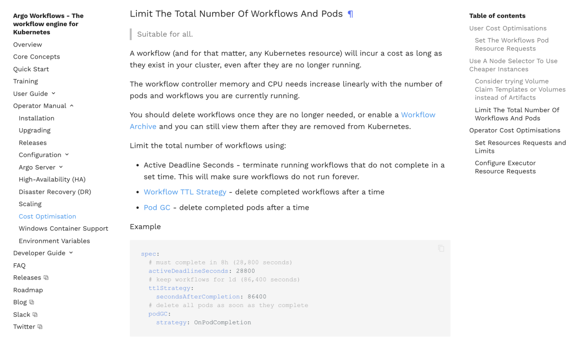
# Argo WorkflowGCPod

**Content Menu**

- TTLStrategy
- PodGC

WorkflowPodArgo Workflowhttps://argoproj.github.io/argo-workflows/cost-optimisation/#limit-the-total-number-of-workflows-and-pods



**Argo Workflows - The workflow engine for Kubernetes**

Overview
Core Concepts
Quick Start
Training
User Guide
Operator Manual
  Installation
  Upgrading
  Releases
  Configuration
  Argo Server
  High-Availability (HA)
  Disaster Recovery (DR)
  Scaling
  Cost Optimisation
  Windows Container Support
  Environment Variables
Developer Guide
FAQ
Releases
Roadmap
Blog
Slack
Twitter

Limit The Total Number Of Workflows And Pods ¶

> Suitable for all.

A workflow (and for that matter, any Kubernetes resource) will incur a cost as long as they exist in your cluster, even after they are no longer running.

The workflow controller memory and CPU needs increase linearly with the number of pods and workflows you are currently running.

You should delete workflows once they are no longer needed, or enable a Workflow Archive and you can still view them after they are removed from Kubernetes.

Limit the total number of workflows using:

- Active Deadline Seconds - terminate running workflows that do not complete in a set time. This will make sure workflows do not run forever.
- Workflow TTL Strategy - delete completed workflows after a time
- Pod GC - delete completed pods after a time

Example

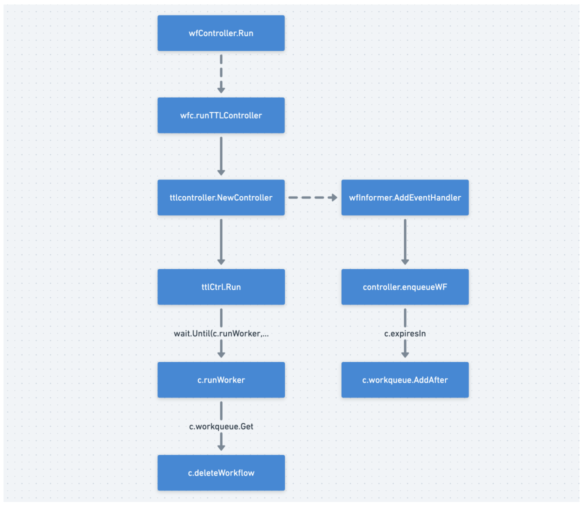
```yaml
spec:
  # must complete in 8h (28,800 seconds)
  activeDeadlineSeconds: 28800
  # keep workflows for 1d (86,400 seconds)
  ttlStrategy:
    secondsAfterCompletion: 86400
  # delete all pods as soon as they complete
  podGC:
    strategy: OnPodCompletion
```

**Table of contents**

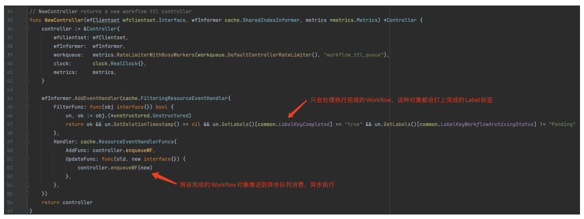
User Cost Optimisations
  Set The Workflows Pod Resource Requests
Use A Node Selector To Use Cheaper Instances
  Consider trying Volume Claim Templates or Volumes instead of Artifacts
  Limit The Total Number Of Workflows And Pods
Operator Cost Optimisations
  Set Resources Requests and Limits
  Configure Executor Resource Requests

