AWS multi-tenant

Upgrade EKS

What This Script Does

This script automates the installation and upgrade of **Karpenter** on an **Amazon EKS (Elastic Kubernetes Service)** cluster using **Helm**.

What is Karpenter?

Karpenter is an **autoscaler** for Kubernetes that automatically provisions and deprovisions EC2 instances based on workload demands. Unlike the traditional **Cluster Autoscaler**, Karpenter directly interacts with EC2, making scaling decisions faster and more efficient.

Script Breakdown

The script:

1. **Finds an existing EKS cluster**
2. **Finds necessary AWS IAM roles and resources**
3. **Deploys/Upgrades Karpenter using Helm**
4. **Configures Karpenter with proper IAM permissions and Kubernetes settings**
5. **Deploys additional Karpenter components (EC2NodeClass, NodePool)**

Step-by-Step Execution

1. Load Environment Variables & Functions

```
. ${WORKSPACE}/scripts/functions.sh

```

This loads helper functions (like `plog` for logging and `die` for error handling) from a script.

2. Enable Debug Mode (Optional)

```
debug=false
if [ "$1" == "debug" ];then
    debug=true
fi

```

If the script is run with `debug` mode (`./script.sh debug`), it enables Helm's `--debug` and `--dry-run` flags.

---

3. Get the EKS Cluster Name

```
cluster_name=$(aws eks list-clusters | jq -r '.clusters[0]')
if [ -z "${cluster_name}" ];then
    die "Can't get cluster_name"
fi

```

- Retrieves the **first** EKS cluster in AWS using `aws eks list-clusters`.
- If no cluster is found, it **exits with an error**.

✅ **Why?** Karpenter **must** be installed in an existing EKS cluster.

---

4. Get the Karpenter IAM Role

```
karpenter_role=$(aws iam list-roles| jq -r '.[][].Arn' | grep KarpenterController)
if [ -z "${karpenter_role}" ];then
    die "Can't get karpenter_role"
fi

```

- Searches for an **IAM Role** with `KarpenterController` in its name.
- If no role is found, the script **exits**.

✅ **Why?** Karpenter requires an IAM Role to interact with EC2 and autoscale nodes.

---

5. Get the Karpenter Node IAM Role

```
karpenter_role_name=$(aws iam list-roles| jq -r '.[][].RoleName' | grep Karpenter-eks)
if [ -z "${karpenter_role_name}" ];then
    die "Can't get karpenter_role_name"
```

```
fi

```

- Finds the IAM Role Name for worker nodes managed by Karpenter.

✅ **Why?** Each Karpenter-provisioned node needs an **IAM Role** with EC2 permissions.

---

6. Set Default Versions for Karpenter

```
if [ -z "${version_tag}" ];then
    version_tag="0.35.4"
fi

if [ -z "${version_digest}" ];then
    version_digest="sha256:d59b5d4c58011615b593dd80f115106efad034104c2d7f016c01c19d8e0b21d9"
fi

if [ -z "${image_repo}" ];then
        image_repo="690359577697.dkr.ecr.us-east-1.amazonaws.com/infra"
fi

if [ -z "${chart_version}" ];then
        chart_version="0.36.0"
fi

```

- Defines **default versions** for:
  - Karpenter Helm Chart
  - Karpenter Docker Image
  - Image repository location

✅ **Why?** Allows easy upgrades by modifying **chart_version** and **image_tag**.

---

7. Get the Karpenter SQS Queue

```
karpenter_queue=$(aws sqs list-queues | jq -r '.QueueUrls[]' | grep Karpenter | awk -F\/ '{print $NF}')
if [ -z "${karpenter_queue}" ];then
        die "Can't get karpenter_queue"
fi

```

- Retrieves the **SQS queue name** for Karpenter.
- This queue handles **interruption events** (e.g., when a spot instance is about to be terminated).

✅ **Why?** Karpenter needs an **SQS queue** to gracefully handle instance terminations.

---

8. Get the Cluster Endpoint

```
cluster_endpoint=$(aws eks describe-cluster --name ${cluster_name} | jq -r '.cluster.endpoint')
if [ -z "${cluster_endpoint}" ];then
        die "Can't get cluster_endpoint"
fi

```

- Retrieves the **API endpoint** of the EKS cluster.

✅ **Why?** Karpenter must know the **Kubernetes API** location to manage nodes.

