

Rubiscape CI/CD Onboarding

& Upgradation - Rollbacks

This Document describes the details about the CI/CD Implementation in AWS environment for the Upgradation, Rollbacks for various partners .

Table of Contents

[Overview 1](#_2jig7flnpnzi)

[Pipelines Involved 1](#_vnhk7kq0abdt)

[Master Pipeline Functionality 2](#_r447jqjy0524)

[Rollback Mechanism 3](#_fn0lg1loxhsl)

[Summary 3](#_hl2ygrcw14tt)

[Code walkthrough for various stages : 3](#_6y9sllczl006)

[Pipeline Stages and Logic 3](#_yrl5xye6ye9p)

[1. Set Build Name 3](#_3utwru1r6nxq)

[2. Read and Print Environment File 4](#_dj7prrjmtaxt)

[3. Trigger Pipelines Based on Flags 5](#_xjpf8px3dsho)

[Summary 8](#_wdcj14m0j8b1)

[Jenkins Frontend-Pipeline Documentation 9](#_4oy1w15y2v5b)

[Overview 9](#_ksvveiua753d)

[Stages Explanation 9](#_z1nxg2iy4qji)

[Backend Pipeline 19](#_s933czwr92wa)

[Stages Explanation 20](#_gy4gcdr1gtew)

[Purpose 37](#_hwtl9earfwgb)

[New Client Onboarding 41](#_6psuk9iekf1y)

[Overview 41](#_q6misbv26sdn)

[Parameters 42](#_t2pknulwmcsw)

[Environment Variables 42](#_535pmh8k12nv)

[Stages Breakdown 43](#_m72rwvs7yedk)

# 

# 

# 

# **Overview**

The upgrade process is managed through three distinct Jenkins pipelines, each responsible for handling different aspects of the upgrade workflow. If any stage fails, a rollback will be triggered immediately as a post-failure action at the stage where the failure occurred.

# **Pipelines Involved**

1. Master Pipeline
2. Frontend Pipeline
3. Backend Pipeline

# Master Pipeline Functionality

The Master Pipeline serves as the orchestrator and determines which specific pipeline(s) should be triggered based on configuration settings. The decision is made using a configuration file located at:

|  |
| --- |
| /var/lib/jenkins/workspace/master-pipeline/customers.env |

Configuration File Format

This configuration file (customers.env) contains environment variables that define which pipeline(s) need to be executed. Below are examples illustrating how different values in the file dictate the pipeline execution:

* Example 1: Backend Upgrade Only

|  |
| --- |
| interim=backend dummy=false datalab=false partner=false sense=false ALL=false |

In this case, only the Backend Pipeline will be triggered for the upgrade process.

Example 2: Both Frontend and Backend Upgrades

|  |
| --- |
| interim=both dummy=false datalab=false partner=false sense=false ALL=false |

This configuration instructs the Master Pipeline to trigger both the Frontend Pipeline and Backend Pipeline for the upgrade process.

## Rollback Mechanism

* If any pipeline fails at any stage, a rollback action will be executed at that specific stage.
* Rollback is performed as a post-failure step, ensuring that the environment remains stable and consistent.

## Summary

* The Master Pipeline decides which pipelines to run based on customers.env.
* The Frontend Pipeline handles frontend-specific upgrades.
* The Backend Pipeline manages backend upgrades.
* In case of failure at any stage, rollback is triggered automatically at that exact stage.

This document provides an overview of how the upgrade process works, ensuring a structured approach to handling pipeline execution and failure recovery.

## Code walkthrough for various stages :

## Pipeline Stages and Logic

### 1. Set Build Name

Purpose:

* Generates a unique build name based on the current date and build number.
* Ensures the pipeline name is set correctly.

Logic:

* Retrieves the current date in YYYYMMDD format.
* Extracts the build number from the Jenkins environment variables (defaults to 0 if not set).
* Uses PIPELINE\_NAME if available; otherwise, sets it to Rubiscape\_Master\_3.4.2.
* Constructs a build name using the format: PIPELINE\_NAME\_YYYYMMDD\_BUILDNUMBER.
* Updates the build display name in Jenkins.

Code

|  |
| --- |
| stage('Set Build Name') {  steps {  script {  def BUILD\_DATE = sh(script: "date +%Y%m%d", returnStdout: true).trim()  def buildNumber = env.BUILD\_NUMBER ?: "0"  def pipelineName = env.PIPELINE\_NAME ?: "Rubiscape\_Master\_3.4.2"    def buildName = "${pipelineName}\_${BUILD\_DATE}\_${buildNumber}"  currentBuild.displayName = buildName  echo "🚀 Build name set to: ${buildName}"  }  } } |

### 

### 2. Read and Print Environment File

Purpose:

* Loads and validates the customers.env configuration file.

Logic:

* Checks if customers.env exists in the workspace.
* Reads the file content and prints it for verification.
* If the file is missing, the pipeline fails with an error.

Code:

|  |
| --- |
| stage('Read and Print Environment File') {  steps {  script {  def envFile = 'customers.env'   if (fileExists(envFile)) {  def content = readFile(envFile)  echo "🔹 Customers Environment File Loaded:\n${content}"  } else {  error "❌ customers.env file not found!"  }  }  } } |

### 3. Trigger Pipelines Based on Flags

Purpose:

* Determines which customer pipelines need to be triggered based on the customers.env configuration file.

Logic:

* Reads customers.env line by line, ignoring comments and empty lines.
* Parses key-value pairs where the key represents the customer name and the value represents the pipeline type (frontend, backend, or both).
* If ALL=true, triggers both frontend and backend pipelines for all customers.
* If specific customers have frontend, backend, or both enabled, triggers the corresponding pipelines for those customers.
* Uses Jenkins build step to trigger the respective pipelines (frontend-pipeline and backend-pipeline).
* Captures build numbers for reference and logs them.
* If no valid customers are found, logs a warning and skips pipeline execution.

Code:

|  |
| --- |
| stage('Trigger Pipelines Based on Flags') {  steps {  script {  def envFile = 'customers.env'  def customerBuilds = [:]  def allEnabled = false   if (fileExists(envFile)) {  def envVars = readFile(envFile).split("\n")   for (line in envVars) {  line = line.trim()  if (line && !line.startsWith("#")) {  def parts = line.split("=")  if (parts.length != 2) continue   def customer = parts[0].trim()  def pipelineType = parts[1].trim().toLowerCase()   if (customer == "ALL" && pipelineType == "true") {  allEnabled = true  break  }   if (pipelineType in ["frontend", "backend", "both"]) {  echo "🚀 Triggering ${pipelineType} pipeline for ${customer}..."   if (pipelineType == "frontend" || pipelineType == "both") {  def frontendBuild = build job: "frontend-pipeline", wait: true, parameters: [string(name: 'CUSTOMER', value: customer)]  customerBuilds["${customer}-frontend"] = frontendBuild.getNumber()  }   if (pipelineType == "backend" || pipelineType == "both") {  def backendBuild = build job: "backend-pipeline", wait: true, parameters: [string(name: 'CUSTOMER', value: customer)]  customerBuilds["${customer}-backend"] = backendBuild.getNumber()  }  }  }  }   if (allEnabled) {  echo "✅ ALL is enabled. Triggering both pipelines for all customers."    def allCustomers = envVars.findAll { it.contains("=") && !it.startsWith("ALL=") }  .collect { it.split("=")[0].trim() }    for (customer in allCustomers) {  echo "🚀 Triggering both pipelines for ${customer}..."  def frontendBuild = build job: "frontend-pipeline", wait: true, parameters: [string(name: 'CUSTOMER', value: customer)]  def backendBuild = build job: "backend-pipeline", wait: true, parameters: [string(name: 'CUSTOMER', value: customer)]   customerBuilds["${customer}-frontend"] = frontendBuild.getNumber()  customerBuilds["${customer}-backend"] = backendBuild.getNumber()  }  }  } else {  error "❌ customers.env file not found!"  }   if (!customerBuilds.isEmpty()) {  echo "🔹 Build Numbers for Triggered Pipelines:"  customerBuilds.each { customer, buildNumber ->  echo "✅ Pipeline triggered for ${customer} (Build #${buildNumber})"  }  } else {  echo "⚠️ No enabled customers found in customers.env. Skipping pipeline triggers."  }  }  } } |

## Summary

* Set Build Name: Generates a structured build name.
* Read and Print Environment File: Loads and validates customers.env.
* Trigger Pipelines Based on Flags: Reads customers.env and triggers the required pipelines dynamically.
* Rollback Mechanism: If a triggered pipeline fails, it triggers rollback at the failed stage.

