

# Argo Workflow From OpenWriteup

## Table of contents

Introduction: .....	4
Overview .....	5
Why Argo Workflow .....	5
Argo Workflow Architecture .....	5
Argo Workflow Overview: .....	6
Workflow controller architecture.....	7
Argo Server and Argo UI.....	8
Installation.....	9
Setup Argo Cli.....	10
<b>Installation options</b> .....	10
Controller and Server .....	11
Workflow RBAC.....	12
Workflow Controller Security.....	13
Controller Permissions .....	13
User Permissions .....	13
UI Access.....	14
<b>Argo Server</b> .....	14
Access through UI.....	15
The Workflow.....	17
Definition .....	17
Workflow Spec.....	17
Template.....	19
Script type .....	20
Resource Template.....	21
Suspend Template .....	24
Step Template .....	25
Template DAG .....	28
WorkflowTemplates .....	31
Template vs WorkflowTemplate.....	31
WorkflowTemplate Spec.....	32
Valid WorkflowTemplates (v 2.7) .....	33
Valid for (v 2.4-2.6).....	34

Adding labels/annotations to Workflows with workflowMetadata (v 2.10 and after).....	34
Working with parameters .....	36
Cluster Workflow Template.....	36
<b>Referencing other ClusterWorkflowTemplates .....</b>	<b>36</b>
CronWorkflows.....	38
Timezone parameter .....	39
Add other parameters in cronjob workflow.....	40
Command line options for cronworkflow .....	42
Template Types .....	43
Argo Agent .....	44
Container Set Template.....	46
The Emissary Executor .....	49
Lopsided requests .....	51
ContainerSet Retry .....	54
Artifacts.....	55
Input Parameters .....	55
Run from command line with inputs .....	56
DAG: Previous Step Outputs As Inputs .....	57
Key-Only Artifacts.....	58
Configuring Artifactory .....	58
Setup the S3 .....	59

Introduction:

Mail us: [info@openwriteup.com](mailto:info@openwriteup.com)

## Overview

Argo workflow is an opensource container native workflow engine for orchestration jobs on Kubernetes. Argo workflow is implemented as a Kubernetes CRD (Custom Resource Definition).

As the part of prerequisites, Kubernetes environment must be available to setup Argo Workflow, since it is going to implement the workflow on top of it.

## Why Argo Workflow

In case of complex workflow, when we have multiple stages in Kubernetes environment. Argo workflow solves that complex workflow multiple steps, dependencies and conditions.

It provides a UI interface, where we can see all the activities happening on the argo side. Logging and monitoring on each stage is well integrated. Scaling up and version control is also well managed.

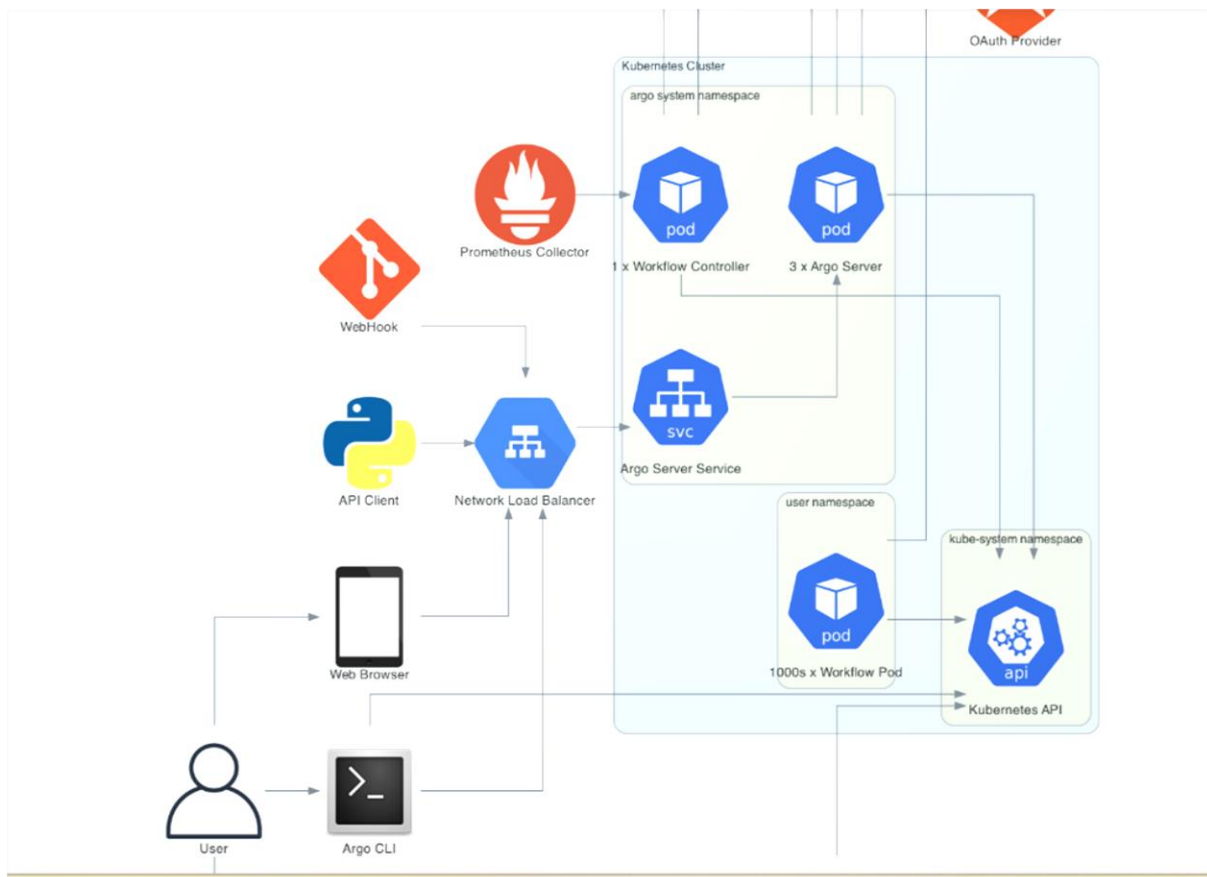
As a solution, Argo workflow provides single windows for complex workflow on Kubernetes environment

