Tech Task WP07 Jenkins aka CloudBees CI by Computacenter

Requirements and analysis

Jenkins instance setup and security management on Kubernetes

Objective:

To set up a Jenkins instance with dummy jobs on Kubernetes, back it up, and handle a security breach scenario using Terraform, Ansible, or similar tools. The deployment will be done in Azure using AKS (Azure Kubernetes Service).

1. Create a Jenkins Instance with Dummy Jobs on Kubernetes

☒ Set up an AKS cluster in Azure.
  - AKS setup README

☒ Deploy a Jenkins instance on the AKS cluster.
  - Argo CD setup README
  - CloudBees

☒ Create several dummy jobs in Jenkins to simulate a real environment.

2. Backup the Instance and Validate the Backup

☒ Implement a backup strategy for the Jenkins instance.
☒ Validate that the backup is complete and can be restored successfully.

3. Security Breach Scenario: Master Key Exposure

☒ Make the Instance Unavailable to Others
  ☒ Scale down the Jenkins deployment to zero replicas.
  ☒ Alternatively, remove the route to the Jenkins service or block access using a proxy.

☒ Rotate the Master Key
  ☒ Generate a new master key for Jenkins.
  ☒ Update the Jenkins configuration with the new master key.

☒ Re-encrypt All Credentials
  ☒ Re-encrypt all stored credentials in Jenkins using the new master key.

☒ Make the Instance Available to Others
  ☒ Scale the Jenkins deployment back up.
  ☒ Restore the route to the Jenkins service or unblock access via the proxy.

☒ Validate that the Jobs are Still Working
  ☒ Ensure all dummy jobs are functioning correctly after the security changes.

Our apporch:

Computacenter decided to pursue a solution using the enterprise version of CloudBees CI instead of open-source Jenkins. We are confident that this solution

can fulfill these and many other requirements more effectively, and we would like to demonstrate this through this task. Our goal is to reduce implementation and operational effort while increasing security by leveraging the features of an enterprise solution.

CloudBees CI

CloudBees CI is a fully-featured, cloud native capability that can be hosted on-premise or in the public cloud used to deliver CI at scale. It provides a shared, centrally managed, self-service experience for all your development teams running Jenkins. CloudBees CI on modern cloud platforms is designed to run on Kubernetes. CloudBees CI on traditional platforms has been developed for on-premise installations. I includes the following features:

- **Jenkins (HA-setup):** CloudBees CI provides built-in high-availability (HA) configurations, reducing the complexity and effort required to maintain a resilient Jenkins environment.
- **On-prem and cloud-based setup:** CloudBees CI seamlessly integrates with both on-premises and cloud environments, simplifying the management and scalability of hybrid setups.
- **Jenkins instances interconnection and workload shifting:** The platform allows for easier interconnection of Jenkins instances and automated workload shifting, ensuring optimal performance and resource utilization.
- **Plugin management, approval, lifecycle:** CloudBees CI automates plugin lifecycle management, from approval to updates, significantly reducing manual intervention and potential errors.
- **Feature implementation:** With CloudBees CI, new features can be implemented more quickly and reliably, supported by enterprise-level tools and best practices that minimize downtime.
- **Technical meetings with tool supplier:** The comprehensive support provided by CloudBees CI reduces the need for frequent technical meetings, as issues are resolved more efficiently through the platform's robust features.
- **Data synchronization:** CloudBees CI includes advanced data synchronization tools that ensure consistency across all Jenkins instances, improving data integrity and reducing manual synchronization efforts.
- **Archiving CSD, class, etc.:** Automated archiving features within CloudBees CI help maintain compliance and data retention standards with minimal manual intervention.
- **Backup & disaster recovery:** CloudBees CI's integrated backup and disaster recovery capabilities offer enhanced protection and quick recovery options, reducing the risk of data loss and service interruptions.