# 

# 

# 

# 

# 

# Jenkins Frontend-Pipeline Documentation

## Overview

This pipeline automates the build and deployment process for the Rubiscape Frontend application. Each stage is designed to handle specific tasks such as setting up the environment, installing dependencies, building the application, and deploying it to the target server.

## Stages Explanation

Environment Variables :

|  |
| --- |
| pipeline {  agent any   environment {  WORKING\_DIR = 'Rubiscape\_Frontend\_Repository/Rubiscape'  PRODUCT\_VERSION = "3.4.1"  BUILD\_DATE = sh(script: "date +%Y%m%d", returnStdout: true).trim()  BUILD\_NUMBER = "${env.BUILD\_NUMBER}"  BUILD\_NAME = "Rubiscape\_UI\_${PRODUCT\_VERSION}\_${BUILD\_DATE}\_${BUILD\_NUMBER}"  BRANCH\_NAME = 'Release/3.4.1/V\_3.4.1\_rc'  WORK\_DIR = "/home/ubuntu/${params.CUSTOMER}"  REPO\_PATH = "/var/lib/jenkins/workspace/frontend-pipeline"  } |

### 

This environment configuration defines essential variables for the frontend pipeline, ensuring consistency in build and deployment processes. It specifies workspace directories, product version, branch name, and repository path.

The build name is dynamically generated using the product version, date, and build number, enabling better tracking and management of different builds across deployments.

1. Set Build Name

Purpose:

* Define a unique build name to track the deployment.

Logic:

* Uses environment variables like PRODUCT\_VERSION, BUILD\_DATE, and BUILD\_NUMBER to generate a build name.
* Displays the build name in the Jenkins UI.

Code:

|  |
| --- |
| stage('Set Build Name') {  steps {  script {  currentBuild.displayName = BUILD\_NAME  echo "🔹 Build name set to: ${BUILD\_NAME}"  }  } } |

### 

2. Setup Node.js with NVM

Purpose:

* Ensures the correct version of Node.js is installed and used.

Logic:

* Checks if NVM is installed; if not, it installs it.
* Loads NVM and sets the required Node.js version (18.20.1).
* Confirms the installation of Node.js and NPM.

Code:

|  |
| --- |
| stage('Setup Node.js with NVM') {  steps {  script {  sh '''  set -ex  echo "🔹 Checking if NVM is installed..."  if [ ! -d "$HOME/.nvm" ]; then  echo "📥 Installing NVM..."  curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash  fi  echo "🔹 Loading NVM..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  [ -s "$NVM\_DIR/bash\_completion" ] && . "$NVM\_DIR/bash\_completion"  echo "🔹 Installing and using Node.js 18.20.1..."  nvm install 18.20.1  nvm use 18.20.1  echo "✅ Node.js Version: $(node -v)"  echo "✅ NPM Version: $(npm -v)"  '''  }  } } |

3. Clone Repository

Purpose:

* Fetch the latest code from the Azure Git repository.

Logic:

* Uses Jenkins credentials (AZURE\_FRONT\_GIT\_TOKEN) for authentication.
* Clones the repository and checks out the correct branch (BRANCH\_NAME).

Code:

|  |
| --- |
| stage('Clone Repository') {  steps {  withCredentials([string(credentialsId: 'AZURE\_FRONT\_GIT\_TOKEN', variable: 'AZURE\_PAT')]) {  sh '''  set -e  echo "🔹 Cloning repository..."  git clone https://techops:${AZURE\_PAT}@dev.azure.com/rubicslabs/Rubiscape/\_git/Rubiscape\_Frontend\_Repository  cd Rubiscape\_Frontend\_Repository  git checkout ${BRANCH\_NAME}  echo "✅ Repository cloned and checked out to ${BRANCH\_NAME}"  '''  }  } } |

4. Install Dependencies

Purpose:

* Install the required Node.js dependencies for the project.

Logic:

* Uses NPM to install dependencies inside the working directory.
* Ensures the correct Node.js version is used.

Code:

|  |
| --- |
| stage('Install Dependencies') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "📥 Installing dependencies..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  npm install  '''  }  }  } } |

5. Force Update node\_modules

Purpose:

* Ensures dependencies are updated correctly.

Logic:

* Runs the postinstall script to fix issues with dependencies.

Code:

|  |
| --- |
| stage('Force Update node\_modules') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "🔄 Running postinstall script..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  npm run postinstall  '''  }  }  } } |

6. Run prod-build-dev

Purpose:

* Builds the frontend application.

Logic:

* Uses npm run prod-build-dev with increased memory allocation.

Code:

|  |
| --- |
| stage('Run prod-build-dev') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "🏗️ Building project..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  NODE\_OPTIONS=--max\_old\_space\_size=4096 npm run prod-build-dev  '''  }  }  } } |

7. Write Version

Purpose:

* Stores the build version in a file for tracking.

Logic:

* Creates an assets directory and writes the BUILD\_NAME to a file.

Code:

|  |
| --- |
| stage('Write Version') {  steps {  script {  sh '''  set -e  echo "📝 Writing build version..."  mkdir -p Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets  sudo chmod -R 777 Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets  echo "${BUILD\_NAME}" | tee Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets/version.txt  echo "✅ Version file updated to ${BUILD\_NAME}."  '''  }  } } |

8. Copying Root for FRONTEND

Purpose:

* Deploys the new frontend build while preserving the previous version as a backup.

Logic:

* Moves the existing ROOT directory to a timestamped backup folder.
* Deploy New Build: Copies the newly built ROOT directory from the workspace to the deployment target.
* Compare Backend Configs: Runs a script to check for differences in backendservices.json.
* Restore Configuration: If necessary, restore backendservices.json from the previous version.
* Rollback Mechanism: On failure, reverts to the last working ROOT version.

Code:

|  |
| --- |
| stage('Copying Root for FRONTEND') {  steps {  script {  sh '''  set -e  echo "📂 Copying built files to target directory..."  TIMESTAMP=$(date +"%Y%m%d%H%M")  NEW\_DIR="${WORK\_DIR}/ROOT\_Old\_${TIMESTAMP}"  echo "📌 Moving old ROOT to: $NEW\_DIR"  sudo mv "${WORK\_DIR}/ROOT" "$NEW\_DIR"  echo "📌 Copying new ROOT files..."  sudo cp -r $WORKSPACE/Rubiscape\_Frontend\_Repository/Rubiscape/ROOT "${WORK\_DIR}/"  echo "📌 Comparing backend service files..."  sudo /home/ubuntu/scripts/compare\_backendservicefiles.sh "${WORK\_DIR}/ROOT/assets/js/backendservices.json" "$NEW\_DIR/assets/js/backendservices.json"  echo "📌 Restoring backendservices.json..."  sudo rm -f "${WORK\_DIR}/ROOT/assets/js/backendservices.json"  sudo cp "$NEW\_DIR/assets/js/backendservices.json" "${WORK\_DIR}/ROOT/assets/js/backendservices.json"  echo "✅ Completed processing for ${WORK\_DIR}"  '''  }  }  post {  failure {  script {  sh '''  echo "❌ Copying Root failed. Rolling back..."  TIMESTAMP=$(ls -td ${WORK\_DIR}/ROOT\_Old\_\* | head -1)  if [ -d "$TIMESTAMP" ]; then  echo "🔄 Rolling back to previous ROOT version: $TIMESTAMP"  sudo rm -rf "${WORK\_DIR}/ROOT"  sudo mv "$TIMESTAMP" "${WORK\_DIR}/ROOT"  echo "✅ Rollback completed."  else  echo "⚠️ No previous ROOT backup found. Manual intervention required."  fi  '''  }  }  }  }  } |