## Argo Workflow Architecture

The following diagram shows the components of the Argo Workflows architecture. There are two Deployments: Workflow Controller and Argo Server

Workflow Controller: It takes care of reconciliation process. It has bunch of workers who are responsible for handling tasks, these tasks are to-do-list called workers, put in a special queue whenever something new added or updated.

Argo Server: Handles all the workflow, coordinates task and ensures everything running fine. It is backbone of Argo workflow.



In the above diagram, we have Argo namespace. In Argo namespace Workflow controller and Argo server is running, and Argo server is connected to Argo Server Service.

As well to communicate Argo, we have Argo cli and Argo UI option are available.

### Argo Workflow Overview:

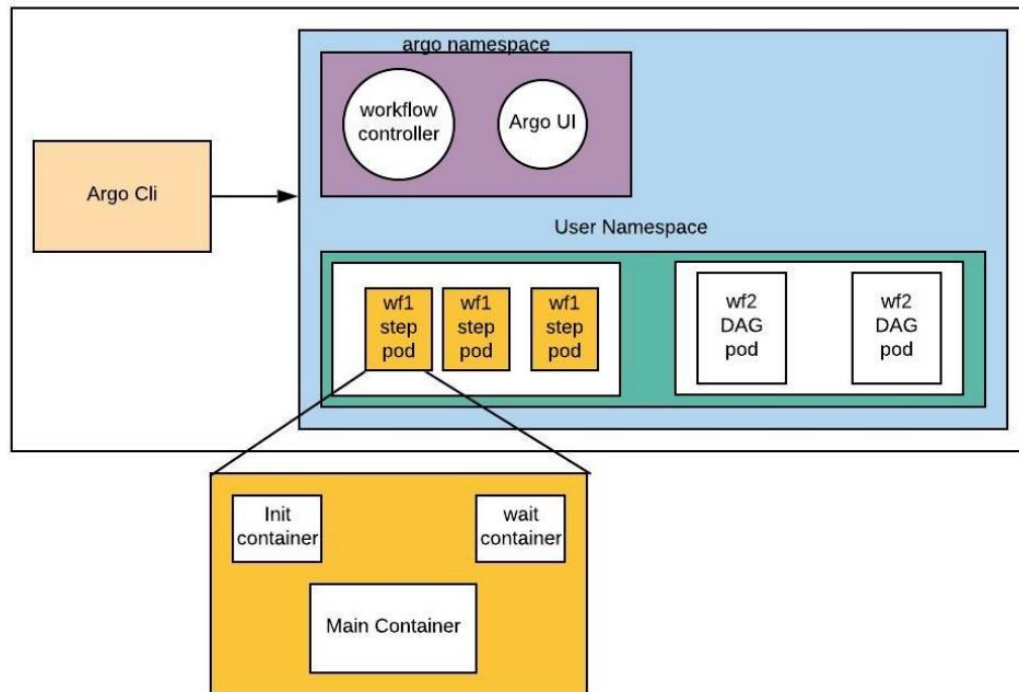
The diagram below provides a little more detail as far as namespaces. The Workflow Controller and Argo Server both run in the argo namespace. Assuming Argo Workflows was installed as a Cluster Install or as a Managed Namespace Install, the Workflows and the Pods generated from them run in a separate namespace.

