Building Continuous Delivery pipelines for Kubernetes with AWS native tooling

This write up demonstrates how to spin up an automated CI/CD pipeline for Kubernetes deployments by a working Wordpress deployment example, using AWS native tools and services. The solution is built by AWS managed services meaning zero maintenance efforts for all Developer Tools. For launching Kubernetes stacks, we have built our example clusters based on our github codebase for launching Kubernetes clusters on AWS.

AWS resources for this tutorial can be launched by Cloudformation templates. You can launch these templates in your own AWS account, therefore you will be charged for the Kubernetes cluster and CI/CD pipeline related resource costs.

This tutorial was written for developers, IT architects, administrators, and DevOps professionals who are planning to implement their production Kubernetes workloads on AWS. By reading this paper, you should have a high level understanding of AWS DevOps services and you can experiment with these services yourself by the working Wordpress example.

Overview

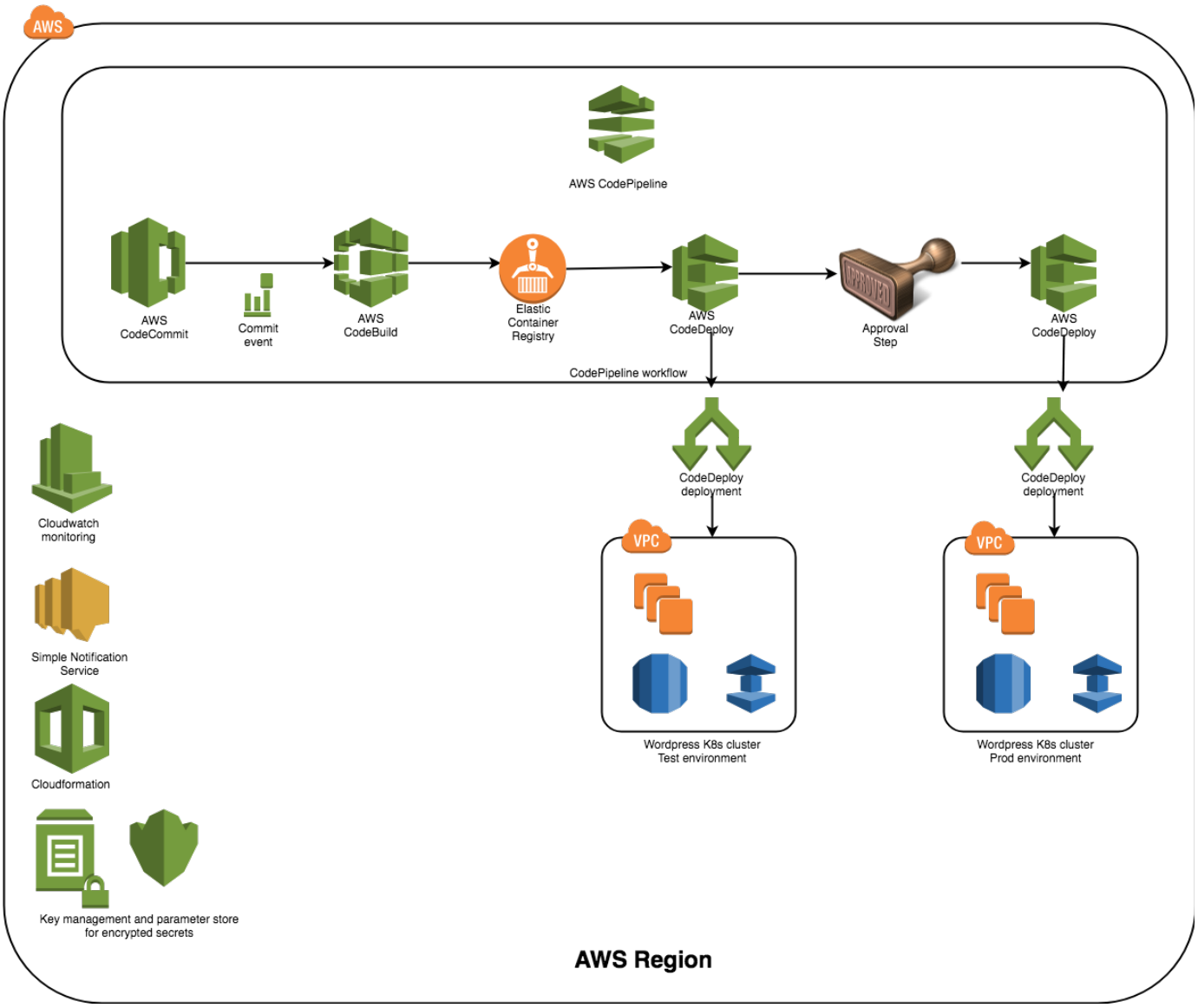
This paper will explain the components and for each their main benefits when creating an end to end AWS CI/CD pipeline for Kubernetes deployments. The pipeline will be implemented by Codepipeline, CodeCommit, Elastic Container Registry, CodeBuild and Codedeploy services, and by using additional services like SSM Parameter Store and KMS to securely store parameters. For deploying to Kubernetes, we have built on our existing K8s CFN stack to deploy a cluster with RDS Mysql and ElastiCache to host multiple Wordpress environments (Test and Production in the tutorial).