- **3rd level support:** The enterprise support provided by CloudBees CI, combined with our certified experts, minimizes the need for extensive third-level support, ensuring faster resolution of issues.

For a detailed comparison of enterprise CloudBees CI features versus Open Source Jenkins, please refer to CB-CI-Jenkins-Comparison-2023.pdf.

1. Create a Jenkins Instance with Dummy Jobs on Kubernetes

Our approach is to provide **everything as code** and **automate** as much as possible. For this task, we chose to set up the Azure infrastructure (AKS) using Terraform. We decided to adopt Argo CD for the central deployment of the CloudBees Operations Center. The CloudBees controllers (Jenkins instances for the teams) can be automated or provisioned with a mouse click. However, we also wanted to take the GitOps approach for the CloudBees Operations Center.

The following diagram illustrates the architecture and how the controllers (Jenkins team instance/instances) are provisioned, configured, and managed by the CloudBees Operations Center.

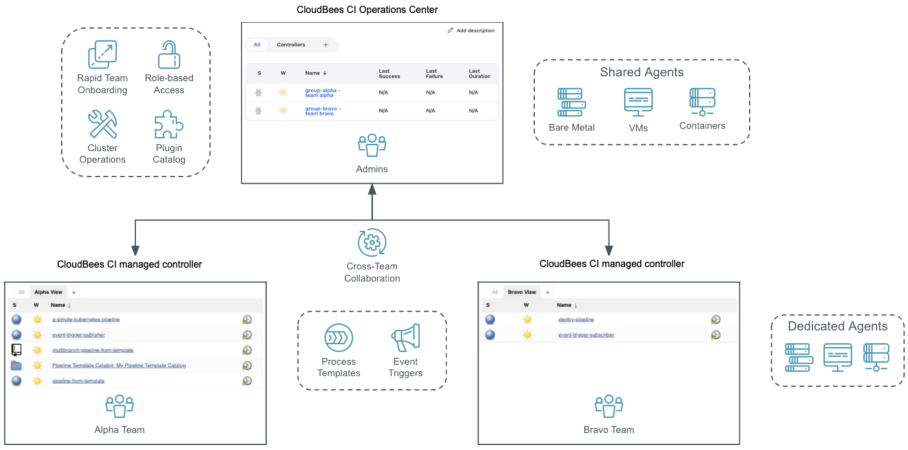


Figure 1. Distributed build environment with CloudBees CI on modern cloud platforms