The internals of a Pod are also shown. Each Step and each DAG Task cause a Pod to be generated, and each of these is composed of 3 containers:

- main container runs the Image that the user indicated, where the argoexec utility is volume mounted and serves as the main command which calls the configured Command as a sub-process
- init container is an InitContainer, fetching artifacts and parameters and making them available to the main container

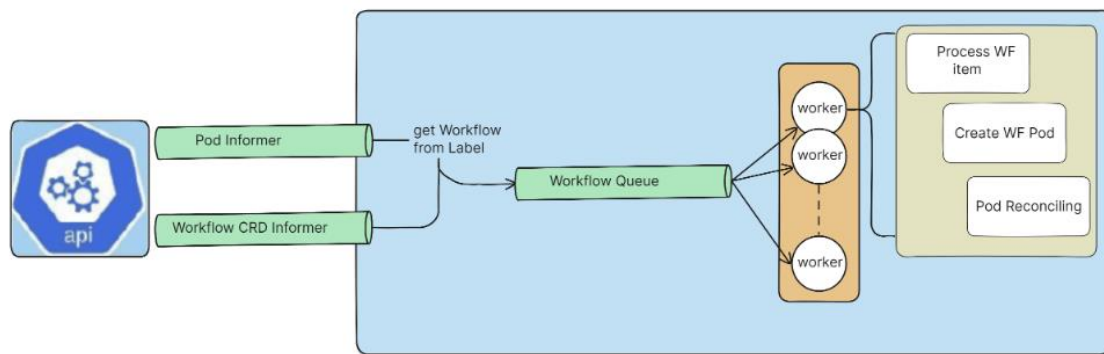
- wait container performs tasks that are needed for clean up, including saving off parameters and artifacts

