Red Hat Ansible Automation Platform 2.4

Red Hat Ansible Automation Platform performance considerations for operator based installations

Configure automation controller for improved performance on operator based installations

Last Updated: 2024-06-14

Red Hat Ansible Automation Platform 2.4 Red Hat Ansible Automation Platform performance considerations for operator based installations

Configure automation controller for improved performance on operator based installations

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide provides recommendations on how to configure automation controller and Container Groups resource requests and other kubernetes configuration options to more efficiently run jobs at scale on operator based installations of automation controller.

Table of Contents

PREFACE 3

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION 4

CHAPTER 1. POD SPECIFICATION MODIFICATIONS 5
- 1.1. INTRODUCTION 5
  - 1.1.1. Customizing the pod specification 7
  - 1.1.2. Enabling pods to reference images from other secured registries 8
- 1.2. RESOURCE MANAGEMENT FOR PODS AND CONTAINERS 9
  - 1.2.1. Requests and limits 9
  - 1.2.2. Resource types 10
  - 1.2.3. Specifying resource requests and limits for pods and containers 10
  - 1.2.4. Resource units in Kubernetes 10
  - 1.2.5. Size recommendations for resource requests 11

CHAPTER 2. CONTROL PLANE ADJUSTMENTS 12
- 2.1. REQUESTS AND LIMITS FOR TASK CONTAINERS 12
- 2.2. CONTAINERS RESOURCE REQUIREMENTS 12
- 2.3. ALTERNATIVE CAPACITY LIMITING WITH AUTOMATION CONTROLLER SETTINGS 13

CHAPTER 3. SPECIFYING DEDICATED NODES 15
- 3.1. ASSIGNING PODS TO SPECIFIC NODES 15
- 3.2. SPECIFY NODES FOR JOB EXECUTION 16
- 3.3. CUSTOM POD TIMEOUTS 18
- 3.4. JOBS SCHEDULED ON THE WORKER NODES 19

CHAPTER 4. CONFIGURING ANSIBLE AUTOMATION CONTROLLER ON OPENSHIFT CONTAINER PLATFORM 20
- 4.1. MINIMIZING DOWNTIME DURING OPENSHIFT CONTAINER PLATFORM UPGRADE 20

PREFACE

Deploying applications to a container orchestration platform such as Red Hat OpenShift Container Platform provides a number of advantages from an operational perspective. For example, an update to the base image of an application can be made through a simple in-place upgrade with little to no disruption. Upgrading the required operating system of an application deployed to traditional virtual machines can be a much more disruptive and risky process.

Although application and operator developers can provide many options to OpenShift Container Platform users regarding the deployment of the application, these configurations must be provided by the end user because they are dependent on the configuration of OpenShift Container Platform.

For example, use of node labels in the Openshift cluster can help ensure different workloads are run on specific nodes. This type of configuration must be provided by the user as the Ansible Automation Platform Operator has no way of inferring this.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

If you have a suggestion to improve this documentation, or find an error, please contact technical support at https://access.redhat.com to create an issue on the Ansible Automation Platform Jira project using the **docs-product** component.

CHAPTER 1. POD SPECIFICATION MODIFICATIONS

1.1. INTRODUCTION

The Kubernetes concept of a pod is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, or managed.

Pods are the equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a life cycle. They are defined, then they are assigned to run on a node, then they run until their containers exit or they are removed for some other reason. Pods, depending on policy and exit code, can be removed after exiting, or can be retained to enable access to the logs of their containers.

Red Hat Ansible Automation Platform provides a simple default pod specification, however, you can provide a custom YAML, or JSON document that overrides the default pod specification. This custom document uses custom fields, such as **ImagePullSecrets**, that can be serialized as valid Pod JSON or YAML.

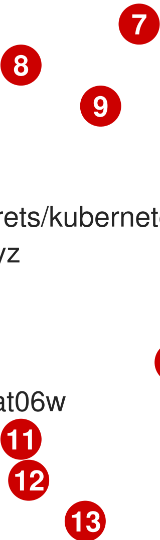
A full list of options can be found in the Openshift Online documentation.

Example of a pod that provides a long-running service.