TC2 Github repository

Architecture

CI/CD pipeline

For Continuous Integration and Deployment, we have built a pipeline from AWS services only. On the diagram below, you can see the high level steps and components used for the pipeline. All the services in use are charged based on actual usage, so costs are calculated based on API calls, storage, per resource (e.g. per user for CodeCommit or per pipeline for CodePipeline) and of course per execution time (e.g. CodeBuild build time). This means costs scale linearly with usage. Assuming you have sensible source code and deployment artifact (Docker image) sizes, if no deployments done for a period of time, your costs should be nearly 0 or very little.



The deployment steps are fairly standard, but each of the components are rich in configuration options, so can serve well for many different use cases.

We keep our Wordpress sources in the CodeCommit Git repository. Once a commit happens on the master branch, there will be a Cloudwatch event generated, which triggers the second step: building our sources with CodeBuild. We will execute our application inside a Docker container, so at this stage we build a Docker image with the latest Wordpress changes from the Git repository, and push it to the ECR private Docker registry. Additionally, scripts and templates for the Kubernetes deployment are prepared and passed to the following steps.

Next, the pipeline will proceed to CodeDeploy step. CodeDeploy uses and agent for deployments, which is installed on one instance in this case in both Test and Production Wordpress environments. The deployable artifact consists of the recently built Docker image from ECR, and additional scripts and deployment templates for Kubernetes, received from the CodeBuild step. For deploying an application, there's a natural need for passing secrets and other parameters for the application configuration. This time it's MySQL hostname, user and password are required to configure the Wordpress stacks. These parameters and secrets are stored in AWS Simple Systems Manager (SSM) Parameter Store. Parameter Store is also integrated with AWS Key Management Service (KMS), which can be used to encrypt secrets in SSM using AWS or self provided encryption keys. If the deployment fails, CodeDeploy is capable of performing an automatic rollback, which restores the latest successful deployment on the node.

In case the deployment was successful, the pipeline will wait for a human approval to continue. This is often used before rolling out a deployment to production. To notify a group of people who can approve, Simple Notification Service can send out messages on various channels (email, SMS, custom webhook or application e.g. to Slack). If this step gets approved, the pipeline will proceed to the production deployment step, using the Production SSM parameters this time.

