core • logic

Report

Guide for updating the Kubernetes cluster

Migrating from the 1.25 to 1.26 version
Pod Security Policy→ Pod Security Admission

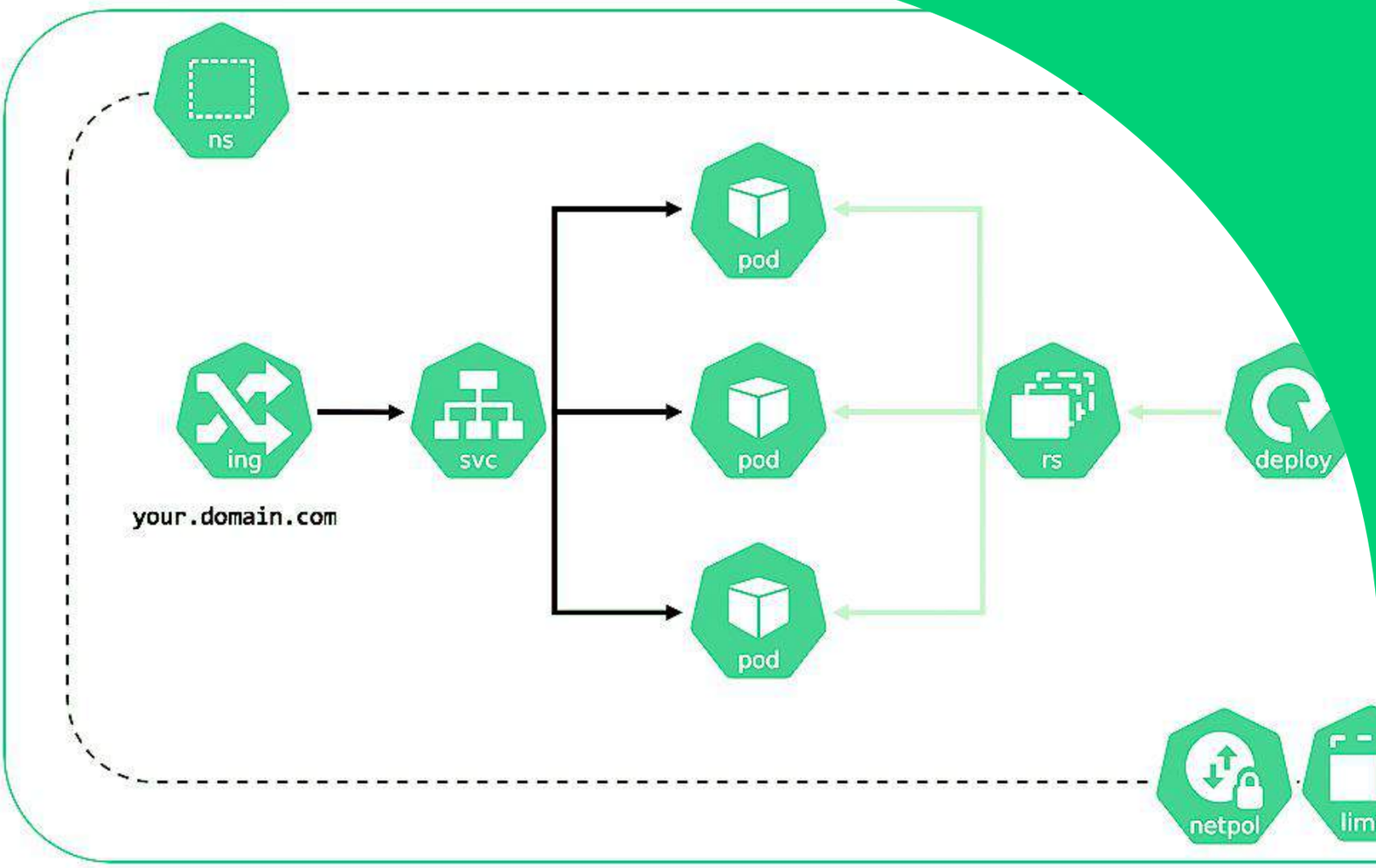


Table of contents

Table of contents [2/2]

Introduction

We present to you the report "Kubernetes cluster upgrade guide - Migrating from version 1.25 to version 1.26 - Pod Security Policy → Pod Security Admission".

Every evolution of technology brings with it challenges, but also new opportunities. The latest version of the Kubernetes cluster, 1.25, sees the departure of PodSecurityPolicy (PSP) in favor of a more flexible and modern method, called Pod Security Admission (PSA). This report discusses key steps to take before upgrading your cluster, focusing on services that have used PSP to date.

Konrad Matyas

Vice CEO of Core Logic

1. Updating services relying on PSP

At first, we present the key elements for the management and monitoring of the cluster when it comes to updates and compliance with the highest security standards. In any situation, it is always best to have the newest software versions, that have been optimized to comply with the highest security standards and to ensure the stability and reliability of the cluster.

1. Metallb – for the management of LoadBalancer

Metallb plays a key role in the management of LoadBalancer services within the cluster. Before the migration, you should have the latest version of Metallb, optimized for cooperation with Pod Security Admission.

2. Portworx – for storage management

When using Portworx for storage management, it is advised to have the latest version, fully compliant with the new security policy.

3. Prometheus Stack – for cluster monitoring

If you are using tools from the Prometheus stack for monitoring the cluster, you need to update Prometheus stack to the newest version, to avoid any issues related to security.

4. Blackbox Exporter -monitorowanie zdalnych usług

It is advised to check that blackbox-exporter – which is responsible for the monitoring of remote services – complies with the newest security standards.
You can get the list of the resources to update with the following command:

```
kubectl get podsecuritypolicies.policy -A
```

```
Warning: policy/v1beta1 PodSecurityPolicy is deprecated in v1.21+, unavailable in v1.25+
NAME                                                PRIV
blackbox-exporter-prometheus-blackbox-exporter-psp  false
controller                                          false
monitoring-grafana                                  false
monitoring-grafana-test                             false
monitoring-kube-prometheus-alertmanager             false
monitoring-kube-prometheus-operator                 false
monitoring-kube-prometheus-prometheus               false
monitoring-kube-state-metrics                       false
monitoring-prometheus-node-exporter                 false
px-operator                                         false
speaker                                             true
```

Where:

- px-operator = portworx,
- speaker, controller = metallb
- monitoring = prometheus
- blackbox-exporter = blackbox-exporter

2. Making a backup copy of the cluster

The process of making a backup is very similar to the updating process of Kubernetes (k8s) to the 1.24 version.

You will find below instructions on how to create a backup copy of the Kubernetes cluster, including a copy of the certificates, manifests, and ETCD data.

To secure the certificates and manifests from the Kubernetes cluster, you can use the rsync tool:

```
rsync -r --exclude=tmp user@host:/etc/kubernetes/ /home/user/k8s-backup/
```

The rsync tool allows you to copy folders from /etc/kubernetes/pki/ and /etc/kubernetes/manifests/ while preserving the structure of catalogs. Additionally, you can exclude useless folders, f.e. tmp. files.

