Deploy a sample Java microservice on Amazon EKS and expose the microservice using an Application Load Balancer

Created by Vijay Thompson (AWS) and Akkamahadevi Hiremath (AWS)

Environment: PoC or pilot	**Technologies:** Containers & microservices	**Workload:** Open-source
AWS services: Amazon EC2 Container Registry; Amazon EKS; Amazon ECR		

Summary

This pattern describes how to deploy a sample Java microservice as a containerized application on Amazon Elastic Kubernetes Service (Amazon EKS) by using the `eksctl` command line utility and Amazon Elastic Container Registry (Amazon ECR). You can use an Application Load Balancer to load balance the application traffic.

Prerequisites and limitations

Prerequisites

- An active AWS account
- The AWS Command Line Interface (AWS CLI) version 1.7 or later, installed and configured on macOS, Linux, or Windows
- A running Docker daemon
- The `eksctl` command line utility, installed and configured on macOS, Linux, or Windows (For more information, see Getting started with Amazon EKS – eksctl in the Amazon EKS documentation.)
- The `kubectl` command line utility, installed and configured on macOS, Linux, or Windows (For more information, see Installing or updating kubectl in the Amazon EKS documentation.)

Limitations

- This pattern doesn't cover the installation of an SSL certificate for the Application Load Balancer.

