**AWS microservice and serverless with containers**

AWS provides different solutions in order to deploy applications using containers.

Although it’s quite easy to start a journey with containers, finding the best solution for handling them can challenging.

In this document we analyse the pros and the cons of using different tools in AWS to deploy containers.

## AWS Elastic Beanstalk

AWS Elastic Beanstalk (EB) is a layer of abstraction on the AWS Elastic Container Service that adds some convenient wrapping around the Elastic Container Services’ features.

Amazon Elastic Container Service (ECS) is a highly scalable, high-performance [container](https://aws.amazon.com/containers/) orchestration service that supports [Docker](https://aws.amazon.com/docker/) containers and allows you to easily run and scale containerized applications on AWS. Amazon ECS eliminates the need for you to install and operate your own container orchestration software, manage and scale a cluster of virtual machines, or schedule containers on those virtual machines.

ECS has a higher learning curve than Elastic Beanstalk, which we can quickly deploy and manage applications in the AWS Cloud without worrying about the infrastructure that runs those applications.

AWS Elastic Beanstalk reduces management complexity without restricting choice or control.

We can simply upload our application, and Elastic Beanstalk automatically handles the details of capacity provisioning, load balancing, scaling, and application health monitoring.

In a nutshell it provides:

* Automatically provisioning infrastructure including EC2 servers, load balancers, security groups, auto scaling groups, CName
* Manages environments (dev, staging, prod, etc)
* Manages code deployment across multiple servers including multiple update strategies

**Platforms supported**

A container defines the infrastructure and software stack to be used for a given environment.

When we deploy our application, Elastic Beanstalk provisions one or more AWS resources, such as Amazon EC2 instances. The software stack that runs on your Amazon EC2 instances depends on the container type.

For example we can upload out Java application (.war file) to Elastic Beanstalk, and define some information about the application within a beanstalk configuration file.

**branch-defaults:  
 default:  
 environment:** dev   
 **group\_suffix:** dev  
 **master:  
 environment:** dev  
 **group\_suffix:** dev  
**deploy:  
 artifact:** target/spring-boot-aws-elasticbeanstalk-0.0.1-SNAPSHOT.jar  
**global:  
 application\_name:** spring-boot-aws-elasticbeanstalk   
 **default\_ec2\_keyname:** aws  
 **default\_platform:** Java 8  
 **default\_region:** us-east-1  
 **profile:** eb-cli

Elastic Beanstalk automatically launches an environment and creates and configures the AWS resources needed to run your code.

After your environment is launched, you can then manage your environment and deploy new application versions.

Elastic Beanstalk provides platforms for programming languages (Java, PHP, Python, Ruby, Go), web containers (Tomcat, Passenger, Puma), and Docker containers, with multiple configurations of each.

Elastic Beanstalk also supports custom platforms that can be based on an AMI that you create and can include further customizations.

Custom platforms are region specific. If you use Elastic Beanstalk in multiple regions, you must create your platforms separately in each region.

## Elastick beanstalk Deployment flow

Elastic Beanstalk’s default deployment process consists of the following stages :

1. Build artifact locally or through a CI to
2. upload code to AWS
3. compile code
4. deploy artifacts to EC2 servers

For Java applications for example we can choose to deploy as standalone Java application, tomcat or docker.

For spring boot application for example we can use the Java option. As in other PaaS like PCF we can just add some config into a yml file and EBS will do everything for us.

## Elastic Beanstalk maven plugin

Even though the AWS CLI can do everything we need, there are legitimate reason to use the [beanstalker](http://beanstalker.ingenieux.com.br/beanstalk-maven-plugin/usage.html) maven plugin instead :

* Keep all Elastic Beanstalk configuration in one place, the pom.xml
* No need to install the CLI
* Out of the box integration with any CI system that can run maven