Making a copy of ETCD

Before making a copy of ETCD, you need to confirm the location of the certificates and the endpoint, with the following command:

```
kubectl -n kube-system describe pod etcd-tst-traficar-cloud-master | grep -E 'listen-client-urls|cert-file|key-file|trusted-ca-file'
```

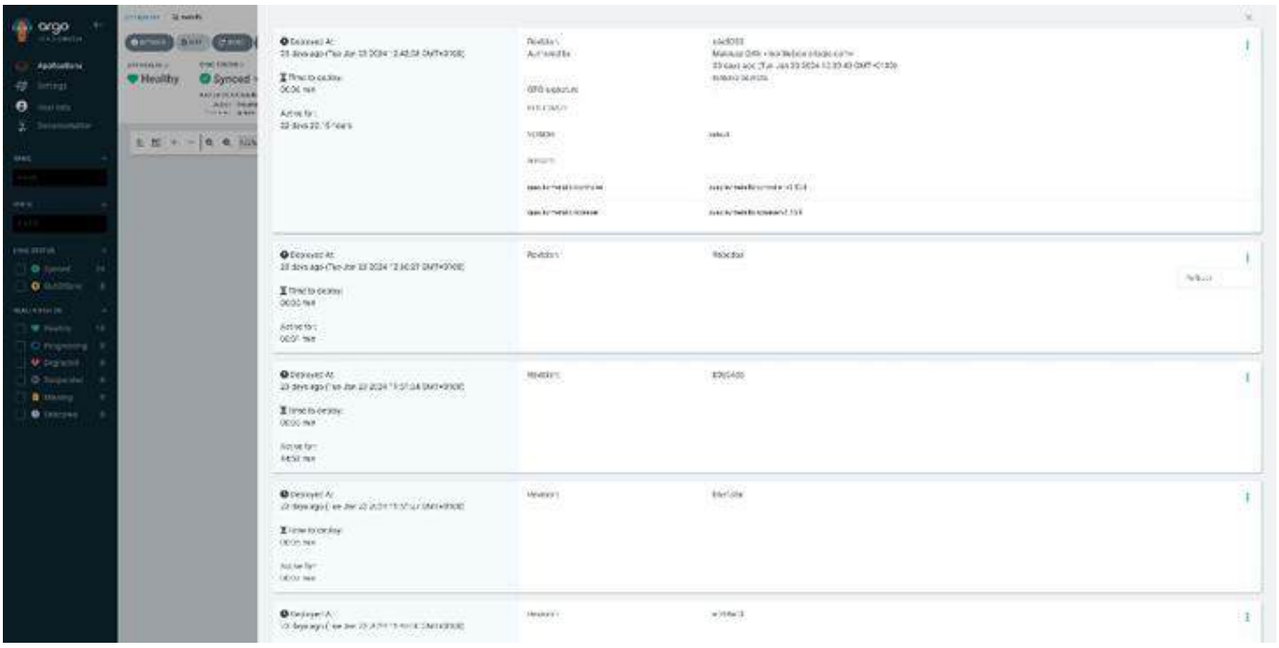
You can then make a backup copy of ETCD, with the following command:

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
snapshot save /opt/snapshot-pre-boot.db
```

The present commands will create an ETCD backup in the snapshot-pre-boot.db. file.

3. ArgoCD

ArgoCD is an essential tool for the safe and monitored upgrade of services in the Kubernetes cluster. Thanks to ArgoCD, it is possible to visually analyze the differences between the prepared manifest and the current state of the environment. But the biggest advantage is the flexible rollback option, allowing you to restore the previous version – making ArgoCD an irreplaceable tool when updating virtually any object of the cluster.



Setting up the application in ArgoCD

When configuring an application in ArgoCD, you should rely on the set of manifests defined in the cloud-gitops repository. While ArgoCD's panel offers the possibility of directly creating applications, the article focuses on the declaration-based approach in the repository, which allows consistent management of applications in different systems.

The structure of the application manifest

The declaration of the application should be adapted to the overall structure of the manifest, which will allow for an easier adaptation to specific services. In the present situation, it was assumed that the methodology used in this case would rely on the cloud-gitops repository.

3. ArgoCD

The declaration itself relies on the creation of a new file/manifest in the path

```
argocd/{{target_env}}/{{target_app_name}}.yml
```

The manifest should contain the following entry:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: {{target_app_name}}
  namespace: argocd
spec:
  destination:
    name: ''
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    path: {{target_app_path}}
    repoURL: {{target_app_repo}}
    targetRevision: {{target_app_repo_branch}}
  project: default

# {{target_app_name}}               - Nazwa aplikacji
# {{target_app_path}}               - Ścieżka do deklaracji / manifestu
# {{target_app_repo}}               - Link do repozytorium Git
# {{target_app_repo_branch}}        - Branch z powyższego repo
```

And this is the example declaration for Metallb prod:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: metallb
  namespace: argocd
spec:
  destination:
    name: ''
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    path: 'kustomize/metallb/overlays/prod'
    repoURL: 'https://repository.domain.com/project/gitops'
    targetRevision: master
  project: default
```

3. ArgoCD

It is advised to create new applications on separate branches in the GIT code repository. This approach allows to maintain the safety of other elements of the cluster, whenever changes are made in a given project. Each application should have its own developing environment, which will make it easier to control and monitor changes. In the situation where previous applications have been created in the cluster, it is advised to turn off the automatic synchronization (Auto-Sync).

It is also advised to make sure that Auto-Sync is turned off, or at least is turned off manually on the created branch. You can do so by changing the value of {{target_app_repo_branch}} in the application's definition.

It is then necessary to delete the entry:

```
syncPolicy:
   automated:
     prune: false
     selfHeal: true
```

After performing the necessary operations, you can synchronize the main ArgoCD application.

kzadlo@core-logic.com
www.core-logic.com

4. Initial operations before the implementation of MetalLB

Updating MetalLB in the ArgoCD context requires an appropriate setup of the cluster, especially when it comes to assigning and managing IP addresses. The following are key steps and important notes for the proper assigning of IP addresses before the implementation of MetalLB.

Proper assigning of IP addresses

Assigning IP addresses to services from the cluster, especially with ArgoCD, is not necessary. However, it is best to maintain the rigid configuration of IP addresses for the objects that require it.
It is worth verifying which services require constant assignment of IP addresses, and which can use dynamically assigned addresses. Before executing the next steps, you should verify if the cluster is ready to be updated. In the case of MetalLB, it is essential to declare the IP addresses, especially if they are permanently required.

Maintaining Stable IP Addresses

It is worth checking that services have their constant IP address assigned, especially for services making use of DNS or having declared fixed IP addresses on other services.

For the current cluster that we are updating, all the services did not necessarily have any assigned IP addresses before the update, which resulted in assigning random addresses from the pool. This can result in unexpected changes if services rely on the provided IP addresses.

We can verify that with the following command:

```
kubectl get svc -A -o custom-columns="NAME":.metadata.name,"TYPE":.spec.type,"CURRENT_IP":.status.loadBalancer
.ingress[0].ip,"REQUESTED_IP":.spec.loadBalancerIP | grep LoadBalancer
```


kzadlo@core-logic.com
www.core-logic.com

4. Initial operations before the implementation of MetalLB

If the address is missing, it should be defined by a manifest, f.e.:

```
apiVersion: v1
kind: Service
metadata:
  name: eaw-documentation
  namespace: default
spec:
  type: LoadBalancer
```

Change the manifest so it looks like this:

```
apiVersion: v1
kind: Service
metadata:
  name: eaw-documentation
  namespace: default
spec:
  type: LoadBalancer
  loadBalancerIP: 10.220.15.166
```

For ArgoCD, the address should be assigned by manually editing the manifest directly on the cluster.

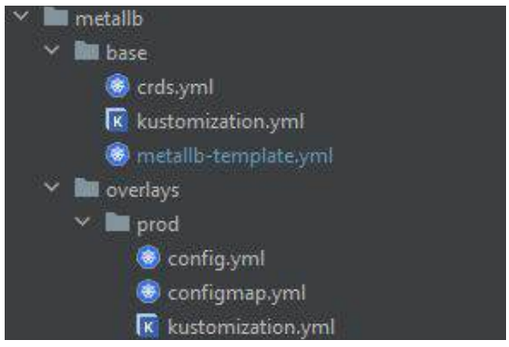
```
kubectl edit svc -n argocd argocd-server-external
```

5. Manifest, configmap and IPAddressPool declaration

Firstly, you should create a path in the ArgoCD application itself, in this case, a folder structure was created.

In this example, using the Kustomize tool for the cluster brings an additional value but is not necessarily required for a correct configuration. The use of the Kustomize tool can improve transparency and facilitate the steps for the configuration of the cluster.

This is what the proper file structure looks like:



In this present cluster, MetalLB has not been implemented yet with the help of ArgoCD – which requires a full reconstruction of the configuration and confronting it with the current state managed by ArgoCD.

Setting up configmap and IPAdressPool

To set up the cluster for updating, there are a couple of essential steps to follow for the configuration, like setting up configmap or IPAdressPool. It is important that they are necessary to update the cluster, they do not need to be performed when comparing ArgoCD to k8s.

In the new version, the configuration has changed – the new method of assigning addresses requires the setup of configmap and IPAdressPool. To adapt to those changes, it is essential to understand the new procedures for the management of IP addresses. Please note that those steps are essential for the proper configuration of the cluster, but are not required during the comparison operation of ArgoCD to k8s.

In the context of changes in the way addresses are assigned and managed after the update, it is worth noting that in the previous version it was based on configmap, which can be viewed with the following command:

```
kubectl get configmaps -n metallb-system config -o yaml
```

5. Manifest, configmap and IPAddressPool declaration

You should see as follows:

```
apiVersion: v1
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
        - 10.10.105.100-10.10.105.200
kind: ConfigMap
metadata:
  name: config
  namespace: metallb-system
```

As a result of those changes, it is essential to implement two other manifests, which will modify the way addresses are managed. In the present situation, we had to open an additional port, without which one of the webhooks would not work. This operation was only necessary in our situation, and results solely from its specifications and the functionalities of the service.

```
apiVersion: crd.projectcalico.org/v1
kind: NetworkPolicy
metadata:
  name: default.allow-metallb-controller-ingress
  namespace: metallb-system
spec:
  ingress:
  - action: Allow
    destination:
      ports:
      - 9443
    protocol: TCP
    source:
      namespaceSelector: global()
      selector: has(node-role.kubernetes.io/control-plane)
  selector: component == 'controller'
  types:
  - Ingress
```

Two additional manifests were created on the /metallb/overlays/prod/configmap.yml path, containing a copy of ConigMapy.

```
apiVersion: v1
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
        - 10.220.21.91-10.220.21.99
        - 10.220.21.50-10.220.21.59
kind: ConfigMap
metadata:
  name: config
  namespace: metallb-system
```

5. Manifest, configmap and IPAddressPool declaration

And a manifest containing new declarations and ad.dress conifg.yml:

```yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  - 10.220.21.91-10.220.21.99
  - 10.220.21.50-10.220.21.59
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: default-advertisement
  namespace: metallb-system
```

kind: L2 Advertisement wymaga wcześniejszego wdrożenia pliku /metallb/base/crds.yml. Bez wcześniejszej deklaracji CustomResourceDefinition zawierającej ten wpis nie będzie on działał.
kind: L2Advertisement requires an initial implementation of the /metallb/base/crds.yml file. Without a previous CustomResourceDefinition declaration containing this entry, it will not work.
Like explained previously with the present cluster, a manifest declaration was necessary to open an additional port 9443 without which one of the WebHooks would not work.

Example of a declaration in the config.yml file below:

```yaml
apiVersion: crd.projectcalico.org/v1
kind: NetworkPolicy
metadata:
  name: default.allow-metallb-controller-ingress
  namespace: metallb-system
spec:
  ingress:
  - action: Allow
    destination:
      ports:
      - 9443
    protocol: TCP
    source:
      namespaceSelector: global()
      selector: has(node-role.kubernetes.io/control-plane)
  selector: component == 'controller'
  types:
  - Ingress
```

Finally, declare the new files in /metallb/overlays/prod/kustomization.yml

```yaml
resources:
  - ../../base
  - configmap.yml
  - config.yml
```

6. Comparing ArgoCD to the environment

The first step is essential for setting up the manifest for the current version of MetalLB. It is essential since in the current example, the manifest for this application was missing in the Repository and ArgoCD.

Without it, we will not be able to safely and efficiently return the operation or do what we call “rollback”. In ArgoCD, rollback consists of returning to the previous, stable state from the last commit of the indicated repository.

The most efficient way for the comparison of environments is the generation of templates for a specific application with the help of Helma. Specify the current version of MetalLB and generate a template using the following command:

```
helm template --versoion={{target_version}} --include-crds=true --namespace=metallb-system metallb metallb/metallb
> metallb-template.yml
```

This process will allow you to accurately determine the differences between the current and the earlier version of the MetalLB application, which is crucial for the possible performance of an effective and safe rollback operation.

The current version can be specified by describing under the controller or speaker:

```
kubectl describe pod controller-7cf77c64fb-pr8t7 -n metallb-system | grep Image:
    Image:         quay.io/metallb/controller:v0.11.0
```

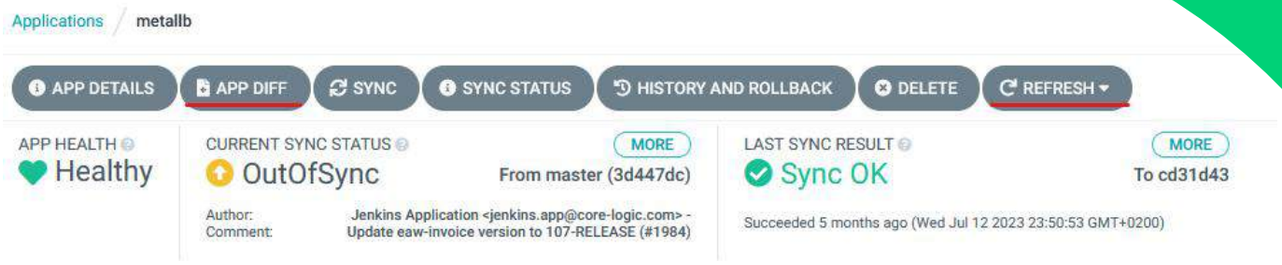
A tip here is to split the manifest into two separate files. One of them should contain all the entries from CustomResourceDefinition, and the other should contain the other objects. The last step here is the addition of appropriate entries to the /metallb/base/kustomization.yml file.