TTLStrategyPodGCWorkflowPod

## TTLStrategy



- wfc.runTTLController Argo Workflow ControllerTTL
- ttlcontroller.NewControllerTTLControllerEventHandlerWorkflow

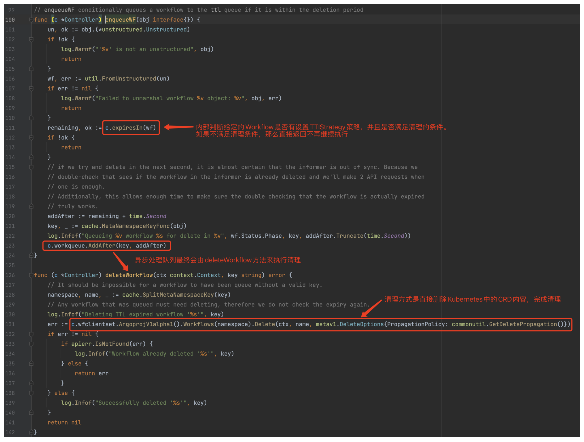


```go
// NewController returns a new workflow ttl controller
func NewController(wfClientset wfclientset.Interface, wfInformer cache.SharedIndexInformer, metrics *metrics.Metrics) *Controller {
    controller := &Controller{
        wfclientset: wfClientset,
        wfInformer:  wfInformer,
        workqueue:   metrics.RateLimiterWithBusyWorkers(workqueue.DefaultControllerRateLimiter(), "workflow_ttl_queue"),
        clock:       clock.RealClock{},
        metrics:     metrics,
    }

    wfInformer.AddEventHandler(cache.FilteringResourceEventHandler{
        FilterFunc: func(obj interface{}) bool {
            un, ok := obj.(*unstructured.Unstructured)
            return ok && un.GetDeletionTimestamp() == nil && un.GetLabels()[common.LabelKeyCompleted] == "true" && un.GetLabels()[common.LabelKeyWorkflowArchivingStatus] != "Pending"
        },
        Handler: cache.ResourceEventHandlerFuncs{
            AddFunc: controller.enqueueWF,
            UpdateFunc: func(old, new interface{}) {
                controller.enqueueWF(new)
            },
        },
    })
    return controller
}
```