## ARGO Workflow Overview

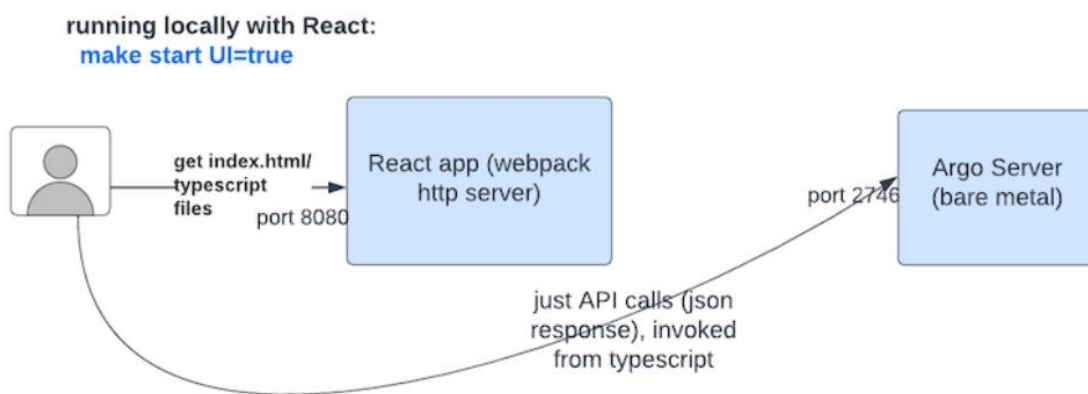


### Workflow controller architecture

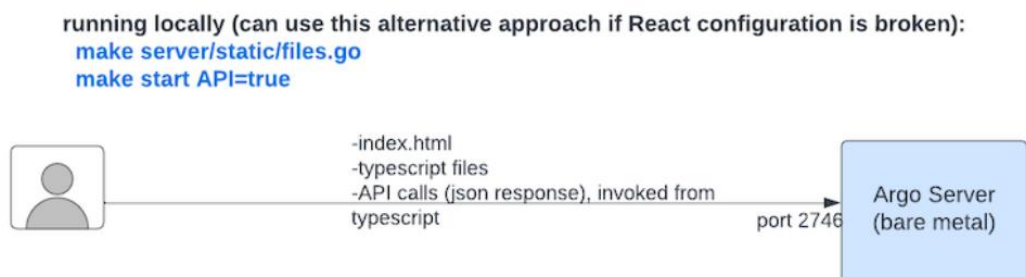
The following diagram shows the process for reconciliation, whereby a set of worker goroutines process the Workflows which have been added to a Workflow queue based on adds and updates to Workflows and Workflow Pods. Note that in addition to the Informers shown, there are Informers for the other CRDs that Argo Workflows uses as well. You can find this code in [workflow/controller/controller.go](https://github.com/argoproj/argo-workflows/blob/master/controller/controller.go). Note that the controller only ever processes a single Workflow at a time.



## Argo Server and Argo UI

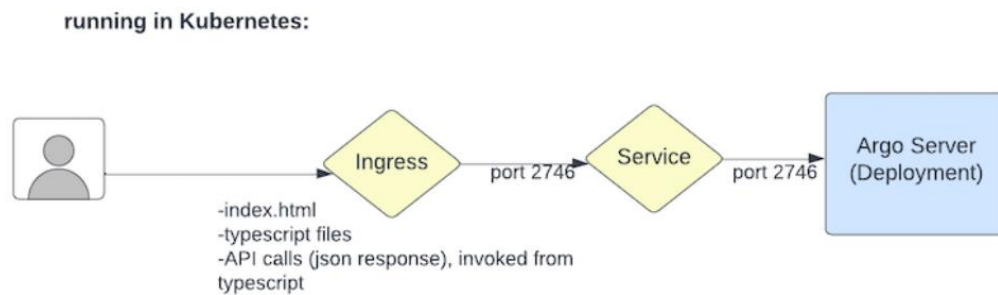


The second diagram is an alternative approach for rare occasions that the React files are broken and you're doing local development. In this case, everything is served from the Argo Server at port 2746.





The third diagram shows how things are configured for a Kubernetes environment. It is similar to the second diagram in that the Argo Server hosts everything for the UI.



## Installation

In this session we will be using ubuntu 22.04 server, All the prerequisites already followed:

- Docker installation: [https://www.openwriteup.com/?page\\_id=785](https://www.openwriteup.com/?page_id=785)
- Kubernetes setup: [https://www.openwriteup.com/?page\\_id=866](https://www.openwriteup.com/?page_id=866)
- Untainted the master (As we are using single host):  
[https://www.openwriteup.com/?page\\_id=887](https://www.openwriteup.com/?page_id=887)

**Note:** In case you are doing first time, please follow the guideline provided for docker and Kubernetes

## Setup Argo Cli

```
# Download the binary
curl -sLO https://github.com/argoproj/argo-workflows/releases/download/v3.5.6/argo-linux-amd64.gz

# Unzip
gunzip argo-linux-amd64.gz

# Make binary executable
chmod +x argo-linux-amd64

# Move binary to path
mv ./argo-linux-amd64 /usr/local/bin/argo

# Test installation
argo version
```

```
Destroy complete! Resources: 2 destroyed.
root@devvm:~/base/system# argo version
argo: v3.2.6
  BuildDate: 2021-12-17T20:59:31Z
  GitCommit: db7d90a1f609685cfda73644155854b06fa5d28b
  GitTreeState: clean
  GitTag: v3.2.6
  GoVersion: go1.16.12
  Compiler: gc
  Platform: linux/amd64
```

## Installation options

Determine your base installation option.

- A **cluster install** will watch and execute workflows in all namespaces. This is the default installation option when installing using the official release manifests.
- A **namespace install** only executes workflows in the namespace it is installed in (typically argo). Look for namespace-install.yaml in the [release assets](#).
- A **managed namespace install**: only executes workflows in a separate namespace from the one it is installed in.