Summary:

* Set Build Name: Assigns a unique name to track the deployment.
* Setup Node.js with NVM: Ensures the correct Node.js version is installed and used.
* Clone Repository: Fetches the latest code from Azure Git.
* Install Dependencies: Installs required Node.js dependencies using npm.
* Force Update node\_modules: Runs postinstall to fix dependency issues.
* Run prod-build-dev: Builds the frontend application with increased memory.
* Write Version: Stores the build version in a file for tracking.
* Copying Root for FRONTEND: Deploys the new build while keeping a backup. On failure, rollbacks to the last working ROOT version.

## Backend Pipeline

This Jenkins pipeline automates the deployment of Rubiscape Backend version 3.4.1. It begins with database backup to S3, clones the repository from Azure DevOps, encrypts Python files, and sets up AWS/EKS integration.

Key stages include copying migrations from live pods, moving built files to working directories, running database migrations, and restarting application pods.   
The pipeline features comprehensive error handling with automatic rollback capability for both code and database issues, ensuring system reliability during upgrades. Each stage includes detailed logging and status checks to verify successful completion.

Environment variables control deployment parameters, allowing customization for different customer environments.

### Stages Explanation

Parameters:

* Defines a parameter CUSTOMER, which allows customization of the deployment directory based on the customer invoking the backend pipeline.

Environment Variables :

|  |
| --- |
| pipeline {  agent any   parameters {  string(name: 'CUSTOMER', defaultValue: 'dummy', description: 'Customer invoking the backend pipeline')  }   environment {  BRANCH\_NAME = "Release/3.4.1/V\_3.4.1\_rc"  REPO\_URL = "https://dev.azure.com/rubicslabs/Rubiscape/\_git/Rubiscape\_Backend\_Repository"  WORK\_DIR = "/home/ubuntu/${params.CUSTOMER}"  RUBISCAPE\_VERSION = "3.4.1"  PYTHON\_BIN = "/home/Python-3.11.3/rubiscape\_env/bin/python3"  PYCONCRETE\_ADMIN = "/home/Python-3.11.3/rubiscape\_env/bin/pyconcrete-admin.py"  EKS\_CLUSTER\_NAME = 'rubiscape-prod-cluster'  AWS\_REGION = 'ap-south-1'  BUILD\_DATE = sh(script: "date +%Y%m%d", returnStdout: true).trim()  DB\_DEPLOYMENT\_SCRIPT = "DatabaseScripts/Release3\_4\_1/\_\_FinalDeploymentScript3\_4\_1.py"  // Rollback environment variables  S3\_BUCKET = "db-backups-main"  S3\_BACKUP\_PATH = "partner/"  LOCAL\_DUMP\_FILE = "latest\_backup.sql"  WORK\_DIR\_INTERIM = "/home/ubuntu/interim"  NAMESPACE = "dummy"  DB\_POD\_PREFIX = "dummy-psql-"  RESTART\_TIMEOUT = "60" // Seconds to wait for pod to restart  POD\_READY\_TIMEOUT = "60" // Seconds to wait for pod to be ready after restart  // Added for database backup  DB\_BACKUP\_S3\_PATH = "datalabs/"  DB\_BACKUP\_NAMESPACE = "datalab"  DB\_BACKUP\_POD\_PREFIX = "postgresql-"  DB\_BACKUP\_FILE = "db\_backup\_${BUILD\_NUMBER}.sql"  } |

The pipeline uses environment variables to manage deployment configuration, including paths, versions, and AWS settings like region and cluster name. Additional environment variables control rollback procedures, database connections, and pod management timeouts to enable flexible, customer-specific deployments.

1. Initialize Variables

Purpose:  
 Define a unique pipeline name to track the deployment.

Logic:

* Uses environment variables like RUBISCAPE\_VERSION, BUILD\_DATE, and BUILD\_NUMBER to generate a pipeline name
* Sets the pipeline name in Jenkins UI for better tracking.

Code:

|  |
| --- |
| stage('Initialize Variables') {  steps {  script {  def BUILD\_DATE = sh(script: "date +%Y%m%d", returnStdout: true).trim()  env.PIPELINE\_NAME = "Rubiscape\_BE\_${env.RUBISCAPE\_VERSION}\_${BUILD\_DATE}\_${env.BUILD\_NUMBER}"  currentBuild.displayName = env.PIPELINE\_NAME  echo "🔹 Pipeline name set to: ${env.PIPELINE\_NAME}"  }  } } |

2. Database Backup

Purpose:  
 Create a backup of the PostgreSQL database and store it in S3.

Logic:

* Configures the Kubernetes context for accessing the cluster.
* Finds the active PostgreSQL pod in the backup namespace.
* Retrieves database credentials from the pod.
* Dumps the PostgreSQL database and saves it as a backup file.
* Copies the backup file from the pod to the Jenkins workspace.
* Uploads the backup to an S3 bucket and cleans up temporary files.

Code:

|  |
| --- |
| stage('Database Backup') {  steps {  script {  echo "🔹 Starting database backup procedure..."  sh "aws eks update-kubeconfig --region ${AWS\_REGION} --name ${EKS\_CLUSTER\_NAME}"   env.POSTGRES\_BACKUP\_POD = sh(script: """  kubectl get pods -n ${DB\_BACKUP\_NAMESPACE} -o custom-columns=:metadata.name | grep ^${DB\_BACKUP\_POD\_PREFIX} | head -n 1  """, returnStdout: true).trim()   if (!env.POSTGRES\_BACKUP\_POD) {  error("⚠️ PostgreSQL pod not found for backup!")  }   env.DB\_BACKUP\_USER = sh(script: "kubectl exec -n ${DB\_BACKUP\_NAMESPACE} ${env.POSTGRES\_BACKUP\_POD} -- printenv POSTGRES\_USER", returnStdout: true).trim()  env.DB\_BACKUP\_PASS = sh(script: "kubectl exec -n ${DB\_BACKUP\_NAMESPACE} ${env.POSTGRES\_BACKUP\_POD} -- printenv POSTGRES\_PASSWORD", returnStdout: true).trim()  env.DB\_BACKUP\_NAME = sh(script: "kubectl exec -n ${DB\_BACKUP\_NAMESPACE} ${env.POSTGRES\_BACKUP\_POD} -- printenv POSTGRES\_DB", returnStdout: true).trim()   echo "✅ Database for backup: ${env.DB\_BACKUP\_NAME}"   sh """  kubectl exec -n ${DB\_BACKUP\_NAMESPACE} ${env.POSTGRES\_BACKUP\_POD} -- bash -c "su - postgres -c 'PGPASSWORD=${env.DB\_BACKUP\_PASS} pg\_dump -U ${env.DB\_BACKUP\_USER} ${env.DB\_BACKUP\_NAME} > /tmp/${DB\_BACKUP\_FILE}'"  """  sh "kubectl cp ${DB\_BACKUP\_NAMESPACE}/${env.POSTGRES\_BACKUP\_POD}:/tmp/${DB\_BACKUP\_FILE} ./${DB\_BACKUP\_FILE}"  sh "aws s3 cp ./${DB\_BACKUP\_FILE} s3://${S3\_BUCKET}/${DB\_BACKUP\_S3\_PATH}${DB\_BACKUP\_FILE}"  sh "rm -f ${DB\_BACKUP\_FILE}"   echo "✅ Backup uploaded to S3 successfully!"  }  } } |

### 

3. Clone Repository

Purpose:  
 Fetch the latest codebase from the Azure repository.