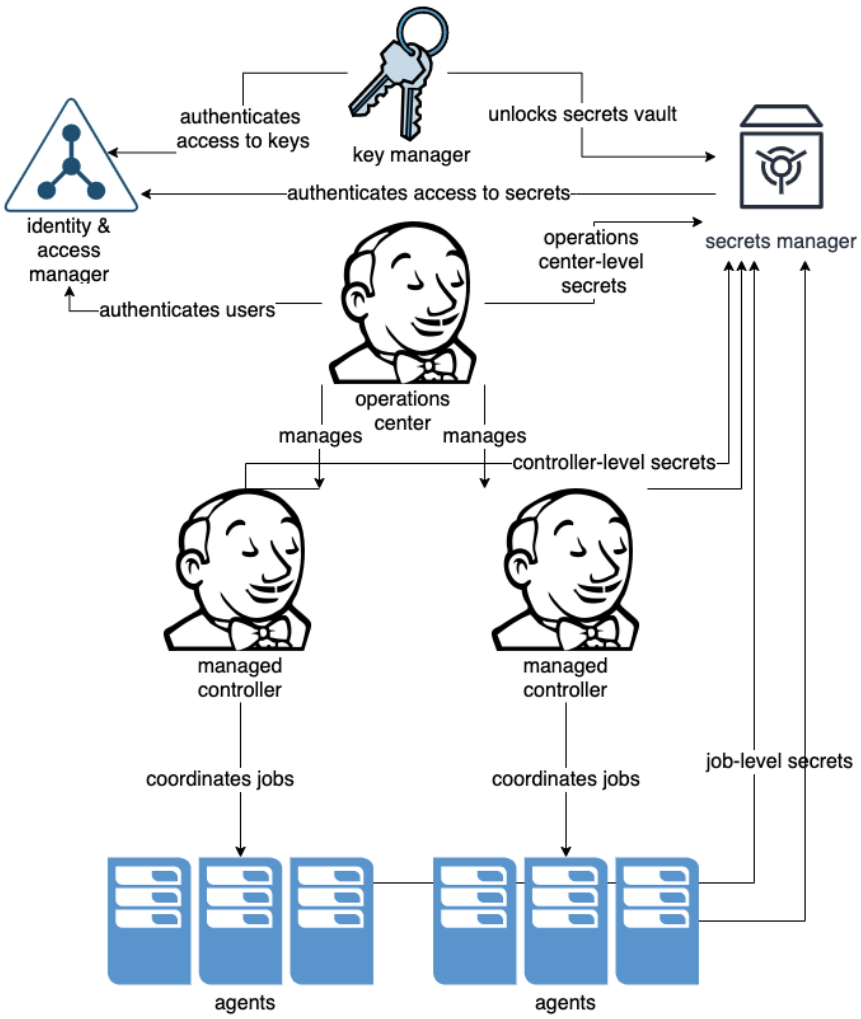
Many software systems use secret data for authentication, such as user passwords, secret keys, access tokens, and certificates. These can, of course, be stored in a Jenkins Credentials Store. However, this means that if the `master.key` is stolen, all credentials are directly compromised.

Our approach was to use an **Azure Key Vault** to manage important credentials that can be accessed via an **Azure Service Principal**. This Azure Service Principal is granted read-only rights and is accessible only from the controller. It is stored in a Kubernetes secret an loaded during controller provisioning.

Although this was not requested in the technical task, we implemented it with

minimal effort. To meet the task requirements (even though we believe they pose a significant security risk), we, for example, store a simple secret in the Jenkins Credentials Store of the respective controller.

Below is an example image of this setup, which we partially implemented.



Due to time and resource constraints, and since it was not a requirement of the technical task, we did not build this in a High Availability (active/active) configuration. Since CloudBees supports this out-of-the-box, it would be possible in this setup with minimal effort. More information can be found here: High Availability (active/active) documentation.

2. Backup the Instance and Validate the Backup

Backing up CloudBees CI involves taking care of Jenkins home directories, configurations, and other important data. Once the backup is completed, validating it by testing restoration in a separate environment ensures that the backup is reliable and can be used in a real disaster recovery situation.

CloudBees provides a Backup Plugin that can automate regular backups of the Jenkins home directory and other necessary files. Open Source Jenkins doesn't provide that plugin based backup approach.

The following supported way was implemented by us. You can all find details here: https://docs.cloudbees.com/docs/cloudbees-ci/latest/backup-restore/cloudbees-backup-plugin https://docs.cloudbees.com/docs/cloudbees-ci/latest/backup-restore/restoring-from-backup-plugin https://docs.cloudbees.com/docs/cloudbees-ci/latest/backup-restore/kubernetes

Implement a backup strategy for the Jenkins instance. As described above, we follow the recommended and supported path. We have set up a 'Backup & Restore' job. In our case, we performed a backup 'every hour'. (This can be customised from use case to use case. Even before each build, for example).

To store the backups, we use Azure Blob Storage where access is carried out via credential.

This 'Backup & Restore' is stored as code and automatically forced and rolled out on all controllers (Jenkins Team Instants). Together with the pre-configured key vault integration, the restore of the backup can be done with one click after a controller is provisioned.