This example demonstrates many features of pods, most of which are discussed in other topics and thus only briefly mentioned here:

```
apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }              (1)
  labels:
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-      (2)
spec:
  containers:                      (3)
  - env:               (4)
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2 (5)
    imagePullPolicy: IfNotPresent
    name: registry
    ports:                     (6)
    - containerPort: 5000
```

```
      protocol: TCP
    resources: {}
    securityContext: { ... }
    volumeMounts:
    - mountPath: /registry
      name: registry-storage
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-br6yz
      readOnly: true
  dnsPolicy: ClusterFirst
  imagePullSecrets:
  - name: default-dockercfg-at06w
  restartPolicy: Always
  serviceAccount: default
  volumes:
  - emptyDir: {}
    name: registry-storage
  - name: default-token-br6yz
    secret:
      secretName: default-token-br6yz
```



Label	Description
annotations:	Pods can be "tagged" with one or more labels, which can then be used to select and manage groups of pods in a single operation. The labels are stored in key:value format in the metadata hash. One label in this example is **docker-registry=default**.
generateName:	Pods must have a unique name within their namespace. A pod definition can specify the basis of a name with the **generateName** attribute, and add random characters automatically to generate a unique name.
containers:	**containers** specifies an array of container definitions. In this case (as with most), defines only one container.
env:	Environment variables pass necessary values to each container.
image:	Each container in the pod is instantiated from its own Docker-formatted container image.
ports:	The container can bind to ports made available on the pod’s IP.
resources:	When you specify a pod, you can optionally describe how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM). Other resources are available.
securityContext:	OpenShift Online defines a security context for containers that specifies whether they are permitted to run as privileged containers, run as a user of their choice, and more. The default context is very restrictive but administrators can change this as required.

Label	Description
volumeMounts:	The container specifies where external storage volumes should be mounted within the container. In this case, there is a volume for storing the registry's data, and one for access to credentials the registry needs for making requests against the OpenShift Online API.
ImagePullSecrets	A pod can contain one or more containers, which must be pulled from some registry. If containers come from registries that require authentication, you can give a list of **ImagePullSecrets:** that refer to **ImagePullSecrets** present in the namespace. Having these specified enables Red Hat OpenShift Container Platform to authenticate with the container registry when pulling the image. For further information, see Resource Management for Pods and Containers in the Kubernetes documentation.
restartPolicy:	The pod restart policy with possible values **Always**, **OnFailure**, and **Never**. The default value is **Always**.
serviceAccount:	Pods making requests against the OpenShift Online API is a common enough pattern that there is a **serviceAccount** field for specifying which service account user the pod should authenticate as when making the requests. This enables fine-grained access control for custom infrastructure components.
volumes:	The pod defines storage volumes that are available to its container(s) to use. In this case, it provides an ephemeral volume for the registry storage and a secret volume containing the service account credentials.

You can change the pod used to run jobs in a Kubernetes-based cluster by using automation controller and editing the pod specification in the automation controller UI. The pod specification that is used to create the pod that runs the job is in YAML format. For further information about editing the pod specifications, see Customizing the pod specification.

1.1.1. Customizing the pod specification

You can use the following procedure to customize the pod.

Procedure

1. In the automation controller UI, go to **Administration → Instance Groups**.
2. Check **Customize pod specification**.
3. In the **Pod Spec Override** field, specify the namespace by using the toggle to enable and expand the **Pod Spec Override** field.
4. Click **Save**.
5. Optional: Click **Expand** to view the entire customization window if you want to provide additional customizations.

The image used at job launch time is determined by the execution environment associated with the job. If a Container Registry credential is associated with the execution environment, then automation controller uses **ImagePullSecret** to pull the image. If you prefer not to give the service account permission to manage secrets, you must pre-create the **ImagePullSecret**, specify it on the pod specification, and omit any credential from the execution environment used.

1.1.2. Enabling pods to reference images from other secured registries

If a container group uses a container from a secured registry that requires a credential, you can associate a Container Registry credential with the Execution Environment that is assigned to the job template. Automation controller uses this to create an **ImagePullSecret** for you in the OpenShift Container Platform namespace where the container group job runs, and cleans it up after the job is done.

Alternatively, if the **ImagePullSecret** already exists in the container group namespace, you can specify the **ImagePullSecret** in the custom pod specification for the **ContainerGroup**.

Note that the image used by a job running in a container group is always overridden by the Execution Environment associated with the job.