## Controller and Server

```
kubectl create namespace argo
kubectl apply -n argo -f https://github.com/argoproj/argo-
workflows/releases/download/v3.5.6/install.yaml
```

```
root@devvm: /home/amlc/argoworkflows# kubectl create namespace argo
kubectl apply -n argo -f https://github.com/argoproj/argo-workflows/releases/download/v3.5
namespace/argo created
customresourcedefinition.apiextensions.k8s.io/clusterworkflowtemplates.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/cronworkflows.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowartifactgctasks.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workfloweventbindings.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflows.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtaskresults.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtasksets.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtemplates.argoproj.io created
serviceaccount/argo created
serviceaccount/argo-server created
role.rbac.authorization.k8s.io/argo-role created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-admin created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-edit created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-view created
clusterrole.rbac.authorization.k8s.io/argo-cluster-role created
clusterrole.rbac.authorization.k8s.io/argo-server-cluster-role created
rolebinding.rbac.authorization.k8s.io/argo-binding created
clusterrolebinding.rbac.authorization.k8s.io/argo-binding created
clusterrolebinding.rbac.authorization.k8s.io/argo-server-binding created
configmap/workflow-controller-configmap created
service/argo-server created
priorityclass.scheduling.k8s.io/workflow-controller created
deployment.apps/argo-server created
deployment.apps/workflow-controller created
```

```

root@devvm: /home/amit/argowf# kubectl create namespace argo
namespace/argo created
customresourcedefinition.apiextensions.k8s.io/clusterworkflowtemplates.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/cronworkflows.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowartifacts.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workfloweventbindings.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflows.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtaskresults.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtasksets.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/workflowtemplates.argoproj.io created
serviceaccount/argo created
serviceaccount/argo-server created
role.rbac.authorization.k8s.io/argo-role created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-admin created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-edit created
clusterrole.rbac.authorization.k8s.io/argo-aggregate-to-view created
clusterrole.rbac.authorization.k8s.io/argo-cluster-role created
clusterrole.rbac.authorization.k8s.io/argo-server-cluster-role created
rolebinding.rbac.authorization.k8s.io/argo-binding created
clusterrolebinding.rbac.authorization.k8s.io/argo-binding created
clusterrolebinding.rbac.authorization.k8s.io/argo-server-binding created
configmap/workflow-controller-configmap created
service/argo-server created
priorityclass.scheduling.k8s.io/workflow-controller created
deployment.apps/argo-server created
deployment.apps/workflow-controller created
root@devvm: /home/amit/argowf# kubectl get pods -n argo

```

NAME	READY	STATUS	RESTARTS	AGE
argo-server-579d446586-w526j	1/1	Running	0	37s
workflow-controller-566fcc48f-pdm8b	1/1	Running	0	37s

```

root@devvm: /home/amit/argowf#

```

## Workflow RBAC

All pods in a workflow run with the service account specified in `workflow.spec.serviceAccountName`, or if omitted, the default service account of the workflow's namespace. The amount of access which a workflow needs is dependent on what the workflow needs to do. For example, if your workflow needs to deploy a resource, then the workflow's service account will require 'create' privileges on that resource.

**Warning:** We do not recommend using the default service account in production. It is a shared account so may have permissions added to it you do not want. Instead, create a service account only for your workflow.

The minimum for the executor to function: (named: `rbac-executor.yaml`)

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: executor
rules:
  - apiGroups:

```

```
- argoproj.io
resources:
- workflowtaskresults
verbs:
- create
- patch
```

```
root@devvm:/home/amit/argowf-setup# kubectl create -f rbac-executor.yaml
role.rbac.authorization.k8s.io/executor created
```

## Workflow Controller Security

This has three parts.

### Controller Permissions

The controller has permission (via Kubernetes RBAC + its config map) with either all namespaces (cluster-scope install) or a single [managed namespace](#) (namespace-install), notably:

- List/get/update workflows, and cron-workflows.
- Create/get/delete pods, PVCs, and PDBs.
- List/get template, config maps, service accounts, and secrets.

```
-----
root@devvm:/home/amit/argowf-setup# kubectl get clusterrole |grep argo
argo-aggregate-to-admin      2024-05-01T09:43:24Z
argo-aggregate-to-edit      2024-05-01T09:43:24Z
argo-aggregate-to-view      2024-05-01T09:43:24Z
argo-cluster-role            2024-05-01T09:43:24Z
argo-server-cluster-role     2024-05-01T09:43:24Z
-----
```

### User Permissions

Users minimally need permission to create/read workflows. The controller will then create workflow pods (config maps etc) on behalf of the users, even if the user does not have permission to do this themselves. The controller will only create workflow pods in the workflow's namespace.

A way to think of this is that, if the user has permission to create a workflow in a namespace, then it is OK to create pods or anything else for them in that namespace.

If the user only has permission to create workflows, then they will be typically unable to configure other necessary resources such as config maps, or view the outcome of their workflow. This is useful when the user is a service.

## UI Access

If you want a user to have read-only access to the entirety of the Argo UI for their namespace, a sample role for them may look like:

<https://github.com/amitopenwriteup/argo-setup.git>

## Argo Server

The Argo Server is commonly exposed to end-users to provide users with a UI for visualizing and managing their workflows. It must also be exposed if leveraging [webhooks](#) to trigger workflows. Both of these use cases require that the argo-server Service to be exposed for ingress traffic (e.g. with an Ingress object or load balancer). Note that the Argo UI is also available to be accessed by running the server locally (i.e. argo server) using local KUBECONFIG credentials, and visiting the UI over <https://localhost:2746>.

We will nodeport to do the port-forwarding