Logic:

* Uses Azure DevOps credentials to authenticate the Git repository.
* Clones the repository into the working directory.
* Check out the specified branch for deployment.

Code:

4.Write Version

Purpose:  
 Generate version files with the pipeline name for tracking builds.

Logic:

* Navigates to the Rubics directory inside the backend repository.
* Creates and writes the README.md and version.txt files with the pipeline name.

Code:

|  |
| --- |
| stage('Write Version') {  steps {  sh '''#!/usr/bin/bash  echo '📌 Writing Version'  cd ${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics  echo '${PIPELINE\_NAME}'    cat > README.md <<EOF ## Rubiscape BE Build Version > ${PIPELINE\_NAME} EOF   cat > version.txt <<EOF ${PIPELINE\_NAME} EOF   echo "✅ Version files created successfully."  '''  } } |

5. Encrypt Python Files

Purpose:  
 Encrypt sensitive Python files to protect the source code.

Logic:

* Moves urls.py outside the Rubics/Rubics directory to avoid encryption.
* Runs the PyConcrete encryption command on specified Python files.
* Moves urls.py back to its original location.

Code:

|  |
| --- |
| stage('Encrypt Python Files') {  steps {  sh '''#!/usr/bin/bash  echo '🔐 Encrypting Python files...'    mv ${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics/Rubics/urls.py ${WORK\_DIR}/Rubiscape\_Backend\_Repository/    ${PYTHON\_BIN} ${PYCONCRETE\_ADMIN} compile --source=${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics/ \  --pye --remove-py -i \  sparkPrefixCode.py modifiedUpdate.py tokenizer.py BasicCode.py rexecute.py asgi.py wsgi.py manage.py \  entityrelation.py sparkPrefixCode.py settings.py taskOrch.py pythonPrefixCode.py IoT\_subscriber.py \  customScheduleWrapper.py   mv ${WORK\_DIR}/Rubiscape\_Backend\_Repository/urls.py ${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics/Rubics/    echo "✅ Encryption completed successfully."  '''  } } |

6. Setup Environment

Purpose:  
 Configure AWS and EKS environment settings before deployment.

Logic:

* Verifies AWS credentials by checking the caller identity.
* Updates the EKS kubeconfig to interact with the Kubernetes cluster.

Code:

|  |
| --- |
| stage('Setup Environment') {  steps {  script {  echo "🔹 Setting up AWS & EKS Environment..."  sh """  aws sts get-caller-identity  aws eks update-kubeconfig --region ${AWS\_REGION} --name ${EKS\_CLUSTER\_NAME}  """  }  } } |

7. Fetch Backend Pod Name

Purpose:  
 Retrieve the name of the running backend pod in a specified namespace.

Logic:

* Searches for the backend pod in the namespace e.g., dummy .
* Extracts the first matching pod name.
* Stores the pod name in an environment variable for later use.=
* Fails the build if no backend pod is found.

Code:

|  |
| --- |
| stage('Fetch Backend Pod Name') {  steps {  script {  def namespace = "dummy"  echo "🔹 Extracting backend pod from namespace: ${namespace}"    def podName = sh(  script: """  kubectl get pods -n "${namespace}" | grep backend | awk '{print \$1}' | head -1  """,  returnStdout: true  ).trim()   if (podName) {  echo "✅ Backend Pod in ${namespace}: ${podName}"  env.POD\_NAME = podName  } else {  error "⚠️ No backend pod found in ${namespace}! Failing the build."  }  }  } } |

8. Copy Migrations to Host

Purpose:  
 Copy Django migration files from the backend pod to the local host machine.

Logic:

* Defines the source (podPath) and destination (hostPath) directories.
* Loops through the list of Django apps and copies their migrations directories using kubectl exec.
* Extracts the copied files into the corresponding directories on the host.

Code:

|  |
| --- |
| stage('Copy Migrations to Host') {  steps {  script {  echo "🔹 Copying Django migrations from pod to host..."  def hostPath = "${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics"  def podPath = "/home/ubuntu/Rubics"  def namespace = "dummy" // Remove when running with def namespace = params.CUSTOMER   def folders = ["Admin", "Rubics", "IoT", "IRS\_Commentry", "Data\_reconciliation", "Visualization"]   for (folder in folders) {  sh """  kubectl exec -n "${namespace}" "${env.POD\_NAME}" -- tar czf - -C ${podPath}/${folder} migrations | tar xzf - -C ${hostPath}/${folder}  """  echo "✅ Copied migrations for ${folder}"  }  }  } } |

9. Copy Built Files to Work Directory

Purpose:  
 Move built files to the working directory while keeping a backup of the previous version.

Logic:

* Generates a timestamp and renames the existing Rubics directory as a backup.
* Copies the latest built files from the backend repository to the work directory.
* Verifies the copied files.
* Uses a script to compare settingConfig.json and restore it from the previous backup.
* Removes the cloned backend repository after copying.

Code:

|  |
| --- |
| stage('Copy Built Files to Work Directory') {  steps {  script {  sh '''  set -e  echo "📂 Copying built files to target directory..."    TIMESTAMP=$(date +"%Y%m%d%H%M")  echo ${TIMESTAMP}    OLD\_DIR="${WORK\_DIR}/Rubics\_Old\_${TIMESTAMP}"  echo ${OLD\_DIR}  echo ${WORK\_DIR}    mv ${WORK\_DIR}/Rubics ${WORK\_DIR}/Rubics\_Old\_${TIMESTAMP}  cp -r ${WORK\_DIR}/Rubiscape\_Backend\_Repository/Rubics ${WORK\_DIR}    ls ${WORK\_DIR}/Rubics  echo "✅ Copy operation completed."    echo "📂 Copying settingConfig.json file..."  sudo /home/ubuntu/scripts/comapre\_settingconf.sh ${WORK\_DIR}/Rubics/settingConfig.json ${OLD\_DIR}/settingConfig.json    echo "📌 Restoring settingConfig.json . . ."  sudo rm -rf ${WORK\_DIR}/Rubics/settingConfig.json  sudo cp "${OLD\_DIR}/settingConfig.json" "${WORK\_DIR}/Rubics/settingConfig.json"    echo "✅ Completed processing for ${WORK\_DIR}"    echo "📌 Removing the cloned directory Rubiscape\_Backend\_Repository . . ."  sudo rm -rf "${WORK\_DIR}/Rubiscape\_Backend\_Repository"  '''  }  } } |

10. Run Database Migrations

Purpose:

This stage is responsible for running Django database migrations inside the backend Kubernetes pod. If the migration fails, it triggers a rollback mechanism, restoring the database from the latest backup stored in an S3 bucket.

Logic:

* Run Migrations:
  + Connects to the backend pod inside the Kubernetes cluster.
  + Activates the Python virtual environment.
  + Navigates to the Django project directory.
  + Runs makemigrations and migrate for multiple Django apps.
  + Executes a custom deployment script.
* Post-Failure Rollback:
  + If the migration fails, the pipeline:
    - Changes ownership of the project directory to Jenkins.
    - Rolls back to the previous application version (if available).
    - Updates the Kubernetes context.
    - Fetches the latest PostgreSQL database backup from S3.
    - Finds the PostgreSQL pod.
    - Retrieves PostgreSQL environment variables (user, password, database name).
    - Copies the backup file into the PostgreSQL pod.
    - Restores the database.
    - If the restore fails, it restarts the PostgreSQL deployment and retries the restore.
    - Cleans up temporary backup files.
    - Marks the pipeline as failed but confirms that rollback was successful.