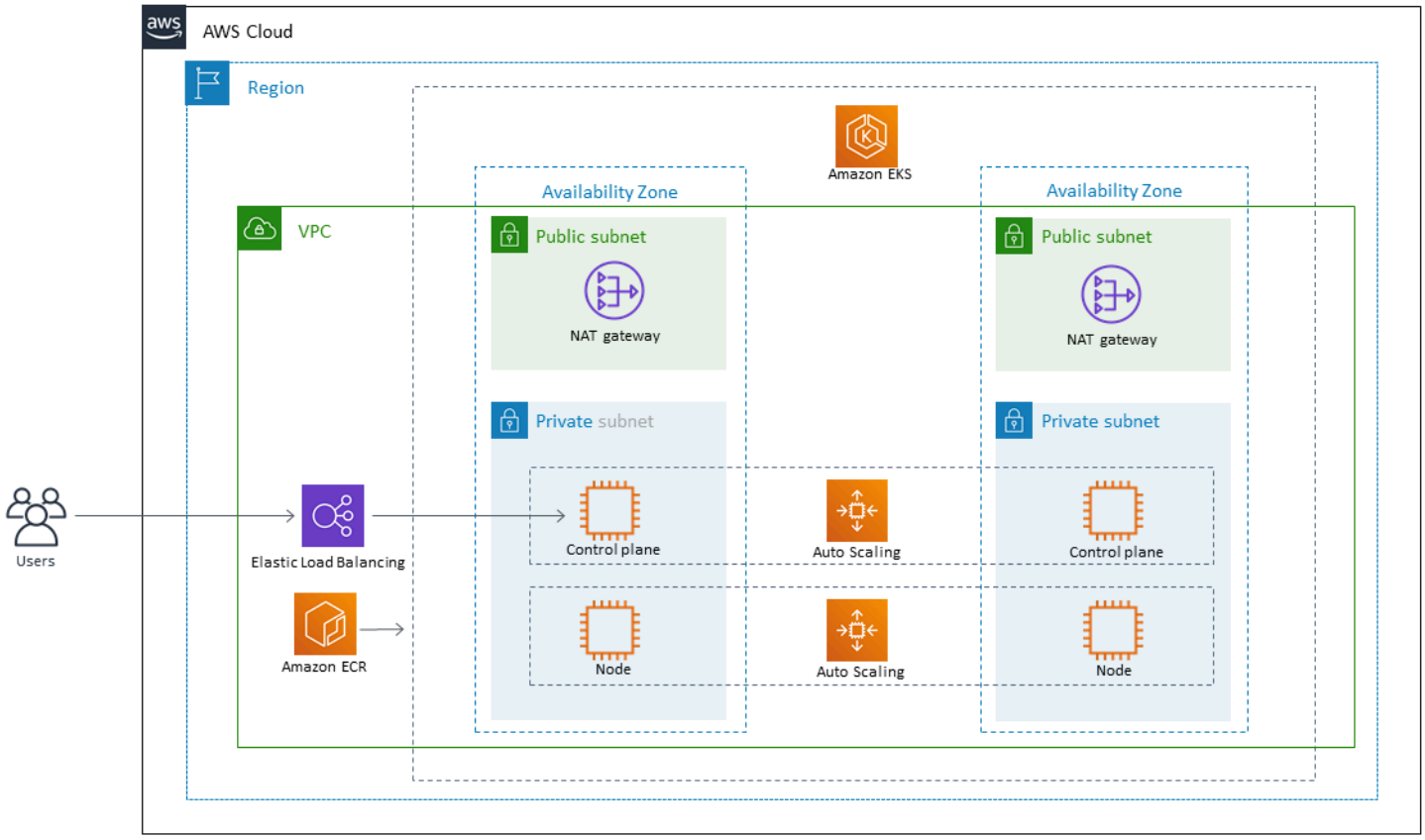
Architecture

Target technology stack

- Amazon ECR
- Amazon EKS
- Elastic Load Balancing

Target architecture

The following diagram shows an architecture for containerizing a Java microservice on Amazon EKS.



Tools

- Amazon Elastic Container Registry (Amazon ECR) is a managed container image registry service that's secure, scalable, and reliable.
- Amazon Elastic Kubernetes Service (Amazon EKS) helps you run Kubernetes on AWS without needing to install or maintain your own Kubernetes control plane or nodes.
- AWS Command Line Interface (AWS CLI) is an open-source tool that helps you interact with AWS services through commands in your command-line shell.
- Elastic Load Balancing automatically distributes your incoming traffic across multiple targets, such as Amazon Elastic Compute Cloud (Amazon EC2) instances, containers, and IP addresses, in one or more Availability Zones.
- eksctl helps you create clusters on Amazon EKS.
- kubectl makes it possible to run commands against Kubernetes clusters.
- Docker helps you build, test, and deliver applications in packages called containers.

Epics

Create an Amazon EKS cluster by using eksctl

Task	Description	Skills required
Create an Amazon EKS cluster.	To create an Amazon EKS cluster that uses two t2.small Amazon EC2 instances as nodes, run the following command: `eksctl create cluster --name <your-cluster-name> --version <version-number> --nodes=1 --node-type=t2.small` **Note**: The process can take between 15 to 20 minutes. After the cluster is created,	Developer, System Admin

Task	Description	Skills required
	the appropriate Kubernetes configuration is added to your kubeconfig file. You can use the `kubeconfig` file with `kubectl` to deploy the application in later steps.	
Verify the Amazon EKS cluster.	To verify that the cluster is created and that you can connect to it, run the `kubectl get nodes` command.	Developer, System Admin

Create an Amazon ECR repository and push the Docker image.

Task	Description	Skills required
Create an Amazon ECR repository.	Follow the instructions from Creating a private repository in the Amazon ECR documentation.	Developer, System Admin
Create a POM XML file.	Create a `pom.xml` file based on the *Example POM file* code in the Additional information section of this pattern.	Developer, System Admin

Create a source file.	Create a source file called `HelloWorld.java` in the `src/main/java/eksExample` path based on the following example:
	`package eksExample;`
	`import static spark.Spark.get;`	

Task	Description	Skills required

	<pre>public class HelloWorl
	d {
	    public static void
	 main(String[] args) {
	        get("/", (req,
	 res) -> {
	            return
	 "Hello World!";
	        });
	    }
	}</pre>
	Be sure to use the following directory structure:
	<pre>### Dockerfile
	### deployment.yaml
	### ingress.yaml
	### pom.xml
	### service.yaml
	### src
	    ### main
	        ### java
	            ###
	 eksExample
	                ###
	 HelloWorld.java</pre>	

Create a Dockerfile.	Create a `Dockerfile` based on the *Example Dockerfile* code in the Additional information section of this pattern.	Developer, System Admin

Task	Description	Skills required

Build and push the Docker image.	In the directory where you want your `Dockerfile` to build, tag, and push the image to Amazon ECR, run the following commands:
	`aws ecr get-login -password --region <region>| docker login --username <username> --password-stdin <account_number>.dkr.ecr.<region>.amazonaws.com`
	`docker buildx build --platform linux/amd64 -t hello-world-java:v1 .`
	`docker tag hello-world-java:v1 <account_number>.dkr.ecr.<region>.amazonaws.com/<repository_name>:v1`
	`docker push <account_number>.dkr.ecr.<region>.amazonaws.com/<repository_name>:v1`
	Note: Modify the AWS Region, account number, and repository details in the preceding commands. Be sure to note the image URL for later use.
	Important: A macOS system with an M1 chip has a problem building an image	

Task	Description	Skills required
	that's compatible with Amazon EKS running on an AMD64 platform. To resolve this issue, use docker buildx to build a Docker image that works on Amazon EKS.	