Use of pre-created ImagePullSecrets (Advanced)

If you want to use this workflow and pre-create the **ImagePullSecret**, you can source the necessary information to create it from your local **.dockercfg** file on a system that has previously accessed a secure container registry.

Procedure

The **.dockercfg file**, or **$HOME/.docker/config.json** for newer Docker clients, is a Docker credentials file that stores your information if you have previously logged into a secured or insecure registry.

1. If you already have a **.dockercfg** file for the secured registry, you can create a secret from that file by running the following command:

```
$ oc create secret generic <pull_secret_name> \
--from-file=.dockercfg=<path/to/.dockercfg> \
--type=kubernetes.io/dockercfg
```

2. Or if you have a **$HOME/.docker/config.json** file:

```
$ oc create secret generic <pull_secret_name> \
--from-file=.dockerconfigjson=<path/to/.docker/config.json> \
--type=kubernetes.io/dockerconfigjson
```

3. If you do not already have a Docker credentials file for the secured registry, you can create a secret by running the following command:

```
$ oc create secret docker-registry <pull_secret_name> \
--docker-server=<registry_server> \
--docker-username=<user_name> \
--docker-password=<password> \
--docker-email=<email>
```

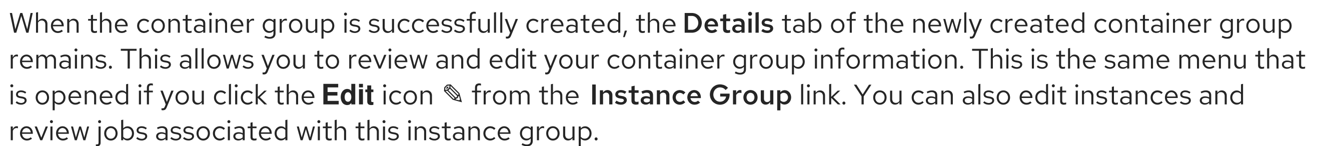
4. To use a secret for pulling images for pods, you must add the secret to your service account. The name of the service account in this example must match the name of the service account the pod uses. The default is the default service account.

```
$ oc secrets link default <pull_secret_name> --for=pull
```

5. Optional: To use a secret for pushing and pulling build images, the secret must be mountable inside a pod. You can do this by running:

```
$ oc secrets link builder <pull_secret_name>
```

6. Optional: For builds, you must also reference the secret as the pull secret from within your build configuration.



When the container group is successfully created, the **Details** tab of the newly created container group remains. This allows you to review and edit your container group information. This is the same menu that is opened if you click the **Edit** icon ✎ from the **Instance Group** link. You can also edit instances and review jobs associated with this instance group.

1.2. RESOURCE MANAGEMENT FOR PODS AND CONTAINERS

When you specify a pod, you can specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM).

When you specify the resource request for containers in a Pod, the kubernetes-scheduler uses this information to allocate the node to place the Pod on.

When you specify a resource limit for a container, the *kubelet*, or node agent, enforces those limits so that the running container is not permitted to use more of that resource than the limit you set. The kubelet also reserves at least the requested amount of that system resource specifically for that container to use.

1.2.1. Requests and limits

If the node where a pod is running has enough resources available, it is possible for a container to use more resources than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

For example, if you set a memory request of 256 MiB for a container, and that container is in a pod scheduled to a Node with 8GiB of memory and no other pods, then the container can try to use more RAM.

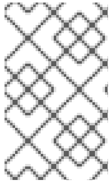
If you set a memory limit of 4GiB for that container, the kubelet and container runtime enforce the limit. The runtime prevents the container from using more than the configured resource limit.

If a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an *Out Of Memory* (OOM) error.

You can implement limits in two ways:

- Reactively: the system intervenes once it sees a violation.
- By enforcement: the system prevents the container from ever exceeding the limit.

Different runtimes can have different ways to implement the same restrictions.



NOTE

If you specify a limit for a resource, but do not specify any request, and no admission-time mechanism has applied a default request for that resource, then Kubernetes copies the limit you specified and uses it as the requested value for the resource.

1.2.2. Resource types

CPU and memory are both resource types. A resource type has a base unit. CPU represents compute processing and is specified in units of Kubernetes CPUs. Memory is specified in units of bytes.