Our "Backup & Restore" folder as code:

```
kind: folder
name: Backup and Restore
description: ''
displayName: Backup and Restore
items:
- kind: backupAndRestore
  name: Backup
  blockBuildWhenDownstreamBuilding: false
  blockBuildWhenUpstreamBuilding: false
  buildersList:
  - backupBuilder:
      subjects:
      - buildRecordSubject: {
          }
      - jobConfigurationSubject: {
          }
      - systemConfigurationSubject:
```

```
          omitMasterKey: true
      format:
        zipFormat: {
        }
      exclusive: false
      store:
        azureBlobStorageStore:
          folder: ''
          accountName: cloudbees
          containerName: cloudbeesbackup
          credentialsId: BackupStorage
          blobEndPointURL: ''
          useMetadata: false
      retentionPolicy:
        upToNRetentionPolicy:
          n: 3
      safeDelaySeconds: 0
  concurrentBuild: false
  description: ''
  disabled: false
  displayName: Backup
  triggers:
  - cron:
      spec: '@hourly'
- kind: backupAndRestore
  name: Restore
  blockBuildWhenDownstreamBuilding: false
  blockBuildWhenUpstreamBuilding: false
  buildersList:
  - restoreBuilder:
      ignoreConfirmationFile: true
      preserveJenkinsHome: false
      ignoreDigestCheck: false
      store:
        azureBlobStorageStore:
          folder: ''
          accountName: cloudbees
          containerName: cloudbeesbackup
          credentialsId: BackupStorage
          blobEndPointURL: ''
          useMetadata: false
      restoreDirectory: ''
  concurrentBuild: false
  description: ''
  disabled: false
  displayName: Restore
```

```
properties:
- envVars: {
    }
- itemRestrictions:
    filter: false
- folderCredentialsProperty:
    folderCredentials:
    - credentials:
      - secretStringCredentials:
          description: ''
          id: XXXXXXXXXXX-ba1e-438a-9c81-XXXXXXXXXX
          secretIdentifier: https://XXXXXXXXXXX.vault.azure.net/secrets/BackupStorage/XXXX
      domain: {
        }
```

Validate that the backup is complete and can be restored successfully. Since the backup and restore process is performed by the supported CloudBees plugin and is always visible through the jobs in the folder, we can be sure that if the backup job has run and completed successfully, the backup is successful. The same applies to the restore process.

We have tested this several times on our AKS setup. And it is a simple task:

1) Activate backup in the pipeline manually or it is executed every hour as we do
2) To restore, simply execute the 'Restore' job. In our case, the last backup is used.
3) Restart the controller and you're done.
4) (To proof it you can run the automatically deployed verification pipeline (CasC) which is connecting to the azure key vault and reading a dummy secret.)

The backup restore job itself is verifying the integrity of the backup during restore by validating a checksum.

Excluding files from a backup job is also possible. Is is important for the next task "Security Breach Scenario: Master Key Exposure".

https://docs.cloudbees.com/docs/cloudbees-ci/latest/backup-restore/cloudbees-backup-plugin

3. Security Breach Scenario: Master Key Exposure

Make the Instance Unavailable to Others

Scale down the Jenkins deployment to zero replicas. It can be achieved using kubectl. This approach is quite similar to Open Source Jenkins.

```
# kubectl scale statefulset <NAME> --replicas=0 --n <xxxxxx>
$ kubectl get statefulset.apps -n cloudbeesci
NAME                    READY   AGE
cjoc                    1/1     40h
democontroller          1/1     17h
rsadowski-playground    1/1     23h
test                    1/1     46m

$ kubectl scale statefulset democontroller --replicas=0 --n cloudbeesci
```

Alternatively, remove the route to the Jenkins service or block access using a proxy. Depending on which forensic methods are to be used where, the following options can be carried out.