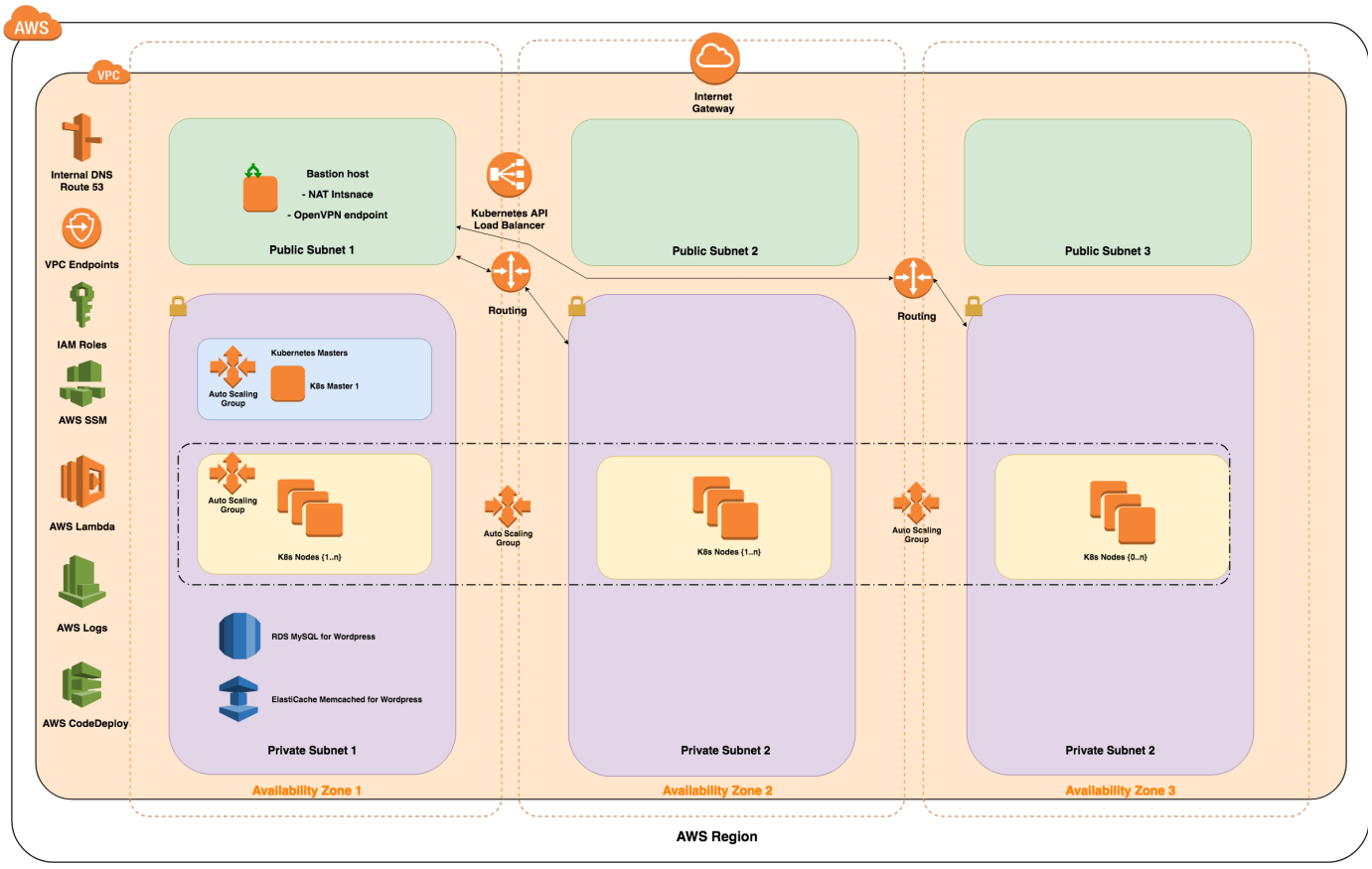
Codepipeline is responsbile for controlling the overall pipeline and workflow: passing input and output parameters between steps, managing pipeline execution threads and so on.

Logs are aggregated for the overall solution in Cloudwatch logs. The pipeline and underlying components are deployed via

Cloudformation templates.

Kubernetes cluster environment for Wordpress

Below is the architecture diagram for one Wordpress environment, so you will need to launch two of these to satisfy the above CI/CD deployment pipeline architecture. The Wordpress Kubernetes cluster demo environment was developed from our Kubernetes dev environment sample template @TC2, with some minor additions: there's an additional RDS Mysql instance (single AZ) and an ElastiCache memcached cluster (single instance) for storing Wordpress sessions and data. The bastion host has CodeDeploy agent installed by default, so it comes ready for deployment as a result of the Cloudformation stack.



For this demo scenario, two K8s nodes will be sufficient for Wordpress deployments. The deployment happens by invoking kubectl deployment templates on the Bastion host (by CodeDeploy, automatically), which by design is allowed to manage the K8s cluster.

CI/CD components

Let's explore the components of the solution one by one and their capabilities.

Docker container and wordpress sources

Wordpress sources are placed in the sources/ folder in the repository. Any development work can be done on these files which can be commited back to the Git repository. In our tutorial, the deployment pipeline will be configured to wait for commits to the Master branch of the Git repo.

Docker container sources live in the build/ directory, because these are used to build new container images. There is a Dockerfile and a docker-entrypoint.sh for this purpose.

Have a look in the Dockerfile. There are various environment variables defined at the beginning, mostly to specify parameters of the Wordpress installation. These parameters can be defined here, and can be overriden in the Kubernetes deployment specification files, which you will see later on. The rest of the file contains library and package installations for Wordpress. Note line 80 and 82 in the Dockerfile, where we copy the entrypoint and the sources directory to the Docker container before building and image so that it contains the latest changes.