```
resources:
  - crds.yml
  - metallb-template.yml
```

Attention!

It is extremely important not to synchronize the application on the ArgoCD side in subsequent stages. Our current objective is solely to verify the changes and compare the manifest in ArgoCD to the current state of the cluster. When it comes to ArgoCD, we limited our use to only the “Refresh” and “App Diff” buttons.

6. Comparing ArgoCD to the environment



The “App Diff” functionality is key for comparing the changes between ArgoCD and the cluster. It is best to use the “Compact diff” to simplify the change analysis process. Our mission is to eliminate all the differences by bringing modifications directly in the manifests, and then the execution of commit and push for/in the repository. It is also worth your time to regularly compare the differences with the help of “App Diff”.

In the case of MetalLB you can also make use of the values,yaml file, that can be used for template generation, adding the -f values.yaml flag to the command. It is worth noting that this functionality can be limited in the case of MetalLB, and the values.yaml file becomes more useful for the update of prometheus-stack. After a successful comparison of ArgoCD with the cluster and once “App Diff” doesn’t declare any major differences, you can start the synchronization process by clicking on “Sync”. Thanks to this process, we gained a restore point, and the implementation of MetalLB by ArgoCD is a complete success

Setting up the manifest

This process is mostly similar when comparing ArgoCD to the environment. A key change is the generation of the template file with the most recent version of MetalLB with the help of this command:

```
helm template --include-crds=true --namespace=metallb-system metallb metallb/metallb  > metallb-template.yml
```

It is important to remember how essential it is to execute all the operations that we described in the previous step. Because we are working on the update of the version, major changes may appear when you click on the App Diff tool.

It is essential that elements like ‘name’ or ‘namespace’ keep their original value. This means that those key values stay the same even in the case of major changes. Remember that differences in App Diff can be quite significant, so there is a need for conscious monitoring of these changes and possible manual correction to ensure consistency and correctness of the environment configuration.

7. The update of MetalLB

After the generation of the manifest for the new version, we finally proceed to the upgrade of the application in ArgoCD.

Remember that before proceeding to the updating process it is final to set up the configuration for configmap and IPAddressPool. It's best to start with the synchronization of CRD objects because those are essential for the configuration and they're worth implementing before the update.

CRDs are often backward compatible, which means that their update should not have any negative effect on the functioning of the cluster or its services. However, in the case of CRD, it is essential to make the synchronization with the "Replace" option. Forgetting this step could lead to synchronization mistakes in ArgoCD.

After the synchronization of CRD objects, you can proceed to the synchronization of the other elements. After the finalization of this process in ArgoCD, check the status of pods in namespace metallb-system with the following command:

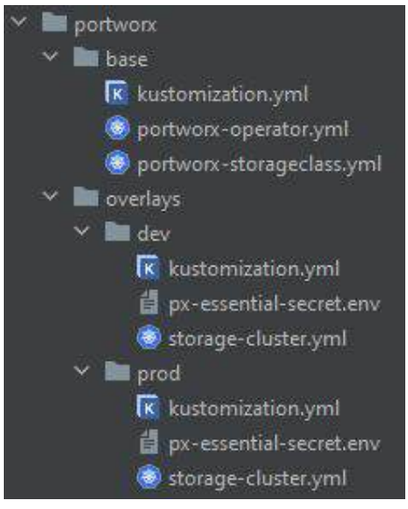
```
watch kubectl get pod -o wide -n metallb-system
```

Each pod should go through the restart process and get the status "Ready 1/1" and "Status Running". Monitoring this process helps to ensure that the update will be made appropriately and that all the subsystems will be stabilized accordingly.

8. Portworx

Before beginning the implementation of Portworx in the Kubernetes cluster with the use of ArgoCD, there are a couple of initial steps that are worth following through.

In order to efficiently manage Portworx with the help of ArgoCD, it is best to create a separate application dedicated to that service. The file structure should comply with the path declared in the ArgoCD application configuration.



When organizing the structures of Portworx files, it is important to check that the path is consistent with the one declared in the ArgoCD application configuration. In the example above, a portworx folder was created, with the kustomization.yaml folder and the structure for eventual overlayers. The portworx.yaml file will contain a specific configuration for Portworx.

Przyrównanie ArgoCD do klastra

If Portworx has not been under the management of ArgoCD yet, it is essential to create the appropriate environment. As with MetalLB, this step will allow us to create a restore point. In the situation where an earlier configuration exists, it is advised to make sure to restore the environment.

In the next step, start the synchronization process of the application in ArgoCD, to implement the changes defined in the configuration files.

9. Setting up the StorageCluster manifest

If the implementation process and the potential initial update have been properly executed, the Storage Cluster manifest should be now available for downloading on the Portworx panel. Below are the necessary steps for the realization of the manifest, as well as for the eventual update of the service.

In the portworx panel, in the section “Install and Run”, you should click on the “Download” option that appears when clicking on “Actions”.

After downloading the manifest, it is important to make sure that the entry for portworx.io/install-source is the same as for key user= and c=, and then if the entry name: is the same as for portworx.io/install-source c=

```
portworx.io/install-source: "https://install.portworx.com/?operator=true&mc=false&kbver=1.25.12&oem=esse&user=190e038c-f42f-
-f42f-11eb-a2c5-ab-a2c5-c24e499c7467&b=true&j=auto&c=px-cluster-2c55c9b4-b839-470b-9ad9-7a46d3b07cf2&stork=false&csi=
true&mon=true&tel=true&st=k8s&promop=true"
```

```
name: px-cluster-2c55c9b4-b839-470b-9ad9-7a46d3b07cf2
```

If differences appear, it is advised to download data directly from the cluster and to insert it directly into the manifest. The ready manifest should be stored in the following file: /portwrox/overlays/dev/storage-cluster.yml.

10. Updating the service manifest

When updating the Portworx service, it is enough to make changes in the appropriate sections of the manifest.

To change the Kubernetes version (kbver=):

- In the manifest folder, find the portworx.io/install-source entry.
- Change the value of kbver= to the appropriate Kubernetes version, f.e. 1.25.11.

Verification of entry (name:):

- Check if the value under name: is the same as in the previous version.
- If differences appear, just change the value of the name: so it's consistent with the previous version.

Verification of the (user=) section:

- Check if the value under user= in the section xxx is the same as in the previous version.
- Check that the key user= is consistent with the previous version.

After implementing those changes, the Portworx service manifest is ready to be applied in the Kubernetes cluster. Do not forget to verify the correctness of the configuration and that the applied changes are consistent with the updating processes.