Option 1, the controller can be shutdown from the Operation Center (OC-> Manage Controller -> Deprovision). Option 2, the user permissions can be modified to deny access (using our RBAC capability) Option 3, the ingress to the controller can be modified by removing the destination path.

```
kubectl get ingress democontroller -n cloudbeesci -o yaml > ingress.yaml
# ... edit the destination path
# One-liner
kubectl get ingress democontroller -n cloudbeesci -o json \
  | jq 'del(.metadata.resourceVersion, .metadata.uid, .metadata.annotations["kubectl.kuberne
        | .spec.rules[].http.paths[].path = "/NEW-PATH-TO-NOWHERE/"' \
  | kubectl apply -f -
```

We will delete the controller anyway as it is no longer trustworthy.

Rotate the Master Key

The master key is usually rotated by deleting the `master.key` and restarting the controller. Here is our procedure.

Generate a new master key for Jenkins.

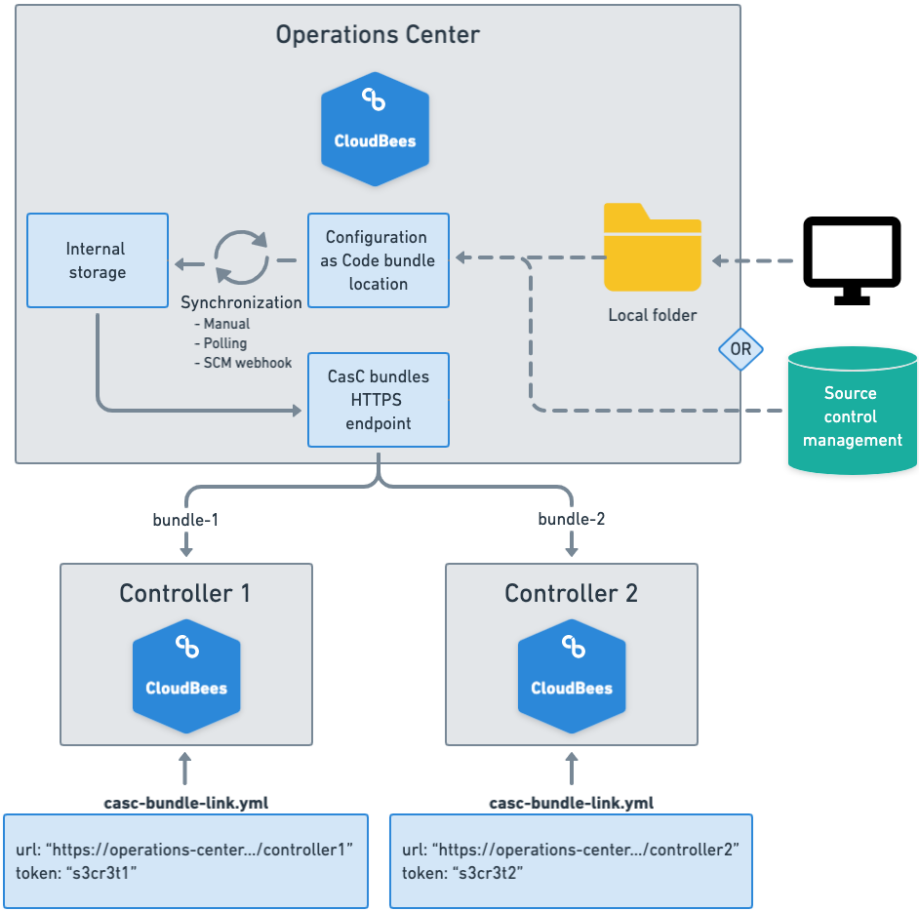
Update the Jenkins configuration with the new master key. First of all we delete the controller in the Operation Center by one click. With the second click we create a new Controller with the same name.

Because our blueprint for the controller is stored in the CasC (Configuration as Code for controllers) and this contains the backup & restore folder with the jobs and gets the Azure Service Principal Credentials from the kubernetes secret, we can restore the last non-compromised backup there!

If we know that the credentials from the Key Vault have also been compromised, they must be changed there. At least the service principal secret should be updated with a new password before starting the new controller.