CPU and memory are collectively referred to as compute resources, or resources. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as pods and services, are objects that can be read and modified through the Kubernetes API server.

1.2.3. Specifying resource requests and limits for pods and containers

For each container, you can specify resource limits and requests, including the following:

```
spec.containers[].resources.limits.cpu
spec.containers[].resources.limits.memory
spec.containers[].resources.requests.cpu
spec.containers[].resources.requests.memory
```

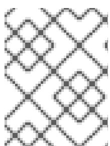
Although you can only specify requests and limits for individual containers, it is also useful to think about the overall resource requests and limits for a pod. For a particular resource, a pod resource request or limit is the sum of the resource requests or limits of that type for each container in the pod.

1.2.4. Resource units in Kubernetes

CPU resource units

Limits and requests for CPU resources are measured in CPU units. In Kubernetes, one CPU unit is equal to one physical processor core, or one virtual core, depending on whether the node is a physical host or a virtual machine running inside a physical machine.

Fractional requests are allowed. When you define a container with **spec.containers[].resources.requests.cpu** set to **0.5**, you are requesting half as much CPU time compared to if you asked for 1.0 CPU. For CPU resource units, the quantity expression 0.1 is equivalent to the expression 100m, which can be read as one hundred millicpu or one hundred millicores. millicpu and millicores mean the same thing. CPU resource is always specified as an absolute amount of resource, never as a relative amount. For example, 500m CPU represents the same amount of computing power whether that container runs on a single-core, dual-core, or 48-core machine.



NOTE

To specify CPU units less than 1.0 or 1000m you must use the milliCPU form. For example, use 5m, not 0.005 CPU.

Memory resource units

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these quantity suffixes: E, P, T, G, M, k. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

```
128974848, 129e6, 129M,  128974848000m, 123Mi
```

Pay attention to the case of the suffixes. If you request 400m of memory, this is a request for 0.4 bytes, not 400 mebibytes (400Mi) or 400 megabytes (400M).

Example CPU and memory specification

The following cluster has enough free resources to schedule a task pod with a dedicated 100m CPU and 250Mi. The cluster can also withstand bursts over that dedicated usage up to 2000m CPU and 2Gi memory.

```
spec:
  task_resource_requirements:
    requests:
      cpu: 100m
      memory: 250Mi
    limits:
      cpu: 2000m
      memory: 2Gi
```

Automation controller will not schedule jobs that use more resources than the limit set. If the task pod does use more resources than the limit set, the container is OOMKilled by Kubernetes and restarted.

1.2.5. Size recommendations for resource requests

All jobs that use a container group use the same pod specification. The pod specification includes the resource requests for the pod that runs the job.

All jobs use the same resource requests. The specified resource requests for your particular job on the pod specification affect how Kubernetes schedules the job pod based on resources available on worker nodes. These are the default values.

- One fork typically requires 100Mb of memory. This is set by using **system_task_forks_mem**. If your jobs have five forks, the job pod specification must request 500Mb of memory.
- For job templates that have a particularly high forks value or otherwise need larger resource requests, you should create a separate container group with a different pod spec that indicates larger resource requests. Then you can assign it to the job template. For example, a job template with the forks value of 50 must be paired with a container group that requests 5GB of memory.
- If the fork value for a job is high enough that no single pod would be able to contain the job, use the job slicing feature. This splits the inventory up such that the individual job “slices” fit in an automation pod provisioned by the container group.

CHAPTER 2. CONTROL PLANE ADJUSTMENTS

The control plane refers to the automation controller pods which contain the web and task containers that, among other things, provide the user interface and handle the scheduling and launching of jobs. On the automation controller custom resource, the number of *replicas* determines the number of automation controller pods in the automation controller deployment.

2.1. REQUESTS AND LIMITS FOR TASK CONTAINERS

You must set a value for the task container’s CPU and memory resource limits. For each job that is run in an execution node, processing must occur on the control plane to schedule, open, and receive callback events for that job.

For Operator deployments of automation controller, this control plane capacity usage is tracked on the **controlplane** instance group. The available capacity is determined based on the limits the user sets on the task container, using the **task_resource_requirements** field in the automation controller specification, or in the OpenShift UI, when creating automation controller.

You can also set memory and CPU resource limits that make sense for your cluster.

