Red Hat OpenShift Pipelines 1.13

Managing resource use

Managing resource consumption in OpenShift Pipelines
Managing resource consumption in OpenShift Pipelines
Abstract

This document provides information about managing resource consumption in OpenShift Pipelines.
# Table of Contents

CHAPTER 1. REDUCING RESOURCE CONSUMPTION OF OPENShift PIPELINES .......................... 3
   1.1. UNDERSTANDING RESOURCE CONSUMPTION IN PIPELINES ............................ 3
   1.2. MITIGATING EXTRA RESOURCE CONSUMPTION IN PIPELINES ......................... 4
   1.3. ADDITIONAL RESOURCES ............................................................................. 5

CHAPTER 2. SETTING COMPUTE RESOURCE QUOTA FOR OPENShift PIPELINES .................. 6
   2.1. ALTERNATIVE APPROACHES FOR LIMITING COMPUTE RESOURCE CONSUMPTION IN OPENShift PIPELINES ................................................................. 6
   2.2. SPECIFYING PIPELINES RESOURCE QUOTA USING PRIORITY CLASS .................. 7
   2.3. ADDITIONAL RESOURCES ............................................................................. 11
CHAPTER 1. REDUCING RESOURCE CONSUMPTION OF OPENSIFT PIPELINES

If you use clusters in multi-tenant environments you must control the consumption of CPU, memory, and storage resources for each project and Kubernetes object. This helps prevent any one application from consuming too many resources and affecting other applications.

To define the final resource limits that are set on the resulting pods, Red Hat OpenShift Pipelines use resource quota limits and limit ranges of the project in which they are executed.

To restrict resource consumption in your project, you can:

- Set and manage resource quotas to limit the aggregate resource consumption.
- Use limit ranges to restrict resource consumption for specific objects, such as pods, images, image streams, and persistent volume claims.

1.1. UNDERSTANDING RESOURCE CONSUMPTION IN PIPELINES

Each task consists of a number of required steps to be executed in a particular order defined in the steps field of the Task resource. Every task runs as a pod, and each step runs as a container within that pod.

Steps are executed one at a time. The pod that executes the task only requests enough resources to run a single container image (step) in the task at a time, and thus does not store resources for all the steps in the task.

The Resources field in the steps spec specifies the limits for resource consumption. By default, the resource requests for the CPU, memory, and ephemeral storage are set to BestEffort (zero) values or to the minimums set through limit ranges in that project.

Example configuration of resource requests and limits for a step

```yaml
spec:
  steps:
    - name: <step_name>
      resources:
        requests:
          memory: 2Gi
          cpu: 600m
        limits:
          memory: 4Gi
          cpu: 900m
```

When the LimitRange parameter and the minimum values for container resource requests are specified in the project in which the pipeline and task runs are executed, Red Hat OpenShift Pipelines looks at all the LimitRange values in the project and uses the minimum values instead of zero.

Example configuration of limit range parameters at a project level

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: <limit_container_resource>
```
1.2. MITIGATING EXTRA RESOURCE CONSUMPTION IN PIPELINES

When you have resource limits set on the containers in your pod, OpenShift Container Platform sums up the resource limits requested as all containers run simultaneously.

To consume the minimum amount of resources needed to execute one step at a time in the invoked task, Red Hat OpenShift Pipelines requests the maximum CPU, memory, and ephemeral storage as specified in the step that requires the most amount of resources. This ensures that the resource requirements of all the steps are met. Requests other than the maximum values are set to zero.

However, this behavior can lead to higher resource usage than required. If you use resource quotas, this could also lead to unschedulable pods.

For example, consider a task with two steps that uses scripts, and that does not define any resource limits and requests. The resulting pod has two init containers (one for entrypoint copy, the other for writing scripts) and two containers, one for each step.

OpenShift Container Platform uses the limit range set up for the project to compute required resource requests and limits. For this example, set the following limit range in the project:

```yaml
spec:
  limits:
  - max:
      memory: 1Gi
      cpu: 500m
    min:
      memory: 500Mi
      cpu: 200m
  default:
    memory: 100Mi
    cpu: 100m
  defaultRequest:
    memory: 100Mi
    cpu: 100m
  type: Container
...
```

In this scenario, each init container uses a request memory of 1Gi (the max limit of the limit range), and each container uses a request memory of 500Mi. Thus, the total memory request for the pod is 2Gi.

If the same limit range is used with a task of ten steps, the final memory request is 5Gi, which is higher than what each step actually needs, that is 500Mi (since each step runs after the other).
Thus, to reduce resource consumption of resources, you can:

- Reduce the number of steps in a given task by grouping different steps into one bigger step, using the script feature, and the same image. This reduces the minimum requested resource.

- Distribute steps that are relatively independent of each other and can run on their own to multiple tasks instead of a single task. This lowers the number of steps in each task, making the request for each task smaller, and the scheduler can then run them when the resources are available.

1.3. ADDITIONAL RESOURCES

- Setting compute resource quota for OpenShift Pipelines
- Resource quotas per project
- Restricting resource consumption using limit ranges
- Resource requests and limits in Kubernetes
CHAPTER 2. SETTING COMPUTE RESOURCE QUOTA FOR OPENSHIFT PIPELINES

A ResourceQuota object in Red Hat OpenShift Pipelines controls the total resource consumption per namespace. You can use it to limit the quantity of objects created in a namespace, based on the type of the object. In addition, you can specify a compute resource quota to restrict the total amount of compute resources consumed in a namespace.

However, you might want to limit the amount of compute resources consumed by pods resulting from a pipeline run, rather than setting quotas for the entire namespace. Currently, Red Hat OpenShift Pipelines does not enable you to directly specify the compute resource quota for a pipeline.

2.1. ALTERNATIVE APPROACHES FOR LIMITING COMPUTE RESOURCE CONSUMPTION IN OPENSHIFT PIPELINES

To attain some degree of control over the usage of compute resources by a pipeline, consider the following alternative approaches:

- Set resource requests and limits for each step in a task.
  
  **Example:** Set resource requests and limits for each step in a task.

  ```yaml
  ...
  spec:
    steps:
      - name: step-with-limits
        resources:
          requests:
            memory: 1Gi
            cpu: 500m
          limits:
            memory: 2Gi
            cpu: 800m
  ...
  ```

- Set resource limits by specifying values for the LimitRange object. For more information on LimitRange, refer to *Restrict resource consumption with limit ranges*.

- Reduce pipeline resource consumption.

- Set and manage resource quotas per project.

- Ideally, the compute resource quota for a pipeline should be same as the total amount of compute resources consumed by the concurrently running pods in a pipeline run. However, the pods running the tasks consume compute resources based on the use case. For example, a Maven build task might require different compute resources for different applications that it builds. As a result, you cannot predetermine the compute resource quotas for tasks in a generic pipeline. For greater predictability and control over usage of compute resources, use customized pipelines for different applications.
NOTE

When using Red Hat OpenShift Pipelines in a namespace configured with a ResourceQuota object, the pods resulting from task runs and pipeline runs might fail with an error, such as: failed quota: <quota name> must specify cpu, memory.

To avoid this error, do any one of the following:

- (Recommended) Specify a limit range for the namespace.
- Explicitly define requests and limits for all containers.

For more information, refer to the issue and the resolution.

If your use case is not addressed by these approaches, you can implement a workaround by using a resource quota for a priority class.

2.2. SPECIFYING PIPELINES RESOURCE QUOTA USING PRIORITY CLASS

A PriorityClass object maps priority class names to the integer values that indicates their relative priorities. Higher values increase the priority of a class. After you create a priority class, you can create pods that specify the priority class name in their specifications. In addition, you can control a pod’s consumption of system resources based on the pod’s priority.

Specifying resource quota for a pipeline is similar to setting a resource quota for the subset of pods created by a pipeline run. The following steps provide an example of the workaround by specifying resource quota based on priority class.

Procedure

1. Create a priority class for a pipeline.

Example: Priority class for a pipeline

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: pipeline1-pc
value: 1000000
description: "Priority class for pipeline1"
```