Code:

|  |
| --- |
| stage('Run Database Migrations') {  steps {  script {  echo "🔹 Running Django migrations inside the backend pod..."  sh """  kubectl exec -it -n "dummy" "${env.POD\_NAME}" -- /bin/bash -c "  source /home/ubuntu/Python-3.11.3/rubiscape\_env/bin/activate  cd Rubics  python3 manage.py makemigrations Admin  python3 manage.py makemigrations Rubics  python3 manage.py makemigrations IoT  python3 manage.py makemigrations IRS\_Commentary  python3 manage.py makemigrations Data\_reconcilation  python3 manage.py makemigrations Visualization  python3 manage.py migrate  python3 manage.py shell -c 'exec(open(\\\"${DB\_DEPLOYMENT\_SCRIPT}\\\").read())'  "  """  }  }  post {  failure {  script {  echo "❌ Database migration failed! Initiating rollback..."    // Rollback to previous version  sh '''  set -e  sudo chown -R jenkins:jenkins /home/ubuntu/interim/Rubics/Rubics  echo "🔄 Rolling back to previous version due to migration failure..."  LAST\_BACKUP=$(ls -td ${WORK\_DIR}/Rubics\_Old\_\* | head -1 || echo "")   if [ -d "$LAST\_BACKUP" ]; then  echo "✅ Found backup: $LAST\_BACKUP. Restoring..."  rm -rf ${WORK\_DIR}/Rubics  mv "$LAST\_BACKUP" ${WORK\_DIR}/Rubics  echo "✅ Rollback successful."  else  echo "⚠️ No backup found! Rollback failed."  exit 1  fi  '''    // Configure Kubernetes context for DB restore  sh "aws eks update-kubeconfig --region ap-south-1 --name rubiscape-prod-cluster"    // Fetch Latest Backup from S3  def latestBackup = sh(script: """  aws s3 ls s3://${env.S3\_BUCKET}/${env.S3\_BACKUP\_PATH} --recursive |   grep -E '\\.sql\$' |   awk '{print \$4, \$1, \$2, \$3}' |   sort -k2,2r -k3,3r |   head -n 1 |   awk '{print \$1}'  """, returnStdout: true).trim()    if (!latestBackup) {  error("No .sql backup file found in S3!")  }    echo "Latest backup file: ${latestBackup}"  sh "aws s3 cp s3://${env.S3\_BUCKET}/${latestBackup} ./${env.LOCAL\_DUMP\_FILE}"    // Find PostgreSQL Pod  env.POSTGRES\_POD = sh(script: """  kubectl get pods -n ${env.NAMESPACE} -o custom-columns=:metadata.name | grep ^${env.DB\_POD\_PREFIX} | head -n 1  """, returnStdout: true).trim()    echo "PostgreSQL Pod: ${env.POSTGRES\_POD}"    if (!env.POSTGRES\_POD) {  error("PostgreSQL pod not found!")  }    env.DB\_USER = sh(script: "kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- printenv POSTGRES\_USER", returnStdout: true).trim()  env.DB\_PASS = sh(script: "kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- printenv POSTGRES\_PASSWORD", returnStdout: true).trim()  env.DB\_NAME = sh(script: "kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- printenv POSTGRES\_DB", returnStdout: true).trim()    // Copy Backup to Pod  sh "kubectl cp ${env.LOCAL\_DUMP\_FILE} ${env.NAMESPACE}/${env.POSTGRES\_POD}:/tmp/${env.LOCAL\_DUMP\_FILE}"  sh "kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- bash -c 'chown postgres:postgres /tmp/${env.LOCAL\_DUMP\_FILE} && chmod 644 /tmp/${env.LOCAL\_DUMP\_FILE}'"    // Restore Database  try {  sh """  kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- bash -c "su - postgres -c 'PGPASSWORD=${env.DB\_PASS} psql -d ${env.DB\_NAME} -f /tmp/${env.LOCAL\_DUMP\_FILE}'"  """  echo "Restore completed successfully"  } catch (Exception e) {  echo "Error during restore: ${e.message}"  def deploymentName = sh(script: "kubectl get deployment -n ${env.NAMESPACE} -l app=postgresql -o custom-columns=:metadata.name | head -n 1", returnStdout: true).trim()  if (deploymentName) {  echo "Restarting PostgreSQL deployment: ${deploymentName}"  sh "kubectl scale deployment ${deploymentName} -n ${env.NAMESPACE} --replicas=0"  sh "kubectl wait --for=delete pod -l app=postgres -n ${env.NAMESPACE} --timeout=30s"  sh "kubectl scale deployment ${deploymentName} -n ${env.NAMESPACE} --replicas=1"  sh "sleep 10"  env.POSTGRES\_POD = sh(script: """  kubectl get pods -n ${env.NAMESPACE} -o custom-columns=:metadata.name | grep ^${env.DB\_POD\_PREFIX} | head -n 1  """, returnStdout: true).trim()  sh "kubectl wait --for=condition=ready pod/${env.POSTGRES\_POD} -n ${env.NAMESPACE} --timeout=${env.RESTART\_TIMEOUT}s"  sh "kubectl cp ${env.LOCAL\_DUMP\_FILE} ${env.NAMESPACE}/${env.POSTGRES\_POD}:/tmp/${env.LOCAL\_DUMP\_FILE}"  sh "kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- bash -c 'chown postgres:postgres /tmp/${env.LOCAL\_DUMP\_FILE} && chmod 644 /tmp/${env.LOCAL\_DUMP\_FILE}'"  sh """  kubectl exec -n ${env.NAMESPACE} ${env.POSTGRES\_POD} -- bash -c "su - postgres -c 'PGPASSWORD=${env.DB\_PASS} psql -d ${env.DB\_NAME} -f /tmp/${env.LOCAL\_DUMP\_FILE}'"  """  } else {  error("Could not find PostgreSQL deployment. Database restore failed.")  }  }    // Cleanup  sh "rm -f ${env.LOCAL\_DUMP\_FILE}"    // Display message indicating that the pipeline failed but rollback completed  echo "❌ Database migration failed but rollback completed successfully."  error("Database migration failed, rollback completed. Pipeline stopping.")  }  }  } } |

11. Fetch and Restart Pods

### Purpose

* This Jenkins pipeline stage is designed to:
* Identify and Restart Pods: It fetches running pods in a specified Kubernetes namespace, filters them based on predefined conditions, and deletes them to force a restart.
* Ensure Pod Readiness: It waits for a predefined timeout (env.POD\_READY\_TIMEOUT) and checks if all restarted pods are back in a running state.
* Perform Rollback if Needed: If any pod fails to come back up, the script restores the previous codebase from a backup and restarts the pods again.
* Final Validation: It ensures all pods are running post-rollback. If issues persist, the pipeline fails with an error message.

Logic :

* Determine the Namespace & Filter Criteria
  + If namespace == "sense", filter pods containing backend, celery, visual, or celery2.
  + Otherwise, filter pods containing backend, celery, or visual.
* Fetch and Restart Pods
  + Uses kubectl get pods with grep and awk to find matching pods.
  + Deletes each identified pod (kubectl delete pod).
  + Stores restarted pod names in an array.
* Wait and Check Pod Readiness
  + Sleeps for env.POD\_READY\_TIMEOUT seconds.
  + Iterates over restarted pods, extracting their names and checking if they are in the Running state.
  + If any pod is not in the Running state, stores it in a list of notRunningPods.
* Perform Rollback if Pods Are Not Running
  + If some pods fail to start, it finds the last backup directory (Rubics\_Old\_\*).
  + Restores the previous version of the code (Rubics directory).
  + Deletes non-running pods again and waits for them to restart.
* Validate Rollback Success
  + Checks if all pods are running after rollback.
  + If successful, the pipeline fails with a message to fix the code in Azure.
  + If unsuccessful, the pipeline fails, indicating rollback issues.