> Configuration as Code for controllers simplifies the management of a CloudBees CI cluster by capturing the configuration of CloudBees CI controllers in human-readable declarative configuration files which can then be applied to a controller in a reproducible way.

In the diagram below you can see how the "CasC" works. The corresponding code for our example can be found at CloudBees Operation + CasC Helm Chart.



Re-encrypt All Credentials

Re-encrypt all stored credentials in Jenkins using the new master key. If a team has stored sensitive data in the credentials store on its controller, this will be restored during the restore process. However, the content is completely useless as it is read with the new `master.key`. This MUST be reset by the team. We have no automatic process here as it is assumed that all data has been compromised and will be reset accordingly. The old data should not and must not be used any more.

Make the Instance Available to Others

The instance was made available by creating a new controller instance per Operation Center in the previous step. The steps described here would scale up on the old stateful set, which was destroyed (since we deleted it in the Operation Center). This is correctly described in the paragraph just below this one.

```
# kubectl scale statefulset <NAME> --replicas=1 --n <xxxxxx>
$ kubectl get statefulset.apps -n cloudbeesci
NAME                    READY   AGE
cjoc                    1/1     40h
democontroller          1/1     17h
rsadowski-playground    1/1     23h
test                    1/1     46m

$ kubectl scale statefulset democontroller --replicas=1 --n cloudbeesci
```

Restore the route to the Jenkins service or unblock access via the proxy. We deleted the controller. This also automatically deleted the Ingress entry. When a new controller was created, it was recreated and should be accessible again.

Generally we recommend to use the LDAP or Microsoft Entra ID (formerly Azure Active Directory) integration and Role-based access control (RBAC) features provided by CloudBees on the Operations Center level. This, together with the concept of providing preconfigured controllers to the single teams, will ensure a secure setup, minimize the impact of security issues inside a single controller and keep administrative configuration out of the reach of attackers as well as enforcing mandatory security and stability configurations over the whole system.

Validate that the Jobs are Still Working

Ensure all dummy jobs are functioning correctly after the security changes. As described above, this is provided by CloudBees Backup & Restore Plugin. The method we tested was described above.

Subpages for All-in-One document (PDF)

Setting up Azure Kubernetes Service (AKS) with Terraform for Tech Task WP07

To set up the AKS cluster, we have decided to use the Terraform Module for deploying an AKS cluster and the following example: application_gateway_ingress.

We opted for this solution in order to quickly implement the AKS Cluster but not to compromise on best practices. The following is our approach. The corresponding Terraform code can be found in the aks folder.

Prerequisites

- Install Terraform
- Install the Azure CLI
- Ensure you have proper permissions in your Azure subscription (Owner or Contributor role)

Security Considerations

- **RBAC:** Enabling Role-Based Access Control (RBAC) for AKS to ensure proper access control.
- **Azure Active Directory (AAD):** Integrate AAD for secure authentication.
- **Premium Disks:** Use instance types that allow premium disks to improve performance and reliability.
- **Private Cluster (Optional):** Restrict access to the API server with a private cluster.
- **Azure Container Registry (ACR):** Attach ACR to ensure secure access to container images.
- **Network Policies:** Ensure network security between pods and external resources.

Initialize and Apply Terraform Configuration

```
terraform init
# Create a Service Principal (or you can skip this if you already have one)
SP=$(az ad sp create-for-rbac --name "wp07-couldbees-sp" --role contributor --scopes /subsc

# Extract the necessary values from the JSON output
ARM_CLIENT_ID=$(echo $SP | jq -r '.appId')
ARM_CLIENT_SECRET=$(echo $SP | jq -r '.password')
ARM_TENANT_ID=$(echo $SP | jq -r '.tenant')
ARM_SUBSCRIPTION_ID=$(az account show --query id -o tsv)

# Export the environment variables for the current session
export ARM_CLIENT_ID ARM_CLIENT_SECRET ARM_TENANT_ID ARM_SUBSCRIPTION_ID

# Output the variables to verify
echo "ARM_CLIENT_ID=$ARM_CLIENT_ID"
echo "ARM_CLIENT_SECRET=$ARM_CLIENT_SECRET"
echo "ARM_TENANT_ID=$ARM_TENANT_ID"
echo "ARM_SUBSCRIPTION_ID=$ARM_SUBSCRIPTION_ID"

terraform plan
terraform apply
```