```
apiVersion: core.libopenstorage.org/v1
kind: StorageCluster
metadata:
  annotations:
# Różnica jedynie w wersji k8s. W razie konieczności podmienić c= na wartość z name:
    portworx.io/install-source: "https://install.portworx.com/?operator=true&mc=false&kbver=1.25.11&oem=esse&u
    ser=190e038c-f42f-11eb-a2c24e499c7467&b=true&j=auto&c=px-cluster-2c55c9b4-b839-470b-9ad9-7a46d3b07cf2&stor
    k=false&csi=true&mon=true&tel=true&st=k8s&promop=true"

    portworx.io/misc-args: "--oem esse"
  finalizers:
    - operator.libopenstorage.org/delete
# Wpis musi być jednakowy z poprzednią wersją. Przy generacji menifestu zmienia się każdorazowo.
  name: px-cluster-2c55c9b4-b839-470b-9ad9-7a46d3b07cf2
  namespace: kube-system
spec:
  image: portworx/oci-monitor:2.13.6
  imagePullPolicy: Always
```


kzadlo@core-logic.com
www.core-logic.com

11. Preparing the portworx-operator manifest

In the portworx panel, go to the “Install and Run” section and keep on clicking on “Next” until you reach the final step. In the “Portworx Operator” section, click on the eye icon next to the operator manifest, copy the Portworx Operator manifest, go to the /portworx/base/catalog, and create the portworx-operator.vml file. Then you just need to insert the copy of the Portwork Operator manifest into the new file.

After executing those steps, the Portworx Operator manifest will be available in the local repository, ready to be used in the configuration of the Kubernetes cluster. Check the correctness of the copied manifest and apply it following the requirements of the application.

kzadlo@core-logic.com
www.core-logic.com

12. Preparing the StorageClass manifest

To access the StorageClass manifest for Portworx, run the following command in the terminal to receive the YAML of the StorageClass manifest.

```
kubectl get storageclass px-storageclass -oyaml
```

After this operation, the manifest will be displayed in the console. You can also save the manifest directly in the file with this command:

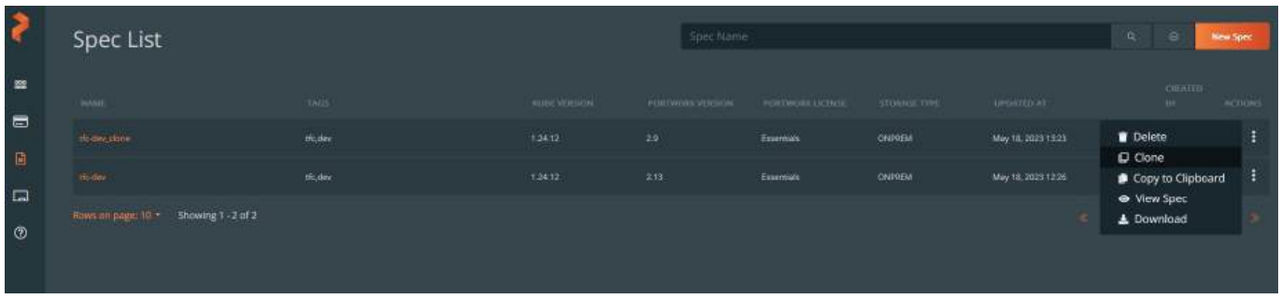
```
kubectl get storageclass px-storageclass -oyaml > portworx-storageclass.yml
```

This step will save the StorageClass manifest to the portworx-storageclass.yml file.

After executing those steps, the StorageClass manifest for Portworx will become available, which you can then edit or use in other configuration processes. It is essential to monitor and verify the values in the manifest before its application.

13. Preparation of manifests

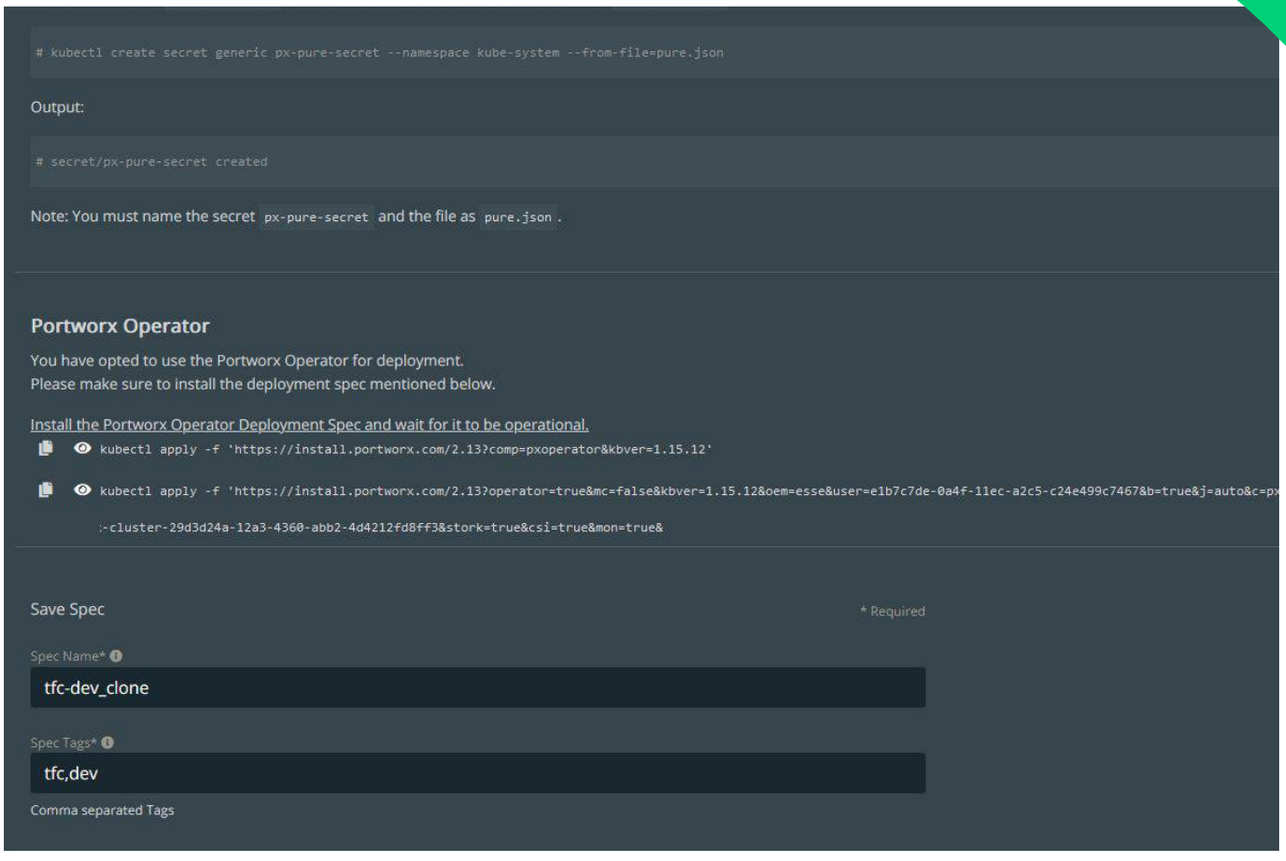
To update the manifest through the Portworx panel, login to the platform and choose the current specification in the “Install and Run”. Then clone it, defining new versions for Portworx and the cluster.



It is essential to define a new version, for both Portworx and the cluster itself. In the next steps, it is required to adapt the specifications to our needs.

The final result is a familiar view from which you can download all the necessary manifests.

13. Preparation of manifests



After the cloning process, download the manifest and insert it in the dedicated path for your environment f.e. /portworx/overlays/dev/. On the last resuming screen will be displayed information concerning the creation of a Secret type object.

In the situation where you have the right Secret object, there is no need to create another one. Thanks to this step, you can update the Portworx manifest, adapting it to the current trends.

14. Modification of manifests

In this step, it is also primordial to verify and potentially conduct some changes.
These data are key, and if necessary, remember to replace them. Additionally, it is essential to create and add the appropriate Secret object that we previously mentioned.

To realize that, you can copy the Secret object directly from the cluster with the following command:

```
kubectl get secret -n kube-system px-essential -oyaml
```

Then, add the copied data to the file in the dedicated path, f.e. /portworx/overlays/px-essential-secret.env. It is also worth remembering to not place the Secret in such a way in the version control system, like GIT. This is valid for all Secret objects, not just this particular one. If its addition to the repository is required, is it advised to encrypt it with a GPS key.

15. Portworx update

Before starting the update, it is required to set up the kustomization.yml files, which will connect all the previously created manifests. Each folder (/base/, /overlays/dev/, /overlays/prod) needs to have a /portworx/base/kustomization.yml file.

The folder /base/ nomen omen is used as a base. The manifests declared in the kustomization.yml file from this folder are the base for manifests from the /overlays/ folder.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
# Zdefiniowany dodany przez nas wcześniej portworx-operator oraz storageclass
  - portworx-operator.yml
  - portworx-storageclass.yml
```

/portworx/overlays/kustomization.yml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

generatorOptions:
  disableNameSuffixHash: true

resources:
# Zdefiniowany folder /base/, o którym pisałem wcześniej
  - ../../base
# Zdefiniowany storage-clister który wygenerowaliśmy w poprzednich krokach
  - storage-cluster.yml

# W tej sekcji dodawany jest secret. W tym konretnym przypadku konieczne jest by secret był już zaszyfrowany przez klucz GPG
secretGenerator:
  - name: px-essential
    namespace: kube-system
    envs:
      - px-essential-secret.env
```

After executing those operations, you can compare the manifest with the cluster. This comparison can be made through ArgoCD after sending changes to the GIT repository. It is best to avoid the use of the Sync option and to prefer the Refresh one, as well as App Diff.
If ArgoCD shows no errors or changes that impact the functionality of the cluster, we can start the synchronization of the ArgoCD application. You can observe the changes on the cluster side with the following command:

```
watch kubectl get pods -o wide -n kube-system
```

After stabilizing the cluster, the update will be complete.

16. Prometheus Stack

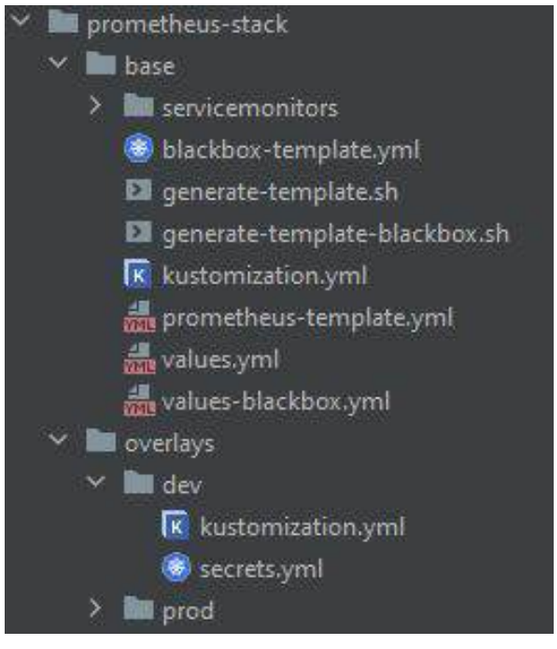
Updating the prometheus-stack into the cluster was one of the most time-consuming and demanding processes. It is mainly caused by the complexity of the Prometheus-stack itself which is made of several components and because of the importance of recreating ArgoCD from scratch.

Despite the many hours required to implement the Prometheus-stack with ARgoCD, it is a justified and necessary decision, especially in the context of long-term development of the cluster and future updates.

Preliminary operations

Before starting the update it is important to remember a couple of key steps – it is important to remember to verify that the created application in ArgoCD is called “monitoring” and that the definition of the application is located in the monitoring.yml file in the repository.

It is also important to verify that the structure of the files corresponds to the cluster’s requirements and that all the necessary files and configurations are available before starting the update. By taking care of those details at the preparation stage, complications are less likely to happen and it ensures a smooth updating process of the prometheus-stack into the cluster.



kzadlo@core-logic.com
www.core-logic.com

17. Preparation of the script and values.yml file

As a first step, it is necessary to prepare the script for the generation of the prometheus-stack with the help of the Helm tool.

Before the first generation of the manifest it is important to define the version that is currently used in the cluster. This way, you have a “clear” manifest for the current version. Because of the complexity and size of the prometheus-stack, the use of values.yml files can be very beneficial (and sometimes is required). You will find below Helm’s instruction, to generate a manifest, taking into consideration the values from values.yml files and the ({{target-version}}):