Code:

|  |
| --- |
| stage('Fetch and Restart Pods') {  steps {  script {  def namespace = "dummy"  def filter = namespace == "sense" ? "backend|celery|visual|celery2" : "backend|celery|visual"  def restartedPods = []   // Fetch pod names matching the filter  def pods = sh(script: "kubectl get pods -n ${namespace} --no-headers | grep -E '${filter}' | awk '{print \$1}'", returnStdout: true).trim()   if (pods) {  echo "Restarting pods in ${namespace}: ${pods}"    // Restart pods  pods.split("\n").each { pod ->  sh "kubectl delete pod ${pod} -n ${namespace}"  restartedPods.add(pod)  }   // Wait for restart  sh "sleep ${env.POD\_READY\_TIMEOUT}"   // Validate pod readiness  def allRunning = true  def notRunningPods = []   restartedPods.each { podName ->  def basePodName = podName.split("-")[0..<-1].join("-")  def podStatus = sh(script: "kubectl get pods -n ${namespace} --no-headers | grep ${basePodName} | awk '{print \$1,\$3}'", returnStdout: true).trim()   if (podStatus) {  def statusParts = podStatus.tokenize(' ')  if (statusParts[1] != "Running") {  allRunning = false  notRunningPods.add("${statusParts[0]}: ${statusParts[1]}")  }  } else {  allRunning = false  notRunningPods.add("${basePodName}: Not found")  }  }   // Handle rollback if necessary  if (!allRunning) {  echo "⚠️ Some pods are not running. Initiating rollback..."  def lastBackupDir = sh(script: "cd ${WORK\_DIR} && ls -ltd Rubics\_Old\_\* | head -1 | awk '{print \$9}'", returnStdout: true).trim()   if (lastBackupDir) {  sh """  cd ${WORK\_DIR}  sudo rm -rf Rubics  sudo mv ${lastBackupDir} Rubics  """   // Restart pods after rollback  notRunningPods.each { podInfo ->  sh "kubectl delete pod ${podInfo.split(':')[0].trim()} -n ${namespace} || true"  }   sh "sleep ${env.POD\_READY\_TIMEOUT}"   // Final validation  def allRunningAfterRollback = sh(script: "kubectl get pods -n ${namespace} --no-headers | grep -E '${filter}' | awk '{print \$1,\$3}'", returnStdout: true).trim().split("\n").every { it.tokenize(' ')[1] == "Running" }   if (allRunningAfterRollback) {  error "✅ ROLLBACK SUCCESSFUL. Pipeline failed, fix Rubics code in Azure repo."  } else {  error "❌ ROLLBACK FAILED. Fix Rubics code and check pods."  }  } else {  error "❌ No backup found. Fix Rubics code in Azure repo."  }  } else {  echo "✅ All pods successfully restarted."  }  } else {  echo "No matching pods found in ${namespace}."  }  }  } }  post {  success {  echo "✅ Pipeline completed successfully!"  }  failure {  echo "❌ Pipeline failed!"  } } |

# New Client Onboarding

## Overview

This Jenkins pipeline automates the deployment of the Rubiscape application on an AWS-based Kubernetes (EKS) cluster. It performs tasks like cloning repositories, setting up AWS resources, configuring Kubernetes, building and deploying the frontend and backend, running database migrations, and restarting necessary services.

This Jenkins pipeline is used to automate the deployment of the Rubiscape application on AWS. It involves:

* Cloning repositories from Azure DevOps.
* Setting up infrastructure using AWS CloudFormation.
* Mounting an AWS EFS file system.
* Deploying backend and frontend services on an AWS EKS cluster.
* Running database migrations.

Repository Link: [Azure Repo](https://dev.azure.com/rubicslabs/Rubiscape-Searce-AWS-migration/_git/New%20Client%20Onboarding)

## Parameters

|  |
| --- |
| string(name: 'REGION', defaultValue: 'ap-south-1', description: 'AWS Region (ap-south-1 for Mumbai)')  string(name: 'TENANT', defaultValue: 'jenkins-interim', description: 'Tenant Name')  string(name: 'tenantDomain', defaultValue: 'cicd.rubiscape.com', description: 'Base domain for the tenant')  string(name: 'backendPath', defaultValue: '/backend/', description: 'Path for backend service')  string(name: 'visualizationPath', defaultValue: '/visualization/', description: 'Path for visualization service') |

REGION: A string parameter to define the AWS region where the deployment will take place. Default is Mumbai (ap-south-1).

## Environment Variables

|  |
| --- |
| environment {  CLUSTER\_NAME = "rubiscape-prod-cluster"  TENANT = "jenkins-interim"  WORKING\_DIR = 'Rubiscape\_Frontend\_Repository/Rubiscape'  PRODUCT\_VERSION = "3.4.1"  BUILD\_DATE = sh(script: "date +%Y%m%d", returnStdout: true).trim()  BUILD\_NUMBER = "${env.BUILD\_NUMBER}"  BUILD\_NAME = "Rubiscape\_UI\_${PRODUCT\_VERSION}\_${BUILD\_DATE}\_${BUILD\_NUMBER}"  BRANCH\_NAME = 'Release/3.4.1/V\_3.4.1\_rc'  RUBISCAPE\_VERSION = "3.4.1"  REPO\_URL = "https://dev.azure.com/rubicslabs/Rubiscape/\_git/Rubiscape\_Backend\_Repository"  PYTHON\_BIN = "/home/Python-3.11.3/rubiscape\_env/bin/python3"  PYCONCRETE\_ADMIN = "/home/Python-3.11.3/rubiscape\_env/bin/pyconcrete-admin.py"  DB\_DEPLOYMENT\_SCRIPT = "DatabaseScripts/StaticTableScripts/A\_MainScript.py" } |

Key Variables

CLUSTER\_NAME: AWS EKS cluster where Kubernetes services are deployed.

TENANT: Tenant name, used to create isolated environments.

WORKING\_DIR: Directory where frontend code is located.

PRODUCT\_VERSION: The current application version.

BUILD\_DATE: Captures the current date dynamically.

BUILD\_NUMBER: Jenkins build number.

BUILD\_NAME: Custom name for the frontend build.

BRANCH\_NAME: The branch to be checked out in repositories.

REPO\_URL: Backend repository URL.

PYTHON\_BIN: Python executable path inside the virtual environment.

PYCONCRETE\_ADMIN: Tool used for encrypting Python files.

DB\_DEPLOYMENT\_SCRIPT: Script for applying database migrations.

## Stages Breakdown

Stage 1: Clone Infra Repository

* Uses Azure DevOps Personal Access Token (PAT) stored in Jenkins credentials (NEW\_ONBOARDING).
* Clones the New Client Onboarding repository from Azure DevOps.
* Saves the repository in a folder named after the TENANT.

|  |
| --- |
| stage('Clone Infra Repository') {  steps {  withCredentials([string(credentialsId: 'NEW\_ONBOARDING', variable: 'NEW\_ONBOARDING\_PAT')]) {  sh '''#!/usr/bin/bash  set -e  echo "🔹 Cloning repository..."  git clone https://${NEW\_ONBOARDING\_PAT}@dev.azure.com/rubicslabs/Rubiscape-Searce-AWS-migration/\_git/New%20Client%20Onboarding ${TENANT}  echo "Repository cloned successfully"  '''  }  }  } } |

Stage 2: Validate CloudFormation Template

* Changes the working directory to TENANT.
* Creates a directory in /home/ubuntu/${TENANT} for AWS deployment.
* Validates the CloudFormation template using AWS CLI.

|  |
| --- |
| stage('Validate Template') {  steps {  script {  sh """  cd ${TENANT}  sudo mkdir -p /home/ubuntu/${TENANT}  aws cloudformation validate-template --template-body file://mum-stacsket-uat.yaml --region ${params.REGION}  """  }  }  } |

Stage 3: Create Stack

|  |
| --- |
| aws cloudformation deploy --template-file "$WORKSPACE/${TENANT}/mum-stacsket-uat.yaml" --stack-name ${TENANT}-stack --parameter-overrides Environment=prod TenantName=${TENANT} --capabilities CAPABILITY\_NAMED\_IAM --region ${params.REGION} |

Deploys the CloudFormation stack using AWS CloudFormation.

