DOCKER BASICS:

1. Introduction:

Cost optimization is often a key business driver of my customers. It’s also a key pillar for a [well-architected](https://wa.aws.amazon.com/index.en.html) design. Usually, there are a few widely applicable strategies for you to save cost on compute resources. For example, you can mix purchase options such as [Savings Plans](https://aws.amazon.com/savingsplans/), on-demand and [Spot](https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc) instances, and you can [right-size](https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-right-sizing/identifying-opportunities-to-right-size.html) your compute resources.

[AWS Graviton2](https://aws.amazon.com/ec2/graviton/) (ARM based) provides up to 40% better price performance over comparable current generation AMD-based instances for a wide variety of workloads. Since the initial launch of Graviton2 instances in Dec 2019, it has more integrations with AWS services such as [Amazon Elastic Kubernetes Service](https://aws.amazon.com/blogs/containers/eks-on-graviton-generally-available/) (Amazon EKS), [Amazon Relational Database Service](https://aws.amazon.com/blogs/aws/new-amazon-rds-on-graviton2-processors/) (RDS), [Elasticache](https://aws.amazon.com/about-aws/whats-new/2020/10/amazon-elasticache-now-supports-m6g-and-r6g-graviton2-based-instances/), [AWS CodeBuild](https://aws.amazon.com/blogs/devops/build-arm-based-applications-using-codebuild/), etc., which makes Graviton2 another widely applicable strategy for you to save cost.

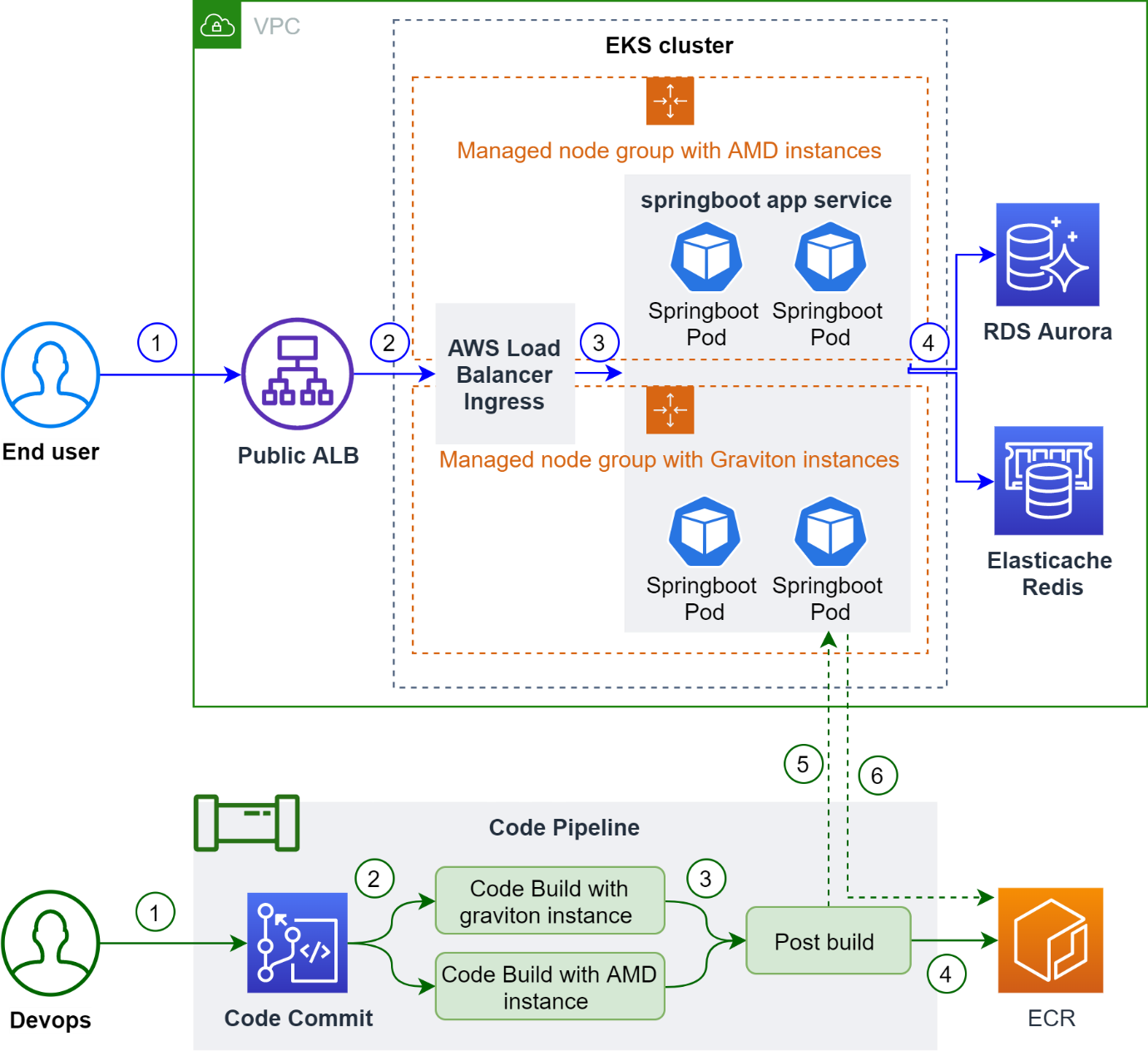
A hybrid ARM and AMD deployment takes advantage of the Graviton2 instances for cost saving, at the same time lowers risks to migrate from AMD to ARM in one shot. With compatibility tested, you will be able to compile, build and deploy your existing code into a multi-architecture EKS cluster without any code change.

