Deployment Process Documentation

# Overview

This documentation outlines the deployment process from code merge in the development branch to the live server. The process involves code hosting on GitHub, continuous integration and continuous deployment (CI/CD) with Jenkins, artifact storage using Nexus, and deployment to Amazon EKS.

# Code Hosting – GitHub

Repositories:

* All source code is hosted on GitHub repositories.
* Location: - rebook-bmpd/
* Branches:

• main

* developer

• uat

Developer push to developer branch which triggers the dev pipelines in Jenkins.

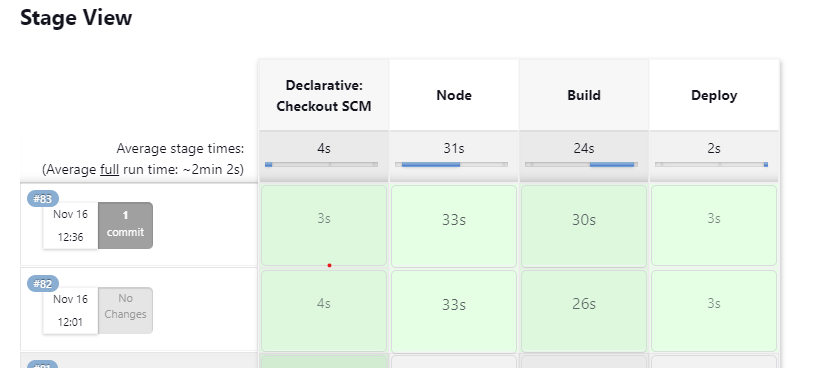
* Developer Pushes to Developer Branch: Developer completes work and pushes changes to the developer branch in the version control system.
* Webhook Triggers Jenkins Pipeline: Webhook notifies Jenkins of new changes in the developer branch.

# Jenkins Server

Jenkins Server is hosted on AWS EC2 instance (admin-bmpd-jenkins-server )

Pipelines:

* Jenkins Pipelines automate the CI/CD process.



• Env.yml: This YAML configuration file that organizes environment-specific variables for an application. It has a base section for common settings and environment-specific sections (e.g., local, QA, production) that inherit or override values. It's used to manage configurations like service URLs, logging preferences, and more across different stages of development, testing, and production.

• Jenkins File: This Jenkinsfile defines a pipeline for building, deploying, and managing a Node.js application using Jenkins. Let's break down the key elements:

- Agent and Build Discarder: The pipeline can run on any available agent (agent any). The build discarder is configured to keep the logs of the last 5 builds.

- Stages:

○ Node: Installs Node.js dependencies and builds the Node.js application. It uses a Docker container named 'bmpd-builder-nodejs' for the build environment.

○ Build: Uses Kaniko to build a Docker image. The image is tagged with the service name and the Git commit ID.

○ Deploy: Deploys the application using Helm. It interacts with a Nexus repository for Helm charts and Docker images. The deployment includes setting various Helm chart values, configuring resource limits, probes, image details, etc.

○ Helm Commands: Helm commands are used to manage Helm charts and releases. The pipeline adds a Helm repository, updates it, and then performs a Helm upgrade or install to deploy the application with specified configurations.

# Artifact Storage – Nexus

Nexus server is hosted on AWS EC2 (admin-bmpd-nexus-server) which is in private subnet require VPN connection for access. Nexus hosts docker images generated during the build process.

* Purpose: The docker images created during build stage are hosted on Nexus repository

Acts as a central repository for artifacts.

Usage:

* Nexus acts as a central repository for artifacts, storing and managing Docker images and node modules.
* Developers and CI/CD pipelines push and pull Docker images to and from Nexus during the build and deployment stages.
* Suppose for “amadeus-service” docker image will be build and pushed to “bmpd-docker-group-nexus/amadeus-service"

# Amazon EKS - Deployment

**Cluster Configuration**

**1. Cluster Type: Amazon EKS (Private)**

Our EKS cluster is configured as a private cluster to enhance security. This means that the control plane is not directly accessible from the internet, ensuring a more robust defense against potential threats.

* Load Balancer: Public Load Balancer for external traffic routing. External traffic is routed through a Public Load Balancer, serving as the entry point to our microservices. This load balancer handles incoming requests and forwards them to internal services within the EKS cluster.
* Ingress: Routing external requests to services within the cluster. Ingress is configured to route external requests intelligently to the appropriate services within the EKS cluster, providing a seamless and scalable experience.
* Internal Services: Exclusively within the EKS cluster.

All microservices are hosted and run exclusively within the EKS cluster, ensuring a contained and secure environment.

## 2. Node Management

* Node Group: Utilizing AWS Node Groups for auto-scaling. For efficient resource management, we utilize AWS Node Groups. These groups are configured for auto-scaling, dynamically adjusting the number of nodes based on the workload demand.
* Autoscaler: Karpenter for dynamic scaling based on resource demand. Karpenter, our chosen cluster autoscaler, plays a crucial role in managing node groups. It monitors resource usage and scales the cluster accordingly, ensuring optimal performance and resource utilization.

## 3. Networking

* VPC Configuration: Private subnets for enhanced security. Our VPC is designed with private subnets, adding an extra layer of isolation and security to our microservices environment.
* Load Balancer: Directs external traffic to internal services. The Public Load Balancer serves as a gateway for external traffic, efficiently directing requests to the internal services within the EKS cluster.

## 4. Logging and Monitoring

* Logging: CloudWatch for container and application logs. Container and application logs are seamlessly integrated with Amazon CloudWatch, offering a centralized platform for monitoring and troubleshooting.
* Monitoring: Prometheus and Grafana for advanced metrics.
* For advanced monitoring, we leverage Prometheus and Grafana, providing comprehensive insights into the performance and health of our microservices.