---

9. Create a Service-Linked Role for Spot Instances

```
aws iam create-service-linked-role --aws-service-name spot.amazonaws.com > /dev/null 2>&1

```

- Ensures AWS has a **service-linked role** for managing **Spot instances**.

✅ **Why?** Karpenter can provision **Spot Instances** to save costs.

---

10. Install/Upgrade Karpenter Using Helm

```
helm upgrade --install ${debug_info} karpenter oci://public.ecr.aws/karpenter/karpenter \
  --version "${chart_version}" --namespace "karpenter" --create-namespace \
  --set "serviceAccount.annotations.eks.amazonaws.com/role-arn=${karpenter_role}" \
  --set "logLevel=debug" \
  --set "settings.clusterName=${cluster_name}" \
  --set "settings.clusterEndpoint=${cluster_endpoint}" \
  --set "settings.interruptionQueue=${karpenter_queue}" \
  --set "controller.image.repository=${image_repo}" \
  --set "controller.image.tag=karpenter-controller-${version_tag}" \
  --set "controller.image.digest=${version_digest}" \
  --set controller.resources.requests.cpu=1 \
  --set controller.resources.requests.memory=1Gi \
  --set controller.resources.limits.cpu=1 \
  --set controller.resources.limits.memory=1Gi \
  --wait

```

- Installs (or upgrades) Karpenter using **Helm**.
- Configures it with:
  - The **IAM Role**
  - The **Cluster API endpoint**
  - The **SQS queue**
  - The **container image**
  - CPU & memory limits

✅ **Why?** Helm simplifies managing Karpenter.

---

11. Install/Upgrade EC2NodeClass and NodePool

```
helm upgrade --install karpenter-ec2nodeclass  karpenter_charts/EC2NodeClass --set "karpenter_node_iam_role_name=${karpenter_role_name}"
helm upgrade --install karpenter-nodepool  karpenter_charts/NodePool

```

- **EC2NodeClass** defines what type of EC2 instances Karpenter can provision.
- **NodePool** defines scaling rules.

✅ **Why?** These are **essential components** for managing EC2 nodes in Kubernetes.

---

12. Verify the Installation

```
helm list -A --time-format "2006-01-02 15:04"

```

- Lists installed Helm charts.

✅ **Why?** Confirms whether **Karpenter was successfully installed**.

What Does This Script Achieve?

✅ **Automates the installation & upgrade** of Karpenter.
✅ **Ensures AWS dependencies (IAM, SQS, EKS) are set up**.
✅ **Deploys Karpenter with proper configurations for Kubernetes autoscaling**.
✅ **Allows easy version updates** using Helm.

Explanation of Your EC2NodeClass Configuration

This **EC2NodeClass** resource defines how Karpenter provisions **EC2 instances** in your Amazon EKS cluster. It controls **AMI selection, storage, networking, security groups, and user data configuration**.

Step-by-Step Breakdown

1. Resource Type and API Version

```
apiVersion: karpenter.k8s.aws/v1beta1
kind: EC2NodeClass
metadata:
  name: default

```

- **apiVersion:** `karpenter.k8s.aws/v1beta1` → This is a Karpenter-specific **CRD (Custom Resource Definition)**.
- **kind:** `EC2NodeClass` → Defines how Karpenter provisions new EC2 instances.
- **metadata.name:** `default` → This NodeClass is named **default**, meaning it may be referenced by NodePools.

✅ **Why?** This **NodeClass** acts as a blueprint for provisioning nodes.

---

2. Specify the AMI (Amazon Machine Image)

```
spec:
  amiFamily: AL2

```

- **AL2** refers to **Amazon Linux 2**.

- Karpenter automatically selects an Amazon-provided **optimized AMI**.

✅ **Why?** Ensures nodes are using a stable and optimized **Kubernetes-compatible** OS.

---

3. Custom User Data (Startup Script)

```
userData: |
  #!/bin/bash
  fallocate -l 32G /swapfile
  chmod 600 /swapfile
  mkswap /swapfile
  swapon /swapfile
  sysctl -w vm.swappiness=10
  jq '.failSwapOn=false' /etc/kubernetes/kubelet/kubelet-config.json | jq '.featureGates.NodeSwap=true' | jq '.memorySwap.swapBehavior="UnlimitedSwap"' > /etc/kubernetes/kubelet/kubelet-config.json.new
  cp -rf /etc/kubernetes/kubelet/kubelet-config.json /etc/kubernetes/kubelet/kubelet-config.json.org
  cp -rf /etc/kubernetes/kubelet/kubelet-config.json.new /etc/kubernetes/kubelet/kubelet-config.json

```

This **userData** script runs when the instance starts:

1. **Creates a 32GB Swap File**

   ```
   fallocate -l 32G /swapfile
   chmod 600 /swapfile
   mkswap /swapfile
   swapon /swapfile
   sysctl -w vm.swappiness=10

   ```

   - **Creates a 32GB swap file** to expand virtual memory.
   - **Sets** `swappiness=10` to reduce aggressive swap usage.