Stage 4: Create Stack

* Waits for the CloudFormation stack to be created or updated.

|  |
| --- |
| aws cloudformation wait stack-create-complete \  --stack-name ${TENANT}-stack \  --region ${params.REGION} || \ aws cloudformation wait stack-update-complete \  --stack-name ${TENANT}-stack \  --region ${params.REGION} |

* The command waits until the stack is fully created or updated.

Stage 5: Get EFS DNS

|  |
| --- |
| aws efs describe-file-systems --region ${params.REGION} \ --query "FileSystems[?Name=='${TENANT}-prod-efs'].FileSystemId" --output text | \ awk '{print $1".efs.${params.REGION}.amazonaws.com"}' |

Uses AWS CLI to find the EFS File System ID and converts it to a valid EFS DNS.

Stage 6: Mount EFS on Jenkins Server

|  |
| --- |
| sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport ${env.EFS\_DNS}:/ ${mountPath} df -h |

Mounts the EFS storage to a specific location on the Jenkins server.

Stage 7: Update Kubeconfig

* Updates the kubectl configuration to interact with the EKS cluster.

|  |
| --- |
| aws eks update-kubeconfig --name ${env.CLUSTER\_NAME} --region ${params.REGION} |

* Enables interaction with the EKS cluster.

Stage 8: Verify Cluster Status

* Lists the nodes in the Kubernetes cluster.

|  |
| --- |
| kubectl get nodes |

* Ensures the EKS cluster is running.

Stage 9. Setup Node.js with NVM

* Installs and configures Node.js using NVM.

|  |
| --- |
| stage('Setup Node.js with NVM') {  steps {  script {  sh '''  set -ex  echo "🔹 Checking if NVM is installed..."  if [ ! -d "$HOME/.nvm" ]; then  echo "📥 Installing NVM..."  curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash  fi  echo "🔹 Loading NVM..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  [ -s "$NVM\_DIR/bash\_completion" ] && . "$NVM\_DIR/bash\_completion"  echo "🔹 Installing and using Node.js 18.20.1..."  nvm install 18.20.1  nvm use 18.20.1  echo "✅ Node.js Version: $(node -v)"  echo "✅ NPM Version: $(npm -v)"  '''  }  }  } |

* Enable Debug Mode: set -ex ensures commands are printed before execution and stops on errors.
* Check NVM Installation: If ~/.nvm does not exist, download and install NVM.
* Load NVM: Set NVM\_DIR, source nvm.sh, and enable bash completion.
* Install & Use Node.js: Install Node.js version 18.20.1 via NVM and set it as the active version.
* Verify Installation: Print installed Node.js (node -v) and NPM (npm -v) versions to confirm setup.

Stage 10. Clone Frontend Repository

* Clones the frontend repository from Azure DevOps.

|  |
| --- |
| git clone https://techops:${AZURE\_PAT}@dev.azure.com/rubicslabs/Rubiscape/\_git/Rubiscape\_Frontend\_Repository cd Rubiscape\_Frontend\_Repository git checkout ${BRANCH\_NAME} |

* Ensures the correct branch is checked out.

Stage 11. Install Dependencies

* Installs the frontend dependencies using npm.

|  |
| --- |
| stage('Install Dependencies') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "📥 Installing dependencies..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  npm install  '''  }  }  }  } |

* Sets the environment variable for NVM.
* source $NVM\_DIR/nvm.sh: Loads NVM into the shell session.
* nvm use 18.20.1: Ensures Node.js 18.20.1 is used.
* npm install: Installs project dependencies from package.json.

Stage 12. Force Update node\_modules

* Runs the postinstall script to resolve dependency issues.

|  |
| --- |
| stage('Force Update node\_modules') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "🔄 Running postinstall script..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  npm run postinstall  '''  }  }  }  } |

* set -e → Stops execution on error.
* export NVM\_DIR="$HOME/.nvm" → Sets NVM directory.
* [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh" → Loads NVM.
* Uses Node.js 18.20.1.
* npm run postinstall → Runs postinstall script from package.json.

Stage 13. Run prod-build-dev

* Builds the frontend project.

|  |
| --- |
| stage('Run prod-build-dev') {  steps {  script {  dir(WORKING\_DIR) {  sh '''  set -e  echo "🏗️ Building project..."  export NVM\_DIR="$HOME/.nvm"  [ -s "$NVM\_DIR/nvm.sh" ] && . "$NVM\_DIR/nvm.sh"  nvm use 18.20.1  NODE\_OPTIONS=--max\_old\_space\_size=4096 npm run prod-build-dev  '''  }  }  }  } |

* NODE\_OPTIONS=--max\_old\_space\_size=4096 npm run prod-build-dev → Allocates 4GB memory and runs the prod-build-dev script from package.json.

Stage 14. Write Frontend Version

* Creates a version file for frontend.

Commands:

|  |
| --- |
| stage('Write Frontend Version') {  steps {  script {  sh '''  set -e  echo "📝 Writing build version..."  mkdir -p Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets  sudo chmod -R 777 Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets  echo "${BUILD\_NAME}" | tee Rubiscape\_Frontend\_Repository/Rubiscape/ROOT/assets/version.txt  echo "✅ Version file updated to ${BUILD\_NAME}."  '''  }  }  } |

* Stores the build name inside a version.txt file.

Stage 15. Copying Root for FRONTEND

* Moves the frontend build files to the deployment directory.

|  |
| --- |
| sudo cp -r $WORKSPACE/Rubiscape\_Frontend\_Repository/Rubiscape/ROOT "/home/ubuntu/${TENANT}/" |

Stage 16. Clone Backend Repository

* Clones the backend repository from Azure DevOps.

|  |
| --- |
| git clone https://${AZURE\_PAT}@dev.azure.com/rubicslabs/Rubiscape/\_git/Rubiscape\_Backend\_Repository cd Rubiscape\_Backend\_Repository git checkout ${BRANCH\_NAME} |

Stage 17. Encrypt Python Files

|  |
| --- |
| stage('Encrypt Python Files') {  steps {  sh '''#!/usr/bin/bash  echo '🔐 Encrypting Python files...'  mv Rubiscape\_Backend\_Repository/Rubics/Rubics/urls.py Rubiscape\_Backend\_Repository/    ${PYTHON\_BIN} ${PYCONCRETE\_ADMIN} compile --source=Rubiscape\_Backend\_Repository/Rubics/ \  --pye --remove-py -i \  sparkPrefixCode.py modifiedUpdate.py tokenizer.py BasicCode.py rexecute.py asgi.py wsgi.py manage.py \  entityrelation.py sparkPrefixCode.py settings.py taskOrch.py pythonPrefixCode.py IoT\_subscriber.py \  customScheduleWrapper.py    mv Rubiscape\_Backend\_Repository/urls.py Rubiscape\_Backend\_Repository/Rubics/Rubics/  echo "✅ Encryption completed successfully."  sudo cp -r Rubiscape\_Backend\_Repository/Rubics /home/ubuntu/${TENANT}  '''  }  } |

* Moves `urls.py` temporarily.
* Uses PyConcrete to encrypt listed Python files, remove `.py` files, and generate `.pye` files.
* Moves `urls.py` back to its original location.
* Copies the encrypted files to the specified tenant directory.