2.2. CONTAINERS RESOURCE REQUIREMENTS

You can configure the resource requirements of tasks and the web containers, at both the lower end (requests) and the upper end (limits). The execution environment control plane is used for project updates, but is normally the same as the default execution environment for jobs.

Setting resource requests and limits is a best practice because a container that has both defined is given a higher *Quality of Service* class. This means that if the underlying node is resource constrained and the cluster has to reap a pod to prevent running memory or other failure, the control plane pod is less likely to be reaped.

These requests and limits apply to the control pods for automation controller and if limits are set, determine the *capacity* of the instance. By default, controlling a job takes one unit of capacity. The memory and CPU limits of the task container are used to determine the capacity of control nodes. For more information about how this is calculated, see Resouce determination for capacity algorithm.

See also Jobs scheduled on the worker nodes

Name	Description	Default
web_resource_requirements	Web container resource requirements	requests: {CPU: 100m, memory: 128Mi}
task_resource_requirements	Task container resource requirements	requests: {CPU: 100m, memory: 128Mi}
ee_resource_requirements	EE control plane container resource requirements	requests: {CPU: 100m, memory: 128Mi}
redis_resource_requirements	Redis control plane container resource requirements	requests: {CPU:100m, memory: 128Mi}

The use of **topology_spread_constraints** to maximally spread control nodes onto separate underlying Kubernetes worker nodes is recommended. A reasonable set of requests and limits would be limits whose sum is equal to the actual resources on the node. If only **limits** are set, then the request is automatically set to be equal to the limit. But because some variability of resource usage between the containers in the control pod is permitted, you can set **requests** to a lower amount, for example to 25% of the resources available on the node. An example of container customization for a cluster where the worker nodes have 4 CPUs and 16 GB of RAM could be:

```
spec:
  ...
  web_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
  task_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 2000m
      memory: 4Gi
  redis_resource_requirements
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
  ee_resource_requirements:
    requests:
      cpu: 250m
      memory: 1Gi
    limits:
      cpu: 1000m
      memory: 4Gi
```

2.3. ALTERNATIVE CAPACITY LIMITING WITH AUTOMATION CONTROLLER SETTINGS

The capacity of a control node in OpenShift is determined by the memory and CPU limits. However, if these are not set then the capacity is determined by the memory and CPU detected by the pod on the filesystem, which are actually the CPU and memory of the underlying Kubernetes node.

This can lead to issues with overwhelming the underlying Kubernetes pod if the automation controller pod is not the only pod on that node. If you do not want to set limits directly on the task container, you can use **extra_settings**, see *Extra Settings* in Custom pod timeouts section for how to configure the following

```
SYSTEM_TASK_ABS_MEM = 3gi
SYSTEM_TASK_ABS_CPU = 750m
```

This acts as a soft limit within the application that enables automation controller to control how much work it attempts to run, while not risking any CPU throttling from Kubernetes itself, or being reaped if memory usage peaks above the required limit. These settings accept the same format accepted by resource requests and limits in the Kubernetes resource definition.

CHAPTER 3. SPECIFYING DEDICATED NODES

A Kubernetes cluster runs on top of many Virtual Machines or nodes (generally anywhere between 2 and 20 nodes). Pods can be scheduled on any of these nodes. When you create or schedule a new pod, use the **topology_spread_constraints** setting to configure how new pods are distributed across the underlying nodes when scheduled or created.

Do not schedule your pods on a single node, because if that node fails, the services that those pods provide also fails.

Schedule the control plane nodes to run on different nodes to the automation job pods. If the control plane pods share nodes with the job pods, the control plane can become resource starved and degrade the performance of the whole application.

3.1. ASSIGNING PODS TO SPECIFIC NODES

You can constrain the automation controller pods created by the operator to run on a certain subset of nodes.

- **node_selector** and **postgres_selector** constrain the automation controller pods to run only on the nodes that match all the specified key, or value, pairs.
- **tolerations** and **postgres_tolerations** enable the automation controller pods to be scheduled onto nodes with matching taints. See Taints and Toleration in the Kubernetes documentation for further details.

The following table shows the settings and fields that can be set on the automation controller's specification section of the YAML (or using the OpenShift UI form).