2. Create a resource quota for a pipeline.

Example: Resource quota for a pipeline

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pipeline1-rq
spec:
  hard:
    cpu: "1000"
    memory: 200Gi
```
3. Verify the resource quota usage for the pipeline.

**Example: Verify resource quota usage for the pipeline**

```bash
$ oc describe quota
```

**Sample output**

```
Name:       pipeline1-rq
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     1k
memory      0     200Gi
pods        0     10
```

Because pods are not running, the quota is unused.

4. Create the pipelines and tasks.

**Example: YAML for the pipeline**

```yaml
apiVersion: tekton.dev/v1beta1
class: Pipeline
metadata:
  name: maven-build
spec:
  workspaces:
  - name: local-maven-repo
    resources:
      - name: app-git
        type: git
  tasks:
  - name: build
    taskRef:
      name: mvn
    resources:
      inputs:
        - name: source
          resource: app-git
    params:
      - name: GOALS
        value: ["package"]
  workspaces:
    - name: maven-repo
      workspace: local-maven-repo
    - name: int-test
```
taskRef:
  name: mvn
runAfter: ["build"]
resources:
  inputs:
  - name: source
    resource: app-git
params:
  - name: GOALS
    value: ["verify"]
workspaces:
  - name: maven-repo
    workspace: local-maven-repo
  - name: gen-report
    taskRef:
      name: mvn
      runAfter: ["build"]
      resources:
        inputs:
        - name: source
          resource: app-git
        params:
        - name: GOALS
          value: ["site"]
        workspaces:
        - name: maven-repo
          workspace: local-maven-repo

Example: YAML for a task in the pipeline

apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: mvn
spec:
  workspaces:
  - name: maven-repo
  resources:
    inputs:
    - name: source
      type: git
    params:
    - name: GOALS
      description: The Maven goals to run
      type: array
      default: ["package"]
  steps:
  - name: mvn
    image: gcr.io/cloud-builders/mvn
    workingDir: /workspace/source
    command: ["/usr/bin/mvn"]
    args:
    - -Dmaven.repo.local=$(workspaces.maven-repo.path)
    - "$(params.GOALS)"
5. Create and start the pipeline run.

Example: YAML for a pipeline run

```yaml
apiVersion: tekton.dev/v1beta1
class: PipelineRun
description:
generateName: petclinic-run-
spec:
pipelineRef:
  name: maven-build
podTemplate:
  priorityClassName: pipeline1-pc
workspaces:
- name: local-maven-repo
  emptyDir: {}
resources:
- name: app-git
resourceSpec:
type: git
params:
- name: url
  value: https://github.com/spring-projects/spring-petclinic
```

NOTE

The pipeline run might fail with an error: failed quota: <quota name> must specify cpu, memory.

To avoid this error, set a limit range for the namespace, where the defaults from the LimitRange object apply to pods created during the build process.

For more information about setting limit ranges, refer to Restrict resource consumption with limit ranges in the Additional resources section.

6. After the pods are created, verify the resource quota usage for the pipeline run.

Example: Verify resource quota usage for the pipeline

```bash
$ oc describe quota
```

Sample output

```text
Name:       pipeline1-rq  
Namespace:  default  
Resource    Used  Hard  
--------    ----  ----  
cpu         500m  1k  
memory      10Gi  200Gi  
pods        1     10  
```

The output indicates that you can manage the combined resource quota for all concurrent running pods belonging to a priority class, by specifying the resource quota per priority class.
2.3. ADDITIONAL RESOURCES

- Restrict resource consumption with limit ranges
- Resource quotas in Kubernetes
- Limit ranges in Kubernetes
- Resource requests and limits in Kubernetes