The docker-entrypoint.sh file is mostly used to set Wordpress runtime specific configuration, so when the container starts up, it will be

executed. In this section the database and memcached related settings are set.

Source code management - Codecommit

CodeCommit is a standard Git repository, managed by AWS. The repository for this tutorial will be created as part of the Cloudformation stack.

There are various events and triggers you can set for your CodeCommit repository, but in this case CodePipeline will handle everything for us to keep track and handle Codecommit events.

Build tool - CodeBuild

CodeBuild provider a managed service for building source code into artifacts. Although in the case of Wordpress, PHP does not need to be compiled, our Build phase prepares the Docker image and uploads it to ECR. CodeBuild can provide various Build environments for different programming languages, and plays nicely with CodePipeline to pass parameters to the following stages (e.g.: CodeDeploy).

In this tutorial, we have placed the buildspec.yml build specification file under build/ directory. The build process in this case will build the Dockerfile, copy in the latest changes from sources/ directory as explained above, and it will also prepare the deployment template for Kubernetes (template under build/k8s-deployment-template.yml). Two important things in the buildspec.yml file are the parameters in the beginning: you have to specify the Docker repository and image name for the container images, and also the version number which will be applied when tagging the image.

The other thing is the artifacts definition at the end of the spec file. This specifies what will be the output artifact of the build process. The deploy folder will have a copy of the k8s-deployment-template.yml Kubernetes deployment file, with specific values to the desired latest Docker image properties. The deploy folder will also contain additional scripts for deploying to K8s. At the end of the build process, all files will be packaged up together (with paths discarded) as the output of this step.

Docker Artifact Repository - ECR

Elastic Container Registry is part of Elastic Container Service, and ships a standard private Docker image repository. Our images will land here built by CodeBuild.

CodeDeploy

CodeDeploy is an AWS managed service, which (surprisingly) lets you deploy artifacts to various targets: you can deploy to EC2 machines, no matter if it's manually launched, or launched in an autoscaling group, it's also able to deploy to on premise instances. Besides EC2, you can also deploy by Lambda functions, which in practise enables you to deploy any kind of target with a little bit of coding.

In this case we are using CodeDeploy to deploy to our EC2 bastion hosts, launched by Cloudformation. These bastion hosts for the environments are authorised to launch kubectl, the kubernetes command line admin tool, and are empowered to deploy new versions of containers to the K8s cluster. The CodeDeploy targets are selected by EC2 key=value tags in this case, so you can easily filter on your instances by environment, role etc. depending on your tagging strategy.

CodeDeploy have several events of the deployment called hooks. You can execute your own scripts in different phases of the lifecycle of the deployment process - e.g. taking out a node from the loadbalancer, stopping the application, installing the scripts on the node, validating the deployment and so on. Have a look on the appspec.yml file in the deployment/ folder. You will see we have 4 files needed on the target node: 3 scripts which are part of the deployment and the deployment specification file.

We execute a cleanup before installing the deployment files, to make sure the working directory is clean.

Then we execute the deployment in Kubernetes. Looking at k8s_deploy.sh file in the deploy/ folder, you can see it is retrieving parameters from AWS Simple Systems Manager's Parameter Store service. These parameters can be encrypted and protected by AWS Key Management Service if needed (in case of secrets). We use the provided environment variables by CodeDeploy to find out which Deployment Group we are deploying to, which specifies the cluster environment itself. Using a simple conditional in the code can specify whether we need to use parameters for Test or Production environment in this case.

To make sure the deployed containers in the Kubernetes pods are running and healthy, we execute some simple checks against the deployment status and the pods status if they are running.

In case the validation script fails, there's an option in CodeDeploy to rollback to the last successful deployment. In this case this is not enabled because of the way Kubernetes handles deployments, it will not put unhealthy pods into service, but the CodeDeploy rollback feature can work well in many scenarios.

Kubernetes deployment

Our Kubernetes Deployment file consists of 3 resources:

- a Deployment resource for Wordpress: it will launch or replace 3 pods marked with 'wordpress' label with the specified container

image version. We pass in the database and cache parameters as environment vars from the deployment file, which overrides the settings specified in the Dockerfile. The deployment strategy is left as default on Rolling Update, so new pods are launched and replace old ones gradually.
- a Service created in Kubernetes. This is a NodePort type service, which is required for using AWS native loadbalancers (Application Loadbalancer)
- an Ingress created in Kubernetes. By creating this, Kubernetes will launch or update an ALB resource in AWS, with the corresponding port forwarding and health checks.