Name	Description	Default
postgres_image	Path of the image to pull	postgres
postgres_image_version	Image version to pull	13
node_selector	AutomationController pods' nodeSelector	""
topology_spread_constraints	AutomationController pods' topologySpreadConstraints	""
tolerations	AutomationController pods' tolerations	""
annotations	AutomationController pods' annotations	""
postgres_selector	Postgres pods' nodeSelector	""
postgres_tolerations	Postgres pods' tolerations	""

topology_spread_constraints can help optimize spreading your control plane pods across the compute nodes that match your node selector. For example, with the **maxSkew** parameter of this

option set to **100**, this means maximally spread across available nodes. So if there are three matching compute nodes and three pods, one pod will be assigned to each compute node. This parameter helps prevent the control plane pods from competing for resources with each other.

Example of a custom configuration for constraining controller pods to specific nodes

```
spec:
  ...
  node_selector: |
    disktype: ssd
    kubernetes.io/arch: amd64
    kubernetes.io/os: linux
  topology_spread_constraints: |
    - maxSkew: 100
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: "ScheduleAnyway"
      labelSelector:
        matchLabels:
          app.kubernetes.io/name: "<resourcename>"
  tolerations: |
    - key: "dedicated"
      operator: "Equal"
      value: "AutomationController"
      effect: "NoSchedule"
  postgres_selector: |
    disktype: ssd
    kubernetes.io/arch: amd64
    kubernetes.io/os: linux
  postgres_tolerations: |
    - key: "dedicated"
      operator: "Equal"
      value: "AutomationController"
      effect: "NoSchedule"
```

3.2. SPECIFY NODES FOR JOB EXECUTION

You can add a node selector to the container group pod specification to ensure they only run against certain nodes. First add a label to the nodes you want to run jobs against.

The following procedure adds a label to a node.

Procedure

1. List the nodes in your cluster, along with their labels:

   ```
   kubectl get nodes --show-labels
   ```

   The output is similar to this (shown here in a table):

Name	Status	Roles	Age	Version	Labels

Name	Status	Roles	Age	Version	Labels
worker0	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker0**
worker1	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker1**
worker2	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker2**

2. Choose one of your nodes, and add a label to it by using the following command:

```
kubectl label nodes <your-node-name> <aap_node_type>=<execution>
```

For example:

```
kubectl label nodes <your-node-name> disktype=ssd
```

where **<your-node-name>** is the name of your chosen node.

3. Verify that your chosen node has a **disktype=ssd** label:

```
kubectl get nodes --show-labels
```

4. The output is similar to this (shown here in a table):

Name	Status	Roles	Age	Version	Labels
worker0	Ready	<none>	1d	v1.13.0	**…disktype=ssd,kubernetes.io/hostname=worker0**
worker1	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker1**
worker2	Ready	<none>	1d	v1.13.0	**…,kubernetes.io/hostname=worker2**

You can see that the **worker0** node now has a **disktype=ssd** label.

5. In the automation controller UI, specify that label in the metadata section of your customized pod specification in the container group.

```
apiVersion: v1
kind: Pod
metadata:
  disktype: ssd
  namespace: ansible-automation-platform
spec:
  serviceAccountName: default
  automountServiceAccountToken: false
  nodeSelector:
    aap_node_type: execution
  containers:
    - image: >-
     registry.redhat.io/ansible-automation-platform-22/ee-supported-
rhel8@sha256:d134e198b179d1b21d3f067d745dd1a8e28167235c312cdc233860410ea3ec3e
      name: worker
      args:
        - ansible-runner
        - worker
        - '--private-data-dir=/runner'
      resources:
        requests:
          cpu: 250m
          memory: 100Mi
```

Extra settings

With **extra_settings**, you can pass many custom settings by using the awx-operator. The parameter **extra_settings** is appended to **/etc/tower/settings.py** and can be an alternative to the **extra_volumes** parameter.