2. **Enables Kubernetes NodeSwap**

   ```
   jq '.failSwapOn=false' /etc/kubernetes/kubelet/kubelet-config.json | jq '.featureGates.NodeSwap=true' | jq '.memorySwap.swapBehavior="UnlimitedSwap"' > /etc/kubernetes/kubelet/kubelet-config.json.new

   ```

   - Modifies Kubelet’s configuration to **allow swap memory**.
   - Enables the `NodeSwap` feature gate.
   - Sets **unlimited swap behavior**.

✅ **Why?** This ensures nodes **do not fail due to memory pressure** and allows swap usage.

---

4. IAM Role for Node Instances

```
role: {{ .Values.karpenter_node_iam_role_name }}

```

- Uses the IAM role defined in Helm (`{{ .Values.karpenter_node_iam_role_name }}`).
- The role allows EC2 instances to join the EKS cluster and interact with AWS services.

✅ **Why?** Ensures that nodes **inherit IAM permissions** needed for Kubernetes operations.

---

5. Define EC2 Instance Storage

```
blockDeviceMappings:
  - deviceName: /dev/xvda
    ebs:
      volumeSize: {{ .Values.ec2_node_class_volume_size }}
      volumeType: {{ .Values.ec2_node_class_volume_type }}
      iops: {{ .Values.ec2_node_class_volume_iops }}
      encrypted: true
      deleteOnTermination: true
      throughput: {{ .Values.ec2_node_class_throughput }}

```

- **Device Name** `/dev/xvda` → This is the primary volume.
- **Volume Size** → `{{ .Values.ec2_node_class_volume_size }}` (Defined in Helm).
- **Volume Type** → `{{ .Values.ec2_node_class_volume_type }}` (e.g., `gp3` or `io1`).
- **IOPS (Input/Output Operations Per Second)** → `{{ .Values.ec2_node_class_volume_iops }}` .
- **Throughput** → `{{ .Values.ec2_node_class_throughput }}` (Important for `gp3` volumes).
- **Encrypted:** `true` → Ensures data security.
- **Delete on Termination:** `true` → Deletes the volume when the instance is terminated.

✅ **Why?** Controls **storage configuration**, ensuring performance and security.

---

6. Select AWS Subnets for Node Deployment

```
subnetSelectorTerms:
  - tags:
      karpenter.sh/discovery: eks

```

- Selects **subnets** with the tag `karpenter.sh/discovery: eks`.

✅ **Why?** Ensures instances **launch in the correct VPC subnets**.

---

7. Assign Security Groups to Nodes

```
securityGroupSelectorTerms:
  - tags:
      karpenter.sh/discovery: eks

```

- Selects **security groups** based on the same `karpenter.sh/discovery: eks` tag.

✅ **Why?** Ensures worker nodes **inherit the correct security settings**.

---

8. Apply Tags to Instances

```
tags:
  karpenter.sh/discovery: eks

```

- Tags all created instances with `karpenter.sh/discovery: eks`.

✅ **Why?** Helps Karpenter **track and manage instances**.

Summary: What Does This EC2NodeClass Do?

Feature	Description
AMI Selection	Uses Amazon Linux 2 (AL2).
Startup Script	Creates a **32GB swap file** and enables **NodeSwap**.
IAM Role	Uses a dynamically assigned IAM role for worker nodes.
EBS Storage	Defines **storage size, type, IOPS, throughput, and encryption**.
Subnet Selection	Chooses subnets with the tag `karpenter.sh/discovery: eks`.
Security Groups	Assigns security groups with the tag `karpenter.sh/discovery: eks`.
Tags	Labels EC2 instances for tracking.

Why is This EC2NodeClass Important?

🔹 **Controls how Karpenter provisions EC2 nodes**.
🔹 **Ensures nodes have the right IAM, networking, and storage settings**.
🔹 **Improves performance with swap memory and storage optimizations**.
🔹 **Allows Kubernetes workloads to scale efficiently**.