This post will cover how to:

* Build a multi-architecture ARM and AMD EKS backend.
* Build an automatic deployment pipeline to compile the Java Spring Boot application code into both ARM & AMD versions, and deploy to EKS backend.
* Use the AWS Cloud Development Kit (CDK) to automate the AWS resource deployments.

The estimated hourly cost for running the resources deployed in this blog post with default settings is less than $1. Be sure to complete the cleanup step once completed.

1. Solution Overview :



1. Spring Boot application backend overview

* End users send request to a public application load balancer (ALB).
* ALB sends a request to [AWS Load Balancer Controller](https://docs.aws.amazon.com/eks/latest/userguide/aws-load-balancer-controller.html) in the EKS cluster. The EKS cluster contains two managed node groups, one with AMD instances, the other with Graviton2 (ARM) instances.
* AWS Load Balancer Controller route requests to the Spring Boot application service. Spring Boot application service contains pods running on both AMD and ARM nodes.
* Spring Boot pods talks to Amazon Aurora, Amazon ElastiCache for Redis and generates the response.

1. Deployment pipeline overview

* Commit code to AWS CodeCommit.
* The code commit triggers AWS CodePipeline, starting with a build phase. The build phase has two tasks running on AWS CodeBuild in parallel. One is on the AMD instance to compile source code and build the AMD Docker image. The other is on the Graviton2 instance to build the ARM Docker image. They both push the Docker image to Amazon Elastic Container Registry (Amazon ECR) at the end of the task.
* When both tasks in build phase succeed, a post build phase is triggered. The post build task creates a container image manifest on top of the ARM and AMD Docker images. The manifest alias will route to the ARM or AMD container image image based on the architecture of the requester.
* The post build task commits the manifest to ECR.
* The post build task runs kubectl to apply Kubernetes config changes and update the Spring Boot service image to the newly created manifest alias.
* Spring Boot service picks up the config change, pods download the image matching the architecture of their hosting nodes, via the manifest alias.