```
apiVersion: v1
kind: Service
metadata:
  name: argo-server-nodeport
  namespace: argo
spec:
  type: NodePort
  selector:
    app: argo-server
  ports:
    - protocol: TCP
      port: 2746
      targetPort: 2746
      nodePort: 30000 # Specify the nodePort value you want to use
```

```

root@devvm: /home/amit/argowf# kubectl get svc -n argo
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
argo-server         ClusterIP   10.108.6.173   <none>         2746/TCP         22m
argo-server-nodeport NodePort     10.110.28.69   <none>         2746:30000/TCP   6s

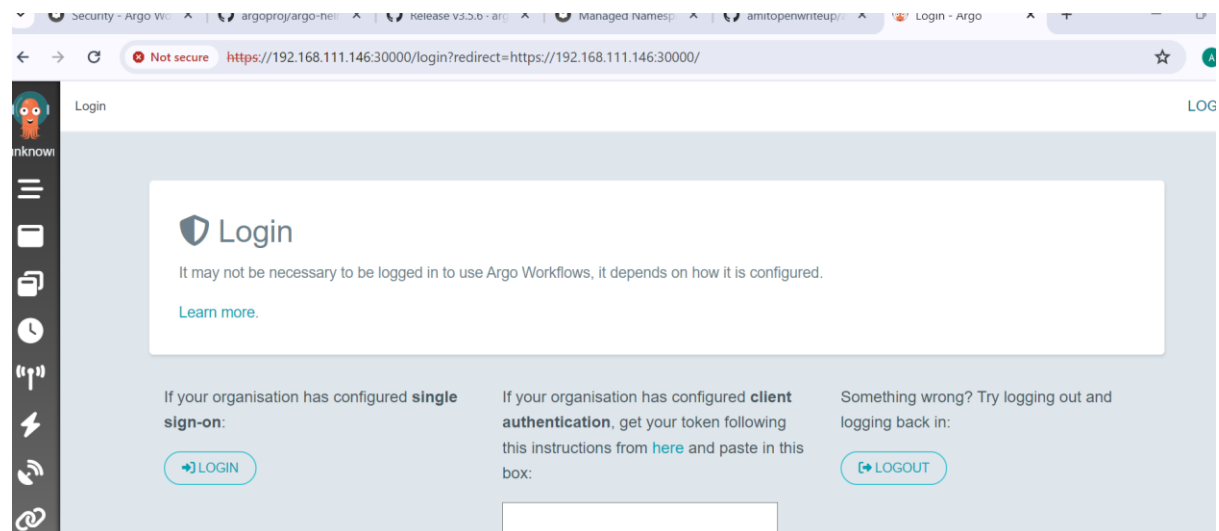
```

## Access through UI

server - in hosted mode, use the kube config of service account, in local mode, use your local kube config.

client - requires clients to provide their Kubernetes bearer token and use that.

sso - since v2.9, use single sign-on, this will use the same service account as per "server" for RBAC. We expect to change this in the future so that the OAuth claims are mapped to service accounts.



For Access token way:

- Create a service account training  
`kubectl create sa training -n argo`
- For training purpose, we are providing full access to service account  
`kubectl create clusterrolebinding training-rb--clusterrole=argo-server-cluster-role--serviceaccount=argo:training`  
`kubectl create clusterrolebinding training-rb2--clusterrole=cluster-admin--serviceaccount=argo:training`
- Generate Auth token

Create secret.yaml

```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: training.service-account-token
  namespace: argo
  annotations:
    kubernetes.io/service-account.name: training
  type: kubernetes.io/service-account-token
```

```
kubectl create-f secret.yaml
```

```
ARGO_TOKEN="Bearer $(kubectl get secret training.service-account-token-n argo-  
o=jsonpath='{.data.token}' | base64--decode)"
```

```
echo $ARGO_TOKEN
```

Copy and paste the token in the below window

