demand. Many cloud providers offers the ability to set up alerts based on these sort of metrics.

2- We want to track metrics at the application level. So for a web API, this would be tracking the number of HTTP requests, including the number of failures, and tracking what the errors code were. For example, if there were lots of 401 Unauthorized response codes, then maybe our system is under attack by hackers, or maybe we’ve misconfigured something. If there’s lots of 500 errors, then that points to a bug somewhere in our code.

If you’re using a message broker, then you’ll want to track whether large numbers of messages are backing up in the queues, in which case, again we might need to scale out. And you’ll also want to track whether messages are being sent to a dead-letter queue, which indicates that you’ve got a problem processing your messages.

It’s better to have a health checkup point for each microservices. This API can be periodically called to check you service is running and functioning correctly.

3- We need the logs to be easily accessible. Each microservice should be emitting logs, and if those logs are aggregated into a centralized place, that will make your life a whole lot easier. Containers usually have a standardize approach to capturing logs.