Explanation of Your Karpenter NodePool Configuration

This **NodePool** resource defines **how Karpenter provisions and manages EC2 instances** in your Kubernetes cluster. It controls **which instance types are allowed, where nodes can be created, autoscaling limits, and node lifecycle policies**.

Step-by-Step Breakdown

1. Define Resource Type and Metadata

```
apiVersion: karpenter.sh/v1beta1
kind: NodePool
metadata:
  name: default

```

- **apiVersion:** `karpenter.sh/v1beta1` → This is a Karpenter-specific **CRD (Custom Resource Definition)**.
- **kind:** `NodePool` → Defines a **scaling policy** for Kubernetes nodes.
- **metadata.name:** `default` → The name of this NodePool is **default**.

✅ **Why?** This **NodePool** tells Karpenter how to provision EC2 nodes.

---

2. Define Node Template (How Nodes Are Created)

```
spec:
  template:
    metadata:
      labels:
        created_by: karpenter

```

- **Labels** → Assigns the label `created_by: karpenter` to each node.

✅ **Why?** This makes it **easy to filter** and manage Karpenter-created nodes.

---

3. Associate with an EC2NodeClass

```
    spec:
      nodeClassRef:
        name: default

```

- **nodeClassRef:** `default` → Links this **NodePool** to an **EC2NodeClass** named `default`.

✅ **Why?** The **EC2NodeClass** defines AMI, storage, IAM roles, and network settings.

---

4. Define Node Scheduling Constraints

```
      requirements:
        - key: "node.kubernetes.io/instance-type"
          operator: In
          values: {{ .Values.karpenter_instance_types | toJson }}

```

- **Instance Type Constraint**
  - This restricts Karpenter to **specific EC2 instance types**.
  - `{{ .Values.karpenter_instance_types | toJson }}` dynamically pulls values from Helm.

Example (after Helm resolves values):

```
      requirements:
        - key: "node.kubernetes.io/instance-type"
          operator: In
          values: ["t3.medium", "m5.large", "c5.xlarge"]

```

✅ **Why?** Ensures that **only approved instance types** are used.

```
        - key: "topology.kubernetes.io/zone"
          operator: In
          values: {{ .Values.azs | toJson }}

```

- **Availability Zone Constraint**
  - Restricts nodes to **specific AWS Availability Zones**.
  - Example values after Helm resolves:

    ```
    values: ["us-east-1a", "us-east-1b", "us-east-1c"]

    ```

✅ **Why?** Ensures workloads **only run in specified zones**.

```
        - key: "karpenter.sh/capacity-type"
          operator: In
          values: {{ .Values.karpenter_instance_capacity_types | toJson }}

```

- **Capacity Type Constraint**
  - Controls whether Karpenter uses:
    - **"on-demand"** instances.
    - **"spot"** instances (cheaper but can be terminated anytime).
  - Example values after Helm resolves:

    ```
    values: ["on-demand", "spot"]
    ```

```

```

✅ **Why?** Allows cost-optimization by mixing **Spot and On-Demand instances**.

---

5. Set CPU Limits for Auto-Scaling

```
limits:
  cpu: 1000

```

- **Limits total vCPUs across all nodes** managed by this **NodePool**.
- **1000 CPU cores** → Limits the maximum cluster size.

✅ **Why?** Prevents **uncontrolled auto-scaling** from over-provisioning nodes.

---

6. Define Node Disruption and Consolidation Policies

```
disruption:
  consolidationPolicy: WhenEmpty
  consolidateAfter: 30s
  expireAfter: 'Never'

```

- `consolidationPolicy: WhenEmpty` →
  - Karpenter **removes underutilized nodes only when they are empty**.
- `consolidateAfter: 30s` →
  - Karpenter checks for **unused nodes every 30 seconds**.
- `expireAfter: 'Never'` →
  - Nodes never expire unless scaled down manually or consolidated.

✅ **Why?** Prevents **unnecessary node deletions** while still optimizing resource usage.

Summary: What Does This NodePool Do?

Feature	Description
Instance Type Selection	Restricts instances to approved types.
Availability Zones	Ensures nodes are provisioned only in selected AZs.
Capacity Type	Supports **Spot** and **On-Demand** instances.
Max CPU Limit	Prevents excessive node scaling beyond **1000 CPUs**.
Consolidation	Automatically removes **empty** nodes after **30 seconds**.
Expiration	Nodes **never expire** automatically.