Stage 18. Setup Environment

|  |
| --- |
| stage('Setup Environment') {  steps {  script {  echo "🔹 Setting up AWS & EKS Environment..."  sh """  aws sts get-caller-identity  aws eks update-kubeconfig --region ${params.REGION} --name ${CLUSTER\_NAME}  """  }  }  } |

### 

* Verifies AWS credentials by retrieving the caller identity.
* Updates kubeconfig to interact with the specified EKS cluster.

Stage 19. Deploy Helm

|  |
| --- |
| stage('Deploy helm') {  steps {  script {  sh """  cd ${TENANT}/helm  aws eks update-kubeconfig --name ${env.CLUSTER\_NAME} --region ${params.REGION}  EFS\_ID=\$(echo "${env.EFS\_DNS}" | cut -d'.' -f1)  helm install ${TENANT} . -n ${TENANT} --create-namespace -f values.yaml --set global.efsId=\${EFS\_ID} --set global.namespace=${TENANT} --set global.tenantName=${TENANT} --set ingress.rules.hostDNS=${params.tenantDomain} --set ingress.rules.visualizationPath=${params.backendPath} --set ingress.rules.visualizationPath=${params.visualizationPath}  sleep 45  """  }  }  } |

* Navigates to the Helm deployment directory.
* Configures kubectl to interact with the specified EKS cluster.
* Extracts the EFS ID from the provided DNS.
* Deploys the Helm chart with specified values for EFS, namespace, tenant, and ingress rules.
* Pauses execution for 45 seconds to allow the deployment to stabilize.

Stage 20. Modify JSON Config

* Updates backend configuration files.

|  |
| --- |
| stage('Modify JSON Config') {  steps {  script {  sh """  sudo chmod +x /home/ubuntu/scripts/URL\_update\_frontend.sh  sudo /home/ubuntu/scripts/URL\_update\_frontend.sh /home/ubuntu/${TENANT}/ROOT/assets/js/backendservices.json ${params.tenantDomain} ${params.backendPath} ${params.visualizationPath}  echo "✅ backendservices.json file updated "  """  }  }  } |

* sudo chmod +x /home/ubuntu/scripts/URL\_update\_frontend.sh` → Grants execute permissions to the script.
* sudo /home/ubuntu/scripts/URL\_update\_frontend.sh /home/ubuntu/${TENANT}/ROOT/assets/js/backendservices.json ${params.tenantDomain} ${params.backendPath} ${params.visualizationPath}` → Executes the script to update `backendservices.json` with provided parameters.

Stage 21. Run Database Migrations

|  |
| --- |
| stage('Update settingsConfig.json') {  steps {  script {  env.POSTGRES\_POD = sh(script: """  kubectl get pods -n ${TENANT} -o custom-columns=:metadata.name | grep postgresql | head -n 1  """, returnStdout: true).trim()   echo "PostgreSQL Pod: ${env.POSTGRES\_POD}"   if (!env.POSTGRES\_POD) {  error("PostgreSQL pod not found!")  }   env.DB\_USER = sh(script: "sudo kubectl exec -n ${TENANT} ${env.POSTGRES\_POD} -- printenv POSTGRES\_USER", returnStdout: true).trim()  env.DB\_PASS = sh(script: "sudo kubectl exec -n ${TENANT} ${env.POSTGRES\_POD} -- printenv POSTGRES\_PASSWORD", returnStdout: true).trim()  env.DB\_NAME = sh(script: "sudo kubectl exec -n ${TENANT} ${env.POSTGRES\_POD} -- printenv POSTGRES\_DB", returnStdout: true).trim()   sh """  sudo chmod +x /home/ubuntu/scripts/update\_db.sh  sudo /home/ubuntu/scripts/update\_db.sh $WORKSPACE/Rubiscape\_Backend\_Repository/Rubics/settingConfig.json ${TENANT}   sudo chmod +x /home/ubuntu/scripts/db\_creds.sh  sudo /home/ubuntu/scripts/db\_creds.sh $WORKSPACE/Rubiscape\_Backend\_Repository/Rubics/settingConfig.json $DB\_NAME $DB\_USER $DB\_PASS   sudo rm -rf /home/ubuntu/${TENANT}/Rubics/settingConfig.json  sudo cp $WORKSPACE/Rubiscape\_Backend\_Repository/Rubics/settingConfig.json /home/ubuntu/${TENANT}/Rubics/settingConfig.json  """  }  }  } |

* Retrieves the name of the first PostgreSQL pod in the namespace.
* Fetches the PostgreSQL username from the pod.
* Fetches the PostgreSQL password from the pod.
* Fetches the PostgreSQL database name from the pod.
* Grants execute permissions to the update\_db.sh script.
* Updates settingConfig.json with tenant-specific configurations.
* Grants execute permissions to the db\_creds.sh script.
* Updates settingConfig.json with database credentials.
* Copies the updated settingConfig.json to the target directory.

### 

Stage 22. Fetch Backend Pod Name

|  |
| --- |
| stage('Fetch Backend Pod Name') {  steps {  script {  def namespace = "${TENANT}"  echo "🔹 Extracting backend pod from namespace: ${namespace}"  def podName = sh(  script: """  kubectl get pods -n "${namespace}" | grep backend | awk '{print \$1}' | head -1  """,  returnStdout: true  ).trim()   if (podName) {  echo "✅ Backend Pod in ${namespace}: ${podName}"  env.POD\_NAME = podName  } else {  error "⚠️ No backend pod found in ${namespace}! Failing the build."  }  }  }  } |

### 

* Retrieves the name of the first backend pod in the specified namespace.
* Stores the retrieved backend pod name in the environment variable POD\_NAME.

Stage 23. Run Database Migrations

|  |
| --- |
| stage('Run Database Migrations') {  steps {  script {  echo " Running Django migrations inside the backend pod..."  sh """  kubectl exec -it -n ${TENANT} "${env.POD\_NAME}" -- /bin/bash -c "  source /home/ubuntu/Python-3.11.3/rubiscape\_env/bin/activate  cd Rubics  pwd  python3 manage.py makemigrations Admin  python3 manage.py makemigrations Rubics  python3 manage.py makemigrations IoT  python3 manage.py makemigrations IRS\_Commentary  python3 manage.py makemigrations Data\_reconcilation  python3 manage.py makemigrations Visualization  python3 manage.py migrate  echo 1 | python3 manage.py shell -c 'exec(open(\\\"${DB\_DEPLOYMENT\_SCRIPT}\\\").read())'  "  """  }  }  } |

* Executes the migration commands inside the backend pod.
* Activates the Python virtual environment.
* Creates migration files for specified Django apps.
* Applies database migrations.
* Executes a database deployment script inside the Django shell.

Stage 24. Fetch and Restart Pods

|  |
| --- |
| stage('Fetch and Restart Pods') {  steps {  script {  def namespace = "${TENANT}"  def filter = namespace == "sense" ? "backend|celery|visual|celery2" : "backend|celery|visual"  def restartedPods = []   // Get pods matching the filter  def pods = sh(script: "kubectl get pods -n ${namespace} --no-headers | grep -E '${filter}' | awk '{print \$1}'", returnStdout: true).trim()    if (pods) {  echo "Restarting pods in ${namespace}:"  echo "${pods}"    pods.split("\n").each { pod ->  echo "Deleting pod: ${pod}"  sh "kubectl delete pod ${pod} -n ${namespace}"  restartedPods.add(pod)  sleep 10  }  }  else {  echo "No matching pods found in ${namespace}."  }  }  }  } |

* Defines the pod name patterns to filter based on the namespace.
* Lists pod names matching the filter in the specified namespace.
* Iterates over each pod name.
* Deletes the matching pods to trigger a restart.
* Waits for 10 seconds after deleting each pod to allow restart stabilization.