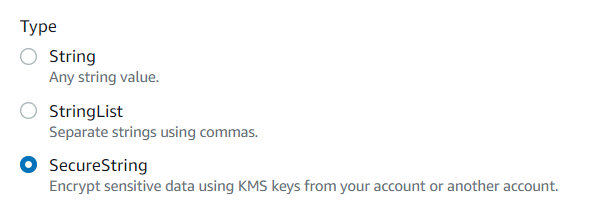
## Solution steps :

Prerequisites

1. Install CDK. Follow the [prerequisites and install guidance](https://docs.aws.amazon.com/cdk/latest/guide/getting_started.html#getting_started_prerequisites). CDK will be used to deploy the application backend and deployment pipeline stacks.
2. Create Docker Hub account and access token in [Docker Hub](https://hub.docker.com/settings/security). The username and token will be used to pull images from Docker Hub during code build phase.
3. Install kubectl. Follow these [instructions](https://kubernetes.io/docs/tasks/tools/install-kubectl/). kubectl will be used to communicate with the EKS cluster.

Step 1: Create AWS Systems Manager Parameter Store in the console

* Search for Systems Manager in services
* Click Parameter Store in the left panel
* Prepare your Docker Hub username and access token
* Click Create to create a new parameter, input ‘Name’ as /springboot-multiarch/dockerhub/username and ‘Value’ as your Docker Hub username. Make sure to choose ‘Type’ as SecureString to protect the data

[](https://d2908q01vomqb2.cloudfront.net/fe2ef495a1152561572949784c16bf23abb28057/2021/02/23/image-2021-02-23T130640.567.png)

* Leave the others as default and click Create Parameter

### **Step 2: Deploy both Spring Boot application backend and deployment pipeline on AWS via CDK**

# Checkout the code

git clone https://github.com/aws-samples/multiarch-eks-springboot-deployment-pipeline-with-cdk.git

# Prepare envcd multiarch-eks-springboot-deployment-pipeline-with-cdk/cdk

python3 -m venv .env

# Run cdk to deploy both springboot application backend and deployment pipeline# Please make sure CodeBuild ARM support(https://aws.amazon.com/codebuild/pricing/) # is available in the chosen region # e.g. ./bootstrap.sh 12345678 us-east-1

./bootstrap.sh {AWS ACCOUNT ID} {REGION}

# Don't forget to note down the CDK outputs# i.e.# backend.EKSConfigCommandxxxx# pipeline.CodeCommitOutput

Bash

### **Step 3: Commit code to CodeCommit to trigger the pipeline**

# Checkout the new codecommit respository created by CDK in step 2

# i.e. value of pipeline.CodeCommitOutput

# (make sure you are in the same filepath as in step 2 where you checkout the code)

# e.g. # ~/environment/multiarch-eks-springboot-deployment-pipeline-with-cdk/cdk (main) $ cd ../..

# ~/environment $

git clone https://git-codecommit.{REGION}.amazonaws.com/v1/repos/springboot-multiarch test

# Copy source code to the new codecommit repositorycd test

cp -r ../multiarch-eks-springboot-deployment-pipeline-with-cdk/\* .

# Commit source code to trigger deployment pipeline

git add \*

git commit -m "trigger commit"

git push

Bash

### **Step 4: Get application load balancer (ALB) address and visit**

# Config kubectl to connect to the EKS cluster created by CDK in step 2# Check CDK output backend.EKSConfigCommandxxxx# e.g. aws eks update-kubeconfig --name {EKS CLUSTER NAME} --region {REGION} --role-arn {EKS MASTER IAM ROLE}# get ALB address from kubernetes cluster

kubectl describe ingress | grep Address

Bash

### **Expected results**

1. Visit the ALB address output from step 4 in the last section. NOTE: You need to wait for about 1 minute before ALB is successfully provisioned.

2. Confirm browser shows an output similar to:

{"RDS Test":"passed","Node Name":"ip-10-xx-xxx-xx.ap-northeast-1.compute.internal","Redis Test":"passed"}

Bash

3. Refresh the page several times to observe the Node Name switch. The nodes are running AMD and ARM (Graviton) correspondingly.

## Cleanup

To tear down the environment, you can execute the following commands:

cd cdk

./cleanup.sh {AWS ACCOUNT ID} {REGION}