```
helm template -f values.yml --include-crds=true --namespace=monitoring --version={{target-version}} monitoring
prometheus-community/kube-prometheus-stack  > prometheus-template.yml
```

Before the generation, it is worth getting acquainted with the documentation of the values.yml file. This file allows the adaptation of the manifest to the specifications of the cluster and other related services, by predefining parts of variables.

Thanks to this process of generating manifests has become more and more flexible and adapted to individual needs as well to the specifications of a given cluster.

kzadlo@core-logic.com
www.core-logic.com

18. Preparation of the restore point

In regards to the consequent size of the prometheus-stack manifest, an efficient approach is the local change comparison. The realization of this task by ArgoCD is therefore quite complicated because of the important quantity of data that has to be treated and showcased in the browser.

In the cluster's situation, it is best to use different methods and use local change comparison. It is however best to create a restore point, to which we will refer when comparing the new manifests. This is valid for manual setup and for downloading the prometheus-stack manifest from the cluster.

This operation should be prepared from a strict manifesto for the current version of the cluster. It is worth using the previously prepared script or the command to generate templates. However, you should execute this command with the option --include-crds=false, which will drastically reduce the size of the file.

You can now proceed to the next steps of setting up the restore point and local change comparison in the manifests for the prometheus-stack.

Setting up the strict manifest for the current version available on the cluster

Setting up the strict manifest for the current version available on the cluster can be made with the help of the previously prepared script or the command to generate templates. However, you should execute this command with the option --include-crds=false, which will drastically reduce the size of the file.

Setting up object backup according to the strict manifest

If the strict manifest is already available, it is worth creating a backup. To achieve this, create a file containing the same objects as the manifest. In other words, you need to create a copy of the raw manifest, by directly downloading the objects from the cluster.

18. Preparation of the restore point

The raw manifest can contain various objects, like f.e. Service, which will be displayed as follows:

```
# Source: kube-prometheus-stack/templates/exporters/kube-controller-manager/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: monitoring-kube-prometheus-kube-controller-manager
  labels:
    app: kube-prometheus-stack-kube-controller-manager
    jobLabel: kube-controller-manager

    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/instance: monitoring
    app.kubernetes.io/version: "28.0.0"
    app.kubernetes.io/part-of: kube-prometheus-stack
    chart: kube-prometheus-stack-28.0.0
    release: "monitoring"
    heritage: "Helm"
  namespace: kube-system
spec:
  clusterIP: None
  ports:
    - name: http-metrics
      port: 10252
      protocol: TCP
      targetPort: 10252
  selector:
    component: kube-controller-manager
  type: ClusterIP
```


kzadlo@core-logic.com
www.core-logic.com

19. Strict manifest for the current version available on the cluster

The creation of a backup is made by copying the objects from the cluster and saving them in a file that will serve as a restore point in case of need to return to the original configuration.

It is advised that the download of the manifest object Service named monitoring-kube-prometheus-kube-controller-manager is made with the following command:

```
kubectl get svc -n kube-system -o yaml monitoring-kube-prometheus-kube-controller-manager >> prometheus-templ
ate-old.yml && echo "---" >> prometheus-template-old.yml
```

Marked sections can be removed successively. They will only interfere with the process of comparing files.

19. Strict manifest for the current version available on the cluster

```yaml
apiVersion: v1
kind: Service
metadata:
# Sekcja annotatnios może zostać usunięta, będzie później przeszkadzać przy porównaniach
#  annotations:
#    kubectl.kubernetes.io/last-applied-configuration: |
#      {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":{"app":"kube-prometheus-stack-kube-controller-manager","app.kubernet
        stack-kube-controller-manager","app.kubernetes.io/instance":"monitoring","app.kubernetes.io/managed-by":"Helm","app.kubernetes.io/part-of":"kube-
#    meta.helm.sh/release-name: monitoring
#    meta.helm.sh/release-namespace: monitoring
# creationTimestamp również może zostać usunięte
#  creationTimestamp: "2021-09-22T13:56:12Z"
  labels:
    app: kube-prometheus-stack-kube-controller-manager
    app.kubernetes.io/instance: monitoring
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/part-of: kube-prometheus-stack
    app.kubernetes.io/version: 28.0.0
    chart: kube-prometheus-stack-28.0.0
    heritage: Helm
    jobLabel: kube-controller-manager
    release: monitoring
  name: monitoring-kube-prometheus-kube-controller-manager
  namespace: kube-system
# resourceVersion oraz uid może zostać usunięte
#  resourceVersion: "522965913"
#  uid: e787cbf2-4c49-4282-9d12-d8276fdb636a
spec:
  clusterIP: None
  clusterIPs:
  - None
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: http-metrics
    port: 10252
    protocol: TCP
    targetPort: 10252
  selector:
    component: kube-controller-manager
  sessionAffinity: None
  type: ClusterIP
# Sekcja status może zostać usunięta
#status:
#  loadBalancer: {}
```

You should repeat this operation for each object from the raw manifest, according to their order. Each object from the manifest should be copied from the cluster and saved in the appropriate file, creating this way a restore point for specific configurations.

After copying all the objects, you need to proceed to a comprehensive file comparison. A practical operation that will facilitate your work is comparing the objects right after saving them in the prometheus-stack-old.yml file. This way, it is possible to make changes and corrections in the values.yaml file on an ongoing basis.

20. Comparing ArgoCD to the cluster

Similar to the situation with the previous application, this step is only necessary if prometheus-stack isn't available from ArgoCD's side. In such a situation, comparing the application to the current state of the cluster is essential.

Creating new definitions – patch.yml and secrets.yml

Unfortunately, not all variables can be defined in the values.yaml file described earlier. Some settings do not find their equivalent there, which may require an alternative approach to define them. In this situation, we can use the kustomize tool, which allows for the flexible addition of extra configurations, or even full manifests.

The kustomization.yml file is defined in the following way:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - ../../base

patches:
  - path: secrets.yml

patchesStrategicMerge:
  - patch.yml
```

20. Comparing ArgoCD to the cluster

The patchesStrategicMerge mechanism allows you to flexibly overwrite only those variables or configurations that are of interest, leaving the rest intact. This keeps previous entries updated rather than completely replacing them.

Entry in the raw manifesto:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name:  monitoring-kube-prometheus-admission
  labels:
    app: kube-prometheus-stack-admission
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/instance: monitoring
    app.kubernetes.io/version: "28.0.0"
    app.kubernetes.io/part-of: kube-prometheus-stack
    chart: kube-prometheus-stack-28.0.0
    release: "monitoring"
    heritage: "Helm"
webhooks:
  - name: prometheusrulemutate.monitoring.coreos.com
    failurePolicy: Ignore
    rules:
      - apiGroups:
          - monitoring.coreos.com
        apiVersions:
          - "*"
        resources:
          - prometheusrules
        operations:
          - CREATE
          - UPDATE
    clientConfig:
      service:
        namespace: monitoring
        name: monitoring-kube-prometheus-operator
        path: /admission-prometheusrules/validate
    admissionReviewVersions: ["v1", "v1beta1"]
    sideEffects: None
```

20. Comparing ArgoCD to the cluster

Content of the patch.yml file:

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: monitoring-kube-prometheus-admission
webhooks:
  - admissionReviewVersions:
      - v1
      - v1beta1
    failurePolicy: Fail
    matchPolicy: Equivalent
    name: prometheusrulemutate.monitoring.coreos.com
    rules:
      - apiGroups:
          - monitoring.coreos.com
        apiVersions:
          - '*'
        operations:
          - CREATE
          - UPDATE
        resources:
          - prometheusrules
        scope: '*'
    timeoutSeconds: 10
```

Result of kustomize build:

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  labels:
    app: kube-prometheus-stack-admission
    app.kubernetes.io/instance: monitoring
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/part-of: kube-prometheus-stack
    app.kubernetes.io/version: 28.0.0
    chart: kube-prometheus-stack-28.0.0
    heritage: Helm
    release: monitoring
  name: monitoring-kube-prometheus-admission
webhooks:
- admissionReviewVersions:
  - v1
  - v1beta1
  clientConfig:
    service:
      name: monitoring-kube-prometheus-operator
      namespace: monitoring
      path: /admission-prometheusrules/validate
# Zmiana z Ignore na Failure
  failurePolicy: Fail
  matchPolicy: Equivalent
  name: prometheusrulemutate.monitoring.coreos.com
  rules:
  - apiGroups:
    - monitoring.coreos.com
    apiVersions:
    - '*'
    operations:
    - CREATE
    - UPDATE
    resources:
    - prometheusrules
# Dodano scope
    scope: '*'
  sideEffects: None
# Dodano timeoutSeconds
  timeoutSeconds: 10
```

Additionally, it is essential to add a Secret object. This task should be performed the same way we did with “Portworx Secret”.

21. Updating Prometheus Stack

The updating process is pretty similar to the ones you went through for previous services – generate the manifest for the most recent version, taking into account the values.yml, patch.yml, and secrets.yml files.

The command for the generation of the manifest is presented as follows:

```
helm template -f values.yml --include-crds=true --namespace=monitoring --version=28.0.0
monitoring prometheus-community/kube-prometheus-stack  > prometheus-template.yml
```

When generating the manifest for the new version, it is important to remember to change the switch – include-crds to true – as well as to separate the manifests, the same way we proceeded until now. Then, the changes were compared using the App Diff tool in ArgoCD.

This allows to catch even the smallest changes that could have been missed earlier. After finalizing this process and making sure there are no key differences or errors, it is time for the synchronization of the application with ArgoCD.

If App Diff does not detect any major differences, you can safely continue and proceed to the synchronization of the application, guaranteeing consistency and compatibility of configuration with cluster expectations and requirements.

22. Blackbox Exporter

For the present cluster, upgrading the Blackbox Exporter version was essential to maintain the stability and efficiency of monitoring services. The Blackbox Exporter update process is not significantly different from the one for the application, but here there was no need to create a separate application before implementing changes.

When it comes to the monitoring architecture, Blackbox Exporter plays a key role, enabling the collection and analysis of data on the status and availability of services. For this reason, it is worth using the existing monitoring application, which has been designed and adapted to for the update of the entire range of monitoring tools, including prometheus-stack.

With the monitoring application that has already been configured and optimized for requirements, the Blackbox Exporter update process could be carried out efficiently and the risk of possible complications or incompatibilities in the monitoring environment was minimized. This way, we ensured smooth monitoring operations and high-quality services by the cluster.

Setting up the manifest

To set up the manifest for the update of Blackbox Exporter, it is advised to rely on the values.yaml file as well as the Helm tool.

To generate the manifest, use the following command:

```
helm template -f values-blackbox.yml --include-crds=true --namespace=monitoring --version={{target-version}} blackbox-exporter prome
theus-blackbox-exporter > blackbox-template.yml
```

In the previous command, {{target-version}} corresponds to the current version of Blackbox on the cluster, based on the values-blackbox.yml file.

Thanks to this operation, we obtained a manifest ready to be implemented and that takes into account the specifications of the monitoring environment and ensures compliances with the clusters' requirements.

23. Preparing ArgoCD for the cluster

Like with the previous versions, it is essential to precisely adapt the file to the requirements and configuration of the cluster. In order to identify the essential changes, it is worth using the AppDiff tool available in ArgoCD, that will allow you to compare te repository with the current state of the cluster.

The goal is to compare the ArgoCD manifest with the real state of the cluster, to verify that the configuration are compatible. This operation is not very different from the previous ones.

After identifying the differences and implementing the essential changes, the new manifest should be added to the /prometheus-stack/base/kustomization.yml file.

The final structure of the file in the main folder should look as follows:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - prometheus-template.yml
  - blackbox-template.yml
```

After having compared the repository with the current state of the environment, we can proceed to the synchronization of the application with the help of ArgoCD. This way, we have a restore point and we ensure consistency and coherence of the application's configuration with the requirements and expectations related to the cluster.

24. Update of the Control-Plane nodes

After updating all the different components, it is time to update Kubernetes.
The first step is the update of the control-plane nodes.

Updating the kubeadm tool

To perform the update, you must first determine the appropriate version that you want to install.
Then, after selecting the appropriate version, you can install it by:

```
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.25.14-00 && \
apt-mark hold kubeadm
```

After installing the appropriate version of the kubeadm tool, it is time to prepare the upgrade plan by executing this command:

```
kubeadm upgrade plan
```

25. Updating the Kubernetes cluster

Then, complete the update of the Kubernetes cluster with the following command:

```
sudo kubeadm upgrade node
```

After the upgrade of the kubeadm tool, you also need to update kubelet and kubectl. Before the update, it is best to deactivate their wall, and to reactivate it after the update. To update kubelet and kubectl, follow these steps:

After the node was designated for updating, it was eliminated from the tasks and the correctness of the information was ensured.

```
kubectl drain <node-to-drain> --ignore-daemonsets
```

Then we updated kubelet and kubectl.

```
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.25.14-00 kubectl=1.25.14-00 && \
apt-mark hold kubelet kubectl
```

After installing the new versions, restart the kubelet service:

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

Ultimately, it is worth verifying that the node was put back to work with the help of the following command. Additionally, check that <node-to-drain> is the name for the nodes that have been updated earlier.

```
kubectl uncordon <node-to-drain>
```

Nody worker

Updating the worker nodes is made possible with the Ansible playbook, which is separated into 2 files:

- update-node-k8s.yml – responsible for the update of Kubernetesa.
- update-node-os.yml – responsible for the update of the operation system and for the restart of the server

Please note, that those operations should be executed one by one, node by node, to ensure the continuity of the service availability.

Summary

This is the end of the guide to upgrading the Kubernetes cluster from version 1.25 to version 1.26. In the latest version of the Kubernetes cluster, a transition has been made from the mechanism of the Pod Security Policy (PSP) to a more flexible and modern method, called Pod Security Admission (PSA). This is a significant change that requires careful preparation and strict adherence to procedures when implementing and updating tools in the cluster. The article discussed key steps before upgrading a cluster, focusing on services that previously used PSP

The report provides a comprehensive overview of the key aspects involved in upgrading a Kubernetes cluster, emphasizing the need to incorporate changes to the security mechanism and follow upgrade procedures for each cluster component.

Konrad Matyas

Vice CEO of Core Logic

core • logic

Konrad Matyas
Vice CEO of Core Logic

Contact us

We will share our knowledge and experience,
to help you solve your problems
and create the product you've always wanted.

Web:
www.core-logic.com

Feliksa Radwanskiego 15/1,
30-065 Krakow, Poland

Phone & Whatsapp:
+48 606 524 052

E-mail:
kzadlo@core-logic.com