Why is This NodePool Important?

🔹 **Controls auto-scaling behavior** for Kubernetes worker nodes.
🔹 **Optimizes costs** by allowing a mix of **Spot** and **On-Demand** instances.
🔹 **Improves scheduling** by defining **which instance types and AZs** are allowed.
🔹 **Reduces waste** by consolidating **idle nodes** after 30s.

ECR lifecycle policies for unused images

AIM: remove unused images from the ECR

- **PRODUCTION_IMAGE_TTL = 90 (Giddy_… , Felix_…)**
- **FB_IMAGE_TTL = 60 (*_CR_FB_ *)**
- **MASTER_IMAGE_TTL = 60 (master) - step 2.1**
- **PRODUCTION_IMAGE_WITH_GA_BUILD -> Not deleting GA images (Were released as GA) - step 2.4**
- **UNKOWN_IMAGE_PATTERN_TTL = 90 (??, need to take a look at the code)**

Step 1: create a list of all current image in the ECR

Skybox’s ECR is residing in “**skybox-shared-services**“ account.

There are a few jobs that update the images that are in the ECR:

- Build_Saas_Agents - https://jenkins-srv/job/Build_Saas_Agents/ - MySQL, Elastic, Artemis, Cloudwatch, Datadog
- non_production_build - https://jenkins-srv/job/non_production_build/

- Version’s pipeline - https://jenkins-srv/job/Giddy/ - Giddty, felix..

I created a script in devopstols/aws_scripts/shared_account_utils/get_all_ecr_images.sh that returns a list of all the images that are in all the repositories within the ECR.

Step 2: handle all “do-not-delete” images

Step 2.1: life cycle policies

Each repository within the ECR has its own life cycle policy (see end of section for the policies as of 11-feb-2025).

The lifecycle policies handle images with the prefixes - `[SKY-]` `,[master_],[cherry-pick-][collector-base],[elastic-SKY-]`, `[mysql-SKY-]`, `[artemis-SKY-]`, `[cloudwatch-SKY-]`, `[datadog-SKY-],[elastic-master_]`, `[mysql-master_]`, `[artemis-master_]`, `[cloudwatch-master_]`, `[datadog-master_]`

Step 2.2: create a “white list” of all currently used by an ECS images

We need to handle the images that are currently in production. To do this we need to create a “white list” of all images that are currently being used by an ECS of an account.

To see image used by ECS = AWS → ECS → task definition

I created a script in devopstols/aws_scripts/shared_account_utils/get_images_used_by_ecs.sh that returns a list of all images that are being used by an ECS (this needs to run for each account).

Step 2.3: create a “white list” of all currently used by an EKS images

f

Step 2.4: get the list of GA versions

GA images are not to be deleted!

Step 3: create a list of images to delete

In step 2.1 we got a list of prefixes that we can ignire, as the life cycle policy handles them. In step 2.2 we got a white list of images that are not to be deleted as they are currently in use. and so, we need to subtract the images from step 2 from the list of all images within the ECR that we got in step 1.

Repositories with No Lifecycle Policy

- `development`
- `ecr-public/eks-distro/kubernetes/pause`
- `skyboxdev/app`
- `skyboxdev/services/service-check-access`
- `skyboxreleases/app`
- `skyboxreleases/collector`
- `skyboxreleases/server`

Repositories with Lifecycle Policies

Short-lived images (1-day expiry)

- `skyboxdev/services/service-vtm-service`
- `skyboxdev/services/service-model-service`
- `skyboxdev/services/service-hello-world`
- `skyboxdev/services/service-conga`
  - **Rule 1**: Expire **all images** after **1 day**

SkyboxDev Server

- **Rule 1**: Expire **untagged images** after **1 day**
- **Rule 2**: Expire **tagged images** with prefix `[SKY-]` after **14 days**
- **Rule 3**: Expire **tagged images** with prefix `[master_]` after **14 days**
- **Rule 4**: Expire **tagged images** with prefix `[cherry-pick-]` after **1 day**

SkyboxDev Collector

- **Rule 1**: Expire **untagged images** after **1 day**
- **Rule 2**: Expire **tagged images** with prefix `[SKY-]` after **7 days**
- **Rule 3**: Expire **tagged images** with prefix `[master_]` after **14 days**
- **Rule 4**: Expire **tagged images** with prefix `[cherry-pick-]` after **1 day**