Name	Description	Default
extra_settings	Extra settings	‘’

Example configuration of **extra_settings** parameter

```
spec:
  extra_settings:
    - setting: MAX_PAGE_SIZE
      value: "500"

    - setting: AUTH_LDAP_BIND_DN
      value: "cn=admin,dc=example,dc=com"

    - setting: SYSTEM_TASK_ABS_MEM
      value: "500"
```

3.3. CUSTOM POD TIMEOUTS

A container group job in automation controller transitions to the **running** state just before you submit the pod to the Kubernetes API. Automation controller then expects the pod to enter the **Running** state before **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** seconds has elapsed. You can set

AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT to a higher value if you want automation controller to wait for longer before canceling jobs that fail to enter the **Running** state. **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** is how long automation controller waits from creation of a pod until the Ansible work begins in the pod. You can also extend the time if the pod cannot be scheduled because of resource constraints. You can do this using **extra_settings** on the automation controller specification. The default value is two hours.

This is used if you are consistently launching many more jobs than Kubernetes can schedule, and jobs are spending periods longer than **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT** in *pending*.

Jobs are not launched until control capacity is available. If many more jobs are being launched than the container group has capacity to run, consider scaling up your Kubernetes worker nodes.

3.4. JOBS SCHEDULED ON THE WORKER NODES

Both automation controller and Kubernetes play a role in scheduling a job.

When a job is launched, its dependencies are fulfilled, meaning any project updates or inventory updates are launched by automation controller as required by the job template, project, and inventory settings.

If the job is not blocked by other business logic in automation controller and there is control capacity in the control plane to start the job, the job is submitted to the dispatcher. The default settings of the "cost" to control a job is 1 *capacity*. So, a control pod with 100 capacity is able to control up to 100 jobs at a time. Given control capacity, the job transitions from *pending* to *waiting*.

The dispatcher, which is a background process in the control plan pod, starts a worker process to run the job. This communicates with the Kubernetes API using a service account associated with the container group and uses the pod specification as defined on the Container Group in automation controller to provision the pod. The job status in automation controller is shown as *running*.

Kubernetes now schedules the pod. A pod can remain in the *pending* state for **AWX_CONTAINER_GROUP_POD_PENDING_TIMEOUT**. If the pod is denied through a **ResourceQuota**, the job starts over at *pending*. You can configure a resource quota on a namespace to limit how many resources may be consumed by pods in the namespace. For further information about ResourceQuotas, see Resource Quotas.

CHAPTER 4. CONFIGURING ANSIBLE AUTOMATION CONTROLLER ON OPENSHIFT CONTAINER PLATFORM

During a Kubernetes upgrade, automation controller must be running.

4.1. MINIMIZING DOWNTIME DURING OPENSHIFT CONTAINER PLATFORM UPGRADE

Make the following configuration changes in automation controller to minimize downtime during the upgrade.

Prerequisites

- Ansible Automation Platform 2.4
- Ansible automation controller 4.4
- OpenShift Container Platform
  - > 4.10.42
  - > 4.11.16
  - > 4.12.0
- High availability (HA) deployment of Postgres
- Multiple worker node that automation controller pods can be scheduled on

Procedure

1. Enable **RECEPTOR_KUBE_SUPPORT_RECONNECT** in AutomationController specification:

```
apiVersion: automationcontroller.ansible.com/v1beta1
kind: AutomationController
metadata:
  ...
spec:
  ...
  ee_extra_env: |
    - name: RECEPTOR_KUBE_SUPPORT_RECONNECT
      value: enabled
    ```
```

2. Enable the graceful termination feature in AutomationController specification:

```
termination_grace_period_seconds: <time to wait for job to finish>
```

3. Configure **podAntiAffinity** for web and task the pod to spread out the deployment in AutomationController specification:

```
task_affinity:
    podAntiAffinity:
```
```

```
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - awx-task
          topologyKey: topology.kubernetes.io/zone
        weight: 100
  web_affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - awx-web
          topologyKey: topology.kubernetes.io/zone
        weight: 100
```

4. Configure **PodDisruptionBudget** in OpenShift Container Platform:

```
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-job-pods
spec:
  maxUnavailable: 0
  selector:
    matchExpressions:
      - key: ansible-awx-job-id
        operator: Exists
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-web-pods
spec:
  minAvailable: 1
  selector:
    matchExpressions:
      - key: app.kubernetes.io/name
        operator: In
        values:
          - <automationcontroller_instance_name>-web
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: automationcontroller-task-pods
spec:
  minAvailable: 1
```

```
selector:
  matchExpressions:
    - key: app.kubernetes.io/name
      operator: In
      values:
        - <automationcontroller_instance_name>-task
```