Verification

To verify the setup:

1. **Access AKS Cluster**:

   Get credentials for your AKS cluster:

   ```
   az aks get-credentials --resource-group aks-resource-group --name aks-cluster
   kubectl get nodes
   ```

2. **Check Namespace and Storage Class**:

   ```
   kubectl get namespace
   kubectl describe storageclass default-storageclass
   ```

Enable AKS HTTP Application Routing via Azure CLI

In your cases, enabling add-ons like **HTTP Application Routing** for your AKS clusters is not possible directly through the used AKS Terraform module (see above). Due to this limitation, we will need to enable these add-ons via the **Azure CLI** after the AKS cluster has been created.

1. Enable HTTP Application Routing Add-on https://learn.microsoft.com/en-us/azure/aks/app-routing-migration

To enable the **HTTP Application Routing Add-on** for your AKS cluster, we run the following CLI command:

```
az aks approuting enable --resource-group <ResourceGroupName> --name <ClusterName>
```

Once this add-on is enabled, the `app-routing-system` namespace will be created, and an NGINX Ingress controller will be set up automatically to manage HTTP traffic routing.

If the corresponding class is set in the CloudBees operator, this ingress is automatically used. https://docs.cloudbees.com/docs/cloudbees-ci-kb/latest/cloudbees-ci-on-modern-cloud-platforms/deploy-a-dedicated-ingress-controller-for-external-communications

Create a static IP and DNS label with the Azure Kubernetes Service (AKS) load balancer https://learn.microsoft.com/en-us/azure/aks/static-ip

Key Configuration Details

- **Node Pool**: The node pool configuration specifies `Standard_D4s_v3` as the VM size, ensuring at least 2 CPUs and 4 GiB of memory per node, with support for premium disks.
- **RBAC & AAD**: RBAC is enabled with Azure Active Directory integration for enhanced authentication and authorization.
- **Azure Container Registry (ACR)**: The AKS cluster is integrated with an ACR to securely pull container images.

- **Storage Class**: A default storage class using Azure Premium disks is defined to support high-performance workloads like CloudBees CI.

ArgoCD

Setup

After having deployed your AKS cluster via Terraform, install ArgoCD via Helm and connect it to the ArgoCD repository.

Make sure you have the Helm CLI installed locally and that you can connect to your Kubernetes cluster via kubectl.

Run the following command to install ArgoCD:

```
helm install argo-cd oci://ghcr.io/argoproj/argo-helm/argo-cd --version 7.5.2 --namespace ar
```

You can monitor the installation via

```
kubectl -n argo-cd get pods -w
```

As soon as the installation is done, retreive your admin password using

```
kubectl get secret -n argo-cd argocd-initial-admin-secret -o jsonpath="{.data.password}" | b
```

and connect to ArgoCD via local port forward:

```
kubectl port-forward services/argo-cd-argocd-server -n argo-cd 8443:443
```

You can now access the instance via your browser: https://localhost:8443

Log in and create a repository in the settings. Connecting via a project based SSH key is recommended.

Afterwards, you can navigate to “Applications” and create a “New App”. Use the repository you just configured and make sure to point to the path `/argocd`.

Attention: Connecting and syncing your applications will also change ArgoCD to be available under the context path `/argocd/`. It will also create an ingress ressource so that ArgoCD is available publicly!

After the sync is done, you can connect to your CloudBees Jenkins Operations Center (CJOC) via the public domain and the context path `/cjoc/`