Infra

- **Rule 1**: Expire **untagged images** after **1 day**
- **Rule 2**: Expire **tagged images** with prefix `[collector-base]` after **1 day**

DevOps Agent

- **Rule 1**: Expire **untagged images** after **1 day**

- **Rules 2-6**: Expire **tagged images** with prefix `[elastic-SKY-]`, `[mysql-SKY-]`, `[artemis-SKY-]`, `[cloudwatch-SKY-]`, `[datadog-SKY-]` after **14 days**
- **Rules 7-11**: Expire **tagged images** with prefix `[elastic-master_]`, `[mysql-master_]`, `[artemis-master_]`, `[cloudwatch-master_]`, `[datadog-master_]` after **14 days**

Deploying with Terraform across an account

- change the neccassry files in mt-deployment/. for example:
  - variables.tf - insert within the map “`bash_scripts_to_s3`“ an additional bash script
  - ssm.tf - create a new SSM document
- Run the pipeline named “aws_eks_account_provisioning” **for each wanted account** (no way to preform this action on multiple accounts for safety reasons)
  - The pipeline will wait for you to approve the plan.json (can be found in the archived artifacts of the specific Jenkins job). Visualize the file through https://sb-tfui and check:
    - If “cluser_creation” is showing go to AWS access portal → account → glovbal/admin credentials → EKS → access → role-global-terraform → delete
    - If anything related to network shows know you will have to re-run the pipeline until they wont show (the exception is - module.tgw_attachment.aws_ec2_transit_gateway_vpc_attachment.local_account)
  - Re-run the pipeline until you don’t see updates/creations that are not wanted and/or no resources added.

Delete PVC in multi-tenant AWS EKS - elasticsearch + mySQL + data

AIM: delete resources connected to a stateful set (mySQL/elasticsearch/data)

Each tenant has its own stateful set, with replica set to 1.

connection to AWS -activated by “session_to_eks.sh“ (creates an SSH tunnel through a bastion EC2 instance, bu using an AWS document called 'port_forward_to_remote_document')

Pipeline - aws_mt_wipedb

- In jenkins
  - create a new item by copying “aws_eks_tenant_resize“ How To Clone a Jenkins Job
  - remove unnecessary parameters by clicking the “configure” button and scrolling down until you reach an unwanted button.
  - if you need to add new parameters How Do I Add a Choice Parameter in Jenkins?
- The Jenkinfile
  - based on aws_eks_tenant_resize/Jenkinsfile

Script - aws_mt_wipedb.sh

- stop stateful set by lowering replica to zero
- delete stateful set
- delete pvc
  - mySQL = data-`data-elastic-skybox-0`
  - elasticsearch = `data-mysql-skybox-0`
  - data = `skybox-data-skybox-0`
- delete files from the bastion of the customer’s cluster. In path “cd /efs/customers/<tenant name>” search the latest “confx” directory and delete the following files:
  - for wipe_db

  -
    ```
    rm -f \"\$latest_confx/*.pem\"
    rm -f \"\$latest_confx/*.cksum\"
    rm -f \"\$latest_confx/*.p12\"
    ```
  - for elastic search

    ```
    rm -f \"\$latest_confx/server.skybox.elastic.crt\"
    rm -f \"\$latest_confx/server.skybox.elastic.key\"
    ```
- recreate stateful - by calling pipeline “aws_eks_tenant_provisioning_update“)
  - search for usage examples in devopstool, if not found - Getting started with Pipeline
  - Snippet Generator - steps for a Scripted Pipeline | `steps` block in a `stage` in a Declarative Pipeline
  - Declarative Directive Generator - sections and directives used to define a Declarative Pipeline.

Test the pipeline

- AWS access portal - access with “skybox-internal-devops03s-dev“ to “skybox-internal-devops03s-dev”
  - make sure you are at one of the following:
    - N. virginia - us-east-1
    - singapore - ap-southeast-1
    - frankfort - eu-central-1
  - EC2 - micro is the bastion that delete_confx_files.sh accesses to delete file -within the instance
    - accese k9 = /efs/k9s
    - healthcheck = bash -x /healthcheck.sh
- https://sb-consul/ui/dc1/kv/aws/ou/non-production-rnd/jenkins_multi_tenant/003857080240/devopscust01/skybox_public_url/edit - the url for devopscust01 within the multi tennat of devops3
  - ensure you can login to it

- username = saasadmin
- password = AWS Systems Manager → Parameter Store → /devopscust01/production_saasadmin_password