Deploy the Java microservices

Task	Description	Skills required
Create a deployment file.	Create a YAML file called `deployment.yaml` based on the *Example deployment file* code in the Additional information section of this pattern. **Note**: Use the image URL that you copied earlier as the path of the image file for the Amazon ECR repository.	Developer, System Admin
Deploy the Java microservices on the Amazon EKS cluster.	To create a deployment in your Amazon EKS cluster, run the `kubectl apply -f deployment.yaml` command.	Developer, System Admin
Verify the status of the pods.	1. To verify the status of the pods, run the `kubectl get pods` command. 2. Wait for the status to change to **Ready**.	Developer, System Admin

Task	Description	Skills required
Create a service.	1. Create a file called `service.yaml` based on the *Example service file* code in the Additional information section of this pattern. 2. Run the `kubectl apply -f service.yaml` command.	Developer, System Admin
Install the AWS Load Balancer Controller add-on.	Follow the instructions from Installing the AWS Load Balancer Controller add-on in the Amazon EKS documentation. **Note**: You must have the add-on installed to create an Application Load Balancer or Network Load Balancer for a Kubernetes service.	Devloper, System Admin
Create an ingress resource.	Create a YAML file called `ingress.yaml` based on the *Example ingress resource file* code in the Additional information section of this pattern.	Developer, System Admin
Create an Application Load Balancer.	To deploy the ingress resource and create an Application Load Balancer, run the `kubectl apply -f ingress.yaml` command.	Developer, System Admin

Test the application

Task	Description	Skills required
Test and verify the applicati on.	1. To get the load balancer' s DNS name from the ADDRESS field, run the `kubectl get ingress.networking .k8s.io/java-micro service-ingress` command. 2. On an EC2 instance in the same VPC as your Amazon EKS nodes, run the `curl -v <DNS address from previous command>` command.	Developer, System Admin

Related resources

- Creating a private repository (Amazon ECR documentation)
- Pushing a Docker image (Amazon ECR documentation)
- Ingress Controllers (Amazon EKS Workshop)
- Docker buildx (Docker docs)

Additional information

Example POM file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>helloWorld</groupId>
  <artifactId>helloWorld</artifactId>
  <version>1.0-SNAPSHOT</version>


  <dependencies>
    <dependency>
      <groupId>com.sparkjava</groupId><artifactId>spark-core</
artifactId><version>2.0.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId><artifactId>maven-jar-plugin</
artifactId><version>2.4</version>
        <configuration><finalName>eksExample</finalName><archive><manifest>
              <addClasspath>true</addClasspath><mainClass>eksExample.HelloWorld</
mainClass><classpathPrefix>dependency-jars/</classpathPrefix>
            </manifest></archive>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId><artifactId>maven-compiler-plugin</
artifactId><version>3.1</version>
        <configuration><source>1.8</source><target>1.8</target></configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId><artifactId>maven-assembly-plugin</
artifactId>
        <executions>
          <execution>
            <goals><goal>attached</goal></goals><phase>package</phase>
            <configuration>
              <finalName>eksExample</finalName>
              <descriptorRefs><descriptorRef>jar-with-dependencies</descriptorRef></
descriptorRefs>
              <archive><manifest><mainClass>eksExample.HelloWorld</mainClass></
manifest></archive>
            </configuration>
          </execution>
        </executions>
```

```
      </plugin>
    </plugins>
  </build>
</project>
```

Example Dockerfile

```
FROM bellsoft/liberica-openjdk-alpine-musl:17

RUN apk add maven
WORKDIR /code

# Prepare by downloading dependencies
ADD pom.xml /code/pom.xml
RUN ["mvn", "dependency:resolve"]
RUN ["mvn", "verify"]

# Adding source, compile and package into a fat jar
ADD src /code/src
RUN ["mvn", "package"]

EXPOSE 4567
CMD ["java", "-jar", "target/eksExample-jar-with-dependencies.jar"]
```

Example deployment file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: java-microservice
  template:
    metadata:
      labels:
        app.kubernetes.io/name: java-microservice
    spec:
      containers:
      - name: java-microservice-container
        image: .dkr.ecr.amazonaws.com/:
```

```
        ports:
        - containerPort: 4567
```

Example service file

```
apiVersion: v1
kind: Service
metadata:
  name: "service-java-microservice"
spec:
  ports:
    - port: 80
      targetPort: 4567
      protocol: TCP
  type: NodePort
  selector:
    app.kubernetes.io/name: java-microservice
```

Example ingress resource file

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: "java-microservice-ingress"
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/load-balancer-name: apg2
    alb.ingress.kubernetes.io/target-type: ip
  labels:
    app: java-microservice
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: "service-java-microservice"
                port:
                  number: 80
```