只会处理执行完成的 Workflow，这种对象都会打上完成的 Label 标签

将该完成的 Workflow 对象推送到异步队列消费，异步执行



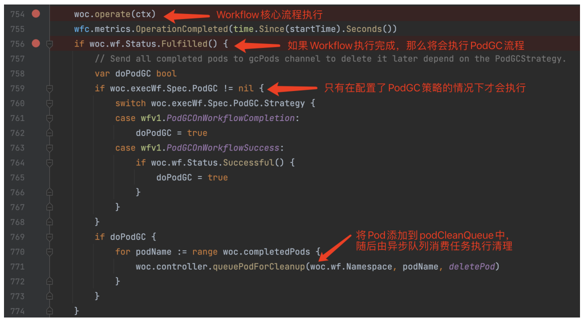
```go
// enqueueWF conditionally queues a workflow to the ttl queue if it is within the deletion period
func (c *Controller) enqueueWF(obj interface{}) {
    un, ok := obj.(*unstructured.Unstructured)
    if !ok {
        log.Warnf("'%v' is not an unstructured", obj)
        return
    }
    wf, err := util.FromUnstructured(un)
    if err != nil {
        log.Warnf("Failed to unmarshal workflow %v object: %v", obj, err)
        return
    }
    remaining, ok := c.expiresIn(wf)
    if !ok {
        return
    }
    // if we try and delete in the next second, it is almost certain that the informer is out of sync. Because we
    // double-check that sees if the workflow in the informer is already deleted and we'll make 2 API requests when
    // one is enough.
    // Additionally, this allows enough time to make sure the double checking that the workflow is actually expired
    // truly works.
    addAfter := remaining + time.Second
    key, _ := cache.MetaNamespaceKeyFunc(obj)
    log.Infof("Queueing %v workflow %s for delete in %v", wf.Status.Phase, key, addAfter.Truncate(time.Second))
    c.workqueue.AddAfter(key, addAfter)
}

func (c *Controller) deleteWorkflow(ctx context.Context, key string) error {
    // It should be impossible for a workflow to have been queue without a valid key.
    namespace, name, _ := cache.SplitMetaNamespaceKey(key)
    // Any workflow that was queued must need deleting, therefore we do not check the expiry again.
    log.Infof("Deleting TTL expired workflow '%s'", key)
    err := c.wfclientset.ArgoprojV1alpha1().Workflows(namespace).Delete(ctx, name, metav1.DeleteOptions{PropagationPolicy: commonutil.GetDeletePropagation()})
    if err != nil {
        if apierr.IsNotFound(err) {
            log.Infof("Workflow already deleted '%s'", key)
        } else {
            return err
        }
    } else {
        log.Infof("Successfully deleted '%s'", key)
    }
    return nil
}
```

内部判断给定的 Workflow 是否有设置 TTlStrategy 策略，并且是否满足清理的条件。
如果不满足清理条件，那么直接返回不再继续执行
异步处理队列最终会由 deleteWorkflow 方法来执行清理
清理方式是直接删除 Kubernetes 中的 CRD 内容，完成清理

## PodGC

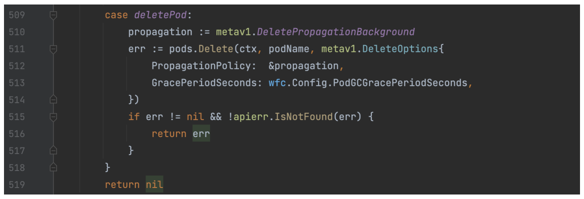
podCleanQueuePodArgo Workflow Controllerwoc.operateWorkflowPodGC



```go
    woc.operate(ctx)
    wfc.metrics.OperationCompleted(time.Since(startTime).Seconds())
    if woc.wf.Status.Fulfilled() {
        // Send all completed pods to gcPods channel to delete it later depend on the PodGCStrategy.
        var doPodGC bool
        if woc.execWf.Spec.PodGC != nil {
            switch woc.execWf.Spec.PodGC.Strategy {
            case wfv1.PodGCOnWorkflowCompletion:
                doPodGC = true
            case wfv1.PodGCOnWorkflowSuccess:
                if woc.wf.Status.Successful() {
                    doPodGC = true
                }
            }
        }
        if doPodGC {
            for podName := range woc.completedPods {
                woc.controller.queuePodForCleanup(woc.wf.Namespace, podName, deletePod)
            }
        }
    }
```

Workflow 核心流程执行
如果 Workflow 执行完成，那么将会执行 PodGC 流程
只有在配置了 PodGC 策略的情况下才会执行
将 Pod 添加到 podCleanQueue 中，
随后由异步队列消费任务执行清理

deletePodKubernetesPod



```go
        case deletePod:
            propagation := metav1.DeletePropagationBackground
            err := pods.Delete(ctx, podName, metav1.DeleteOptions{
                PropagationPolicy:  &propagation,
                GracePeriodSeconds: wfc.Config.PodGCGracePeriodSeconds,
            })
            if err != nil && !apierr.IsNotFound(err) {
                return err
            }
        }
        return nil
```

✅ CompletionSuccessCompletionWorkflowWorkflowRunning/Appending/UnknownSuccessWorkflowexit code0