CodePipeline

CodePipeline can wrap all AWS CI/CD services together (and even supports a lot more external services for Build and Testing), and controls the workflow.

In our case, there's a Source stage, which is waiting for a trigger event from Codecommit - a commit to the master branch. Next, it will proceed to the Build phase, where CodeBuild will build a new Docker image with the latest source code changes, upload it to ECR and prepares the deployment templates. As the next stage is Deploy to Test, it will invoke CodeDeploy to deploy the new container image to Test environment. Once the deployment has finished successfully to Test environment, there is an approve stage. The approve action has an SNS topic it can publish to, so all subscribed users by any subscription channel can get notified if there's a Deploy to Production approval waiting. In our case we are using email channel to notify our approvers. You can choose to approve or reject to the next stage. Deployment to Production is similar to deployment to Test, but in this case our deployment script will apply different - production parameters.

One thing to know about Codepipeline is that it will execute a pipeline for each of the commits, and will keep the parameters and state for the overall process (as you would expect). As an example, if you have 3 commits to CodeCommit in a short period of time, it will kick off 3 deployment in order, and you will see the Approval stage waiting to deploy the first commited changes to Production, then the next and lastly the most recent one, all in order.

If you get some time to have a play with CodePipeline, you will see you can build a lot more complex stages in your pipeline than this very simple sequential workflow. You can choose to have sequential and paralel tasks within one stage. There are also many testing and deployment options besides CodeCommit. A commonly used pattern is to invoke Cloudformation as a step in Codepipeline, to build an environment for UI/stress testing, and just terminate it once it steps over the testing stage.

AWS tooling vs. traditional build/deploy tools

I imagine many DevOps professionals who get to this point in reading and will say - okay, I can get this working, but why would I change from my good old Jenkins/Bamboo/GoCD/whatever deploy tools? So what could be the reasons, to switch to the AWS provided solution over running your own 3rd party CI/CD tools?

- these are managed services: this is a quite obvious one - being a managed service, AWS will make sure these services are up and running and available for you all the time. By taking advantage of the multiple AZs in AWS regions, that would take serious efforts to keep the same level of availability and robustness with 3rd parties running on virtual machines
- AWS services are billed based on actual usage: that's another nice aspect of using managed (and serverless) services: if you have a look at your build tool or version control system running on EC2, how much of its capacity is actually being used, and how long is it sitting idle in most of the days, it's clear that paying for consumed storage and execution time is way cheaper in almost all cases.
- AWS services are tightly integrated with each other: the best thing about AWS, that all the services have embedded monitoring, logging, user management and so on. Most of the cases, it takes hard work and additional costs, to make sure everything is logged, monitored and users are managed in the right way for your CI/CD tools.
- Going further down on the line of integration, from the security perspective, AWS offers enterprise grade security solutions, also integrated with all of their services. Audit logging and user management is taken care for you. In our CI/CD demo, there are no passwords involved when communicating between services, instead IAM roles are being used. These roles authorise specific resources or services to interact with other services. The only password parameters we use in this solution for Mysql credentials are handled by SSM Parameter Store, and encrypted by KMS, where you can bring your own encryption keys in. Have a think about how long it would take, to run your own encryption tool, implement this level of security for all components. Looking from this viewpoint on the used CI/CD ecosystem, AWS has a very strong story if costs and security matters.
- +1: if you noticed all build and deploy related configurations live in files next to the source code. In my opinion this is a great way to enforce, that CI/CD configuration is version controlled and managed together with the source code itself. While a lot of 3rd party tools (e.g. Jenkins 2.0) allow the same config as code approach, changing the code can be often seen as an omitted step with the "I will just quickly fix this issue from the UI and get it into the codebase later on" approach.

Feedback

We welcome your questions and comments. Please reach out to us at Total Cloud Consulting WEB page
You can visit TC2's GitHub repository to download the templates and scripts for this public release of the deployment guide. Total Cloud Consulting will be updating this guide on a regular basis.

Additional resources

AWS Cloudformation product details

AWS CodeCommit product details

AWS CodeBuild product details

AWS CodeDeploy product details

AWS CodePipeline product details

AWS ECR product details

AWS KMS product details

How AWS Systems Manager Parameter Store Uses AWS KMS