<**plugin**>  
 <**groupId**>br.com.ingenieux</**groupId**>  
 <**artifactId**>beanstalk-maven-plugin</**artifactId**>  
 <**version**>1.5.0</**version**>  
 <**configuration**>  
 <**applicationName**>spring-boot-aws-elasticbeanstalk </**applicationName**>  
 <**s3Bucket**>szp-spring-boot-aws</**s3Bucket**>  
 <**s3Key**>${project.artifactId}/${project.build.finalName}.jar</**s3Key**>  
 <**cnamePrefix**>spring-boot-aws-elasticbeanstalk-dev</**cnamePrefix**>  
 <**environmentName**>dev</**environmentName**>  
 <**environmentRef**>dev</**environmentRef**>  
 <**solutionStack**>64bit Amazon Linux 2016.09 v2.4.0 running Java 8</**solutionStack**>  
 </**configuration**>  
</**plugin**>

**Elastic Beanstalk load balancer types**

* [Classic Load Balancer](http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/) – The Elastic Load Balancing previous-generation load balancer. Routes HTTP, HTTPS, or TCP request traffic to different ports on environment instances.
* [Application Load Balancer](http://docs.aws.amazon.com/elasticloadbalancing/latest/application/) – An application layer load balancer. Routes HTTP or HTTPS request traffic to different ports on environment instances based on the request path.
* [Network Load Balancer](http://docs.aws.amazon.com/elasticloadbalancing/latest/network/) – A network layer load balancer. Routes TCP request traffic to different ports on environment instances. Supports both active and passive health checks.

#### **Orchestrating Containers**

In microservice architectures a system is a collection of containers running on hosts in some environment, connected to clients and each other.

What we want is to make building systems as reliable and repeatable as building containers.

Cluster management software helps us in achieving the above.

A “cluster” is a group of hosts optimized for running containers.

Cluster management software aims to make defining and provisioning clusters of container hosts as easy as possible.

At that higher level of abstraction what we need is the ability to define which images are launched into a cluster, what the environment is for each, how they relate to each other and the outside world over the network, etc.

Lat year Amazon announced general availability of Elastic Container Service, a container orchestration offering built on their platform: EC2.

Kubernetes was designed from the ground up to be a provider-independent, functionally complete container orchestration platform.

**Elastic beanstalk and Docker container**

Docker is a container platform that allows to run programs in *containers*, isolated from the rest of a system leveraging Linux control groups. Running applications this way leads to better composability, organization, and helps build immutable infrastructure that’s more robust and easier to manage.

The following docker file tells docker to create a container with Java8 installed and then run the spring boot java application.

**FROM** anapsix**/**alpine-java:8\_server-jre\_unlimited  
**ADD** target**/**spring-boot-aws-elasticbeanstalk-0.0.1**-**SNAPSHOT.jar app.jar  
**RUN** sh **-**c **'touch /app.jar'  
ENV *JAVA\_OPTS***=**""  
ENTRYPOINT** [ **"sh"**, **"-c"**, **"java** $***JAVA\_OPTS* -Djava.security.egd=file:/dev/./urandom -jar /app.jar"** ]

The Docker platform for Elastic Beanstalk has two generic configurations (single container and multicontainer), and several preconfigured containers.

**Elastic beanstalk Docker Single container**

The Single Container Docker platform is against the Docker philosophy practices by forcing you to run your entire application from a single container which simply is not using Docker in the way it was intended.

Using single container we could run into problems when we have multiple machines, lots of containers that may need to communicate with one another, need to reliabilty update containers, and load balance traffic among them.

For this, you need a layer of orchestration that can choose which machines to schedule containers on based on certain criteria (capacity, memory availability, CPU load, etc.), perform rolling deploys of containers while keeping a certain number running, manage network access between containers, allow containers to discover one another, and so on.

**Elastic beanstalk Docker Multi container**

Elastic Beanstalk (multi-container) is an abstraction layer on top of [ECS (Elastic Container Service)](https://aws.amazon.com/ecs/) with some bootstrapped features and some limitations:

* Automatically interacts with ECS and [ELB](https://aws.amazon.com/elasticloadbalancing/)
* Cluster health and metrics are readily available and displayed without any extra effort
* Load balancer must terminate HTTPS and all backend connections are HTTP
* Easily adjustable autoscaling and instance sizing
* Manageable environment variables
* Container logs are all collected in one place, but still segmented by instance – so in a cluster environment finding which instance served a request that logged some important data is a challenge
* Can *only set hard memory limits* in container definitions
* All cluster instances must run the same set of containers

When you’re just getting started with Docker or container technology and your application is young, this can be a compelling solution. Docker images can still be pulled from public or private registries and with some exceptions, running containers is fairly consistent with other platforms. That means there’s very little vendor lock-in using Elastic Beanstalk to get things rolling.

More troublesome however, is the lack of a soft memory limit that’s supported by ECS but strangely not by Elastic Beanstalk. The problem this creates is that unless you know precisely how much memory your containers will use, you’re forced to either:

1) Err on the side of caution and give your containers more memory than they need, reducing the number of containers you can schedule onto an instance (your hard memory limits can’t exceed the memory for the instance type) or forcing you to upgrade your instance type to one with more memory

or

2) Tweak the memory for each container in order to fit all the containers your app needs on a single instance (remember, each instance runs the same containers) and when a container hits the limit, be prepared for problems

**ECS (ELASTIC CONTAINER SERVICE)**

ECS is Amazon’s answer to container orchestration.

All of the limitations imposed by Elastic Beanstalk are lifted.

ECS does have a straightforward approach to declaratively defining containers which allows you to easily compose several “container definitions” within a “task definition”.

An ECS *task* is the primary unit of container orchestration that you use to launch containers. Each task includes one or more container definitions, allowing you to group containers together into a logical set that’s launched and terminated at the same time.

AWS is releasing soon access to AWS EKS (Elastic Container Service for Kubernetes).

If we’re looking for a simple containers service to use on production, we can go with AWS EB, but when you have to cater for the whole infrastructure, different projects, changing requirements and overall variability, probably Kubernetes is the way to go.

**Infrastructure as code**

The main tool to deploy infrastructure in AWS is CloudFormation.

CloudFormation is an AWS tool for deploying infrastructure. You describe your desired infrastructure in YAML or JSON, then submit your CloudFormation template for deployment. It enables "infrastructure as code".

On Top of CloudFormation there are several abastract layers that come to help in deploying servfull and serverless architectures on AWS.

In this document we will give an overview to the Serverless framework and Terraform.

Mixed serverless framework and terraform are not bad idea at all. In fact, with current functions in terraform, you have to mix it with serverless framework to deal with API Gateway.

**Serverless framework**

The Serverless Framework provides a configuration DSL which is designed for serverless applications. It also enables infrastructure as code while removing a lot of the boilerplate required for deploying serverless applications, including permissions, event subscriptions, logging, etc.

When deploying to AWS, the Serverless Framework is using CloudFormation under the hood. This means you can use the Serverless Framework's easy syntax to describe most of your Serverless Application while still having the ability to supplement with standard CloudFormation if needed.

The Serverless Framework is provider-agnostic, so you can use it to deploy serverless applications to AWS, Microsoft Azure, Google Cloud Platform, or many other providers. This reduces lock-in and enables a multi-cloud strategy while giving you a consistent experience across clouds.

Finally, the Serverless Framework assists with additional aspects of the serverless application lifecycle, including building your function package, invoking your functions for testing, and viewing your application logs.

The [Serverless](https://medium.com/@serverlessinc) Framework help us orchestrate all of our AWS Lambda functions.

How it works

**The serverless framework allows us to** define all of your functions and the events that trigger them in a serverless.yml file.

We can also provision additional infrastructure resources in these files, such as DynamoDB table, SQS queues, SNS topics, S3 buckets, etc. This gives you nice "Infrastructure as Code" so we can deploy our whole service with a single command -- *serverless deploy*.

**Terraform**

Terraform is an unopinionated cloud deployment tool. It describes Infrastructure as Code and deploys to multiple clouds and SaaS systems at once. It is comparable to CloudFormation but for multiple clouds.

Compare with the Serveless framework, The Serverless Platform has one strong opinion about how an application is defined, and then is flexible about everything else. It facilitates developing and deploying Serverless Applications, abstracting away the boilerplate required to deploy serverless applications. It also assists with the packaging and monitoring of your serverless applications.

Terraform and the Serverless Platform are not mutually exclusive and can easily be used in tandem, however in Terraform fits more in servful architecture while the Serverless framework is a perfect firt for serverless architectures.

Terraform is good for certain things, but it is not recommend for managing your Lambdas.

It only solves the configuration aspect, so we'll have to write your own code-management scripts and then figure out how to integrate them.

Second, there's a ton of boilerplate in deploying Lambdas + setting up API Gateway. This includes setting up the proper IAM roles, configuring Cloudwatch log groups, setting up all the API Gateway stuff, etc. The other frameworks do the heavy lifting of abstracting this away from us, but we'll have to do it ourself with Terraform.

Terraform can be handy to handle more permanent aspects of their infrastructure, like databases, VPCs, security groups, etc., while using the Serverless Framework for the Lambda parts.

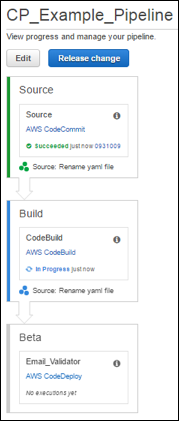
# **AWS CodePipeline vs. Jenkins CI Server**

In a world of shortened release cycles, agile development practices and continuous integration (CI), it is imperative teams have access to a stable tool which can [manage the process](http://www.sumologic.com/devops/understand-build-continuous-delivery-pipeline/) of building, testing and deploying projects as changes are made. For many years, Jenkins has been the tool of choice among Java developers, but as many projects have moved into the cloud, and in particular, Amazon’s [AWS ecosystem](http://www.sumologic.com/aws/), other tools are gaining traction. One such tool is [AWS CodePipeline](http://www.sumologic.com/devops/devops-as-a-service/create-pipeline-with-aws-codepipeline/).

Whether you’ve been operating in a CI environment for years, or you’re new to the concept, it’s important to know what options are available and what features and advantages one tool has over another. This article will look at Jenkins and AWS CodePipeline in-depth and present reasons why you might select one over the other, or why you might consider using a combination of the two to achieve your CI goals.

## What Is AWS CodePipeline and What Does It Do?

Introduced by Amazon in 2015, AWS CodePipeline is a service which allows the user to configure a CI workflow or pipeline within the AWS ecosystem. Using either the Amazon Command Line Interface (CLI), or clean UI configuration process within the Amazon Console, the user can define a process which includes sourcing, building and deploying an application or service.



*On screen display of an AWS CodePipeline running after being triggered by a code change.*

While it is an AWS service, and one that is configurable with other AWS services like CodeCommit, CodeBuild and CodeDeploy, the pipeline can also be configured to use third-party services like GitHub, and even Jenkins itself.

## CopePipeline vs Jenkins: Initial Setup, Cost, and Maintenance

The biggest difference by far between Jenkins and CodePipeline is in the offering itself. Jenkins is an open source solution which can be downloaded and installed free of charge. (Well, free unless you factor in the cost of the physical server, or EC2 instance that you need to provision in order to host it, and the time required to install and maintain it.)

AWS CodePipeline is a service available to any user of the AWS ecosystem. Within minutes of making the decision to try it out, you can be configuring your CI pipeline in the cloud. As with other AWS services, you don’t have to be concerned with infrastructure provisioning or maintenance. And at the time of this writing, a CI Pipeline can be had for just one dollar a month.

CodePipeline is customizable, maintenance is taken care of by Amazon, and you’re not going to be fielding Jenkins support questions from your development team about updates, compatible plugins and why that one project’s build keeps failing.

## CopePipeline vs Jenkins: Customization and Integration with Existing Services

Part of what makes Jenkins so versatile is the plugin architecture. If there is a specific type of technology you’d like to include in your CI process, there is probably a plugin out there that can do the job, and on the off chance that there isn’t a plugin, either you or someone with a little development experience could probably write one.

AWS CodePipeline is configurable for the use of multiple tools, and while the options aren’t limited to just AWS service offerings, it is a rather limited list. However, if you’ve committed to using AWS as your cloud provider, or you use tools such as GitHub as a source code repository, CodePipeline may have all the options you need.

CodePipeline supports:

* Source Code Repositories
  + Amazon S3 (Versioned)
  + AWS CodeCommit
  + GitHub
* Build Providers
  + AWS CodeBuild
  + Jenkins
  + Solano CI
* Deployment Tools
  + AWS Elastic Beanstalk
  + AWS CodeDeploy
  + AWS CloudFormation

Additionally, if CodePipeline doesn’t provide support for a particular process, it is possible to add custom actions. Unfortunately, these actions must be added through the Amazon CLI, and cannot be added through the UI in the Amazon console. Additionally, AWS Lambdas can be included in the pipeline to extend functionality. You can find out more about these Advanced Tasks in the AWS CodePipeline documentation.

Choosing the Right Tool for Your DevOps Use Case

As the guy on my team who has been handling automation stuff for the past couple of years, I’m rather fond of Jenkins. Jenkins has served me well, and generally does what I ask. That aside, with its versatility and ability to be customized, there is a reason I’m the go-to guy on the team. Jenkins is not necessarily difficult to use, but it can be complex at times.

AWS CodePipeline is very clean, and integrates exceptionally well with projects in the Amazon ecosystem. With Jenkins, however, security is controlled on the server itself. With CodePipeline, it falls nicely under the same IAM controls as other AWS services. And, as I mentioned above, someone familiar with AWS, but new to CodePipeline could have a pipeline configured and running within a couple of hours.

For teams and organizations already using Jenkins, perhaps now might not be the time to switch, but CodePipeline is definitely worth taking a look at. The more astute of you may also have noticed that Jenkins is a build provider option within CodePipeline, and there’s the option to keep your build logic in Jenkins, while at the same time integrating it into a new CodePipeline pipeline. Jenkins and CodePipeline can work well together, and need not be considered mutually exclusive.

If you’re new to AWS and new to CI, then selecting CodePipeline is definitely something you and your team should consider. You can learn more about CodePipeline from the [AWS CodePipeline User Guide](http://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html).

The following diagram shows an example Elastic Beanstalk environment configured with three Docker containers running on each Amazon EC2 instance in an Auto Scaling group:

Elastic Beanstalk (EB) is a PaaS solution in AWS family and it provides very high level concepts: you have applications, versions and you create environments.

EC2 Container service (ECS) is a very low level cluster scheduling platform. You have to manually describe a lot of configuration for your Docker containers, link them and also manually setup load balancers and everything else you need.

So, EB is much simpler to use and maintain. ECS is more complicated, but it uses your resources in a very efficient way.

Also, EB has two different Docker types: single-container and multi-container. Multi-container uses ECS internally.

My advice: use Elastic Beanstalk. ECS is a good fit if you have big number of different applications that you need to run efficiently in a cluster.

## Google Kubernetes Engine

GKE requires a more complex approach, but it results in some specific advantages you can notice migrating from AWS EB.

Google Kubernetes Engine takes advantage of [Kubernetes itself](https://kubernetes.io/), “an open-source system for automating deployment, scaling, and management of containerized applications”.

It’s not a framework, it’s not a library, it’s a whole platform. It can be used almost everywhere and it’s completely infrastructure independent. You can use it natively as Google Kubernetes Engine (former Google Container Engine), utilize “kubeadm” provided to set up the cluster on bare-metals or “kops”, which is the solution for AWS at the moment. ( AWS is currently working on EKS, a native solution similar to GKE, it’s, however, in a closed preview phase when this post is being written.) You can even set up everything by hand. It will work as long as there’s a network connection between instances involved in the cluster.

### **Managing environments and configuration**

Kubernetes itself is not working on the instances level as AWS EB does. Instances create a cluster Kubernetes operates on, but the smallest entity is Pod, representing some tightly coupled containers. Going up in the hierarchy of objects, there’s a ReplicaSet consisting of Pods and responsible for holding a current configuration of Pods and their scaling policy. ReplicaSets can be standalone objects, but currently a Deployment object creates ReplicaSets with every update of configuration. Then it can manage the scaling, perform rolling updates and offer simple rollbacking.

Everything is isolated in the cluster by default. If there’s a need for exposing anything, it’s done with Service object type having 3 different types: ClusterIP (exposed on the cluster level internally), NodePort (exposed on specific port number on every node of cluster), and finally – LoadBalancer type, which stands for a real Load Balancer if the cluster is backed on the cloud. It’s worth noting that with Kubernetes being deployed on AWS with kops it’s a bit more complicated to set up a Load Balancer expose type than in GKE and it was one of the advantages we took into consideration when choosing the cloud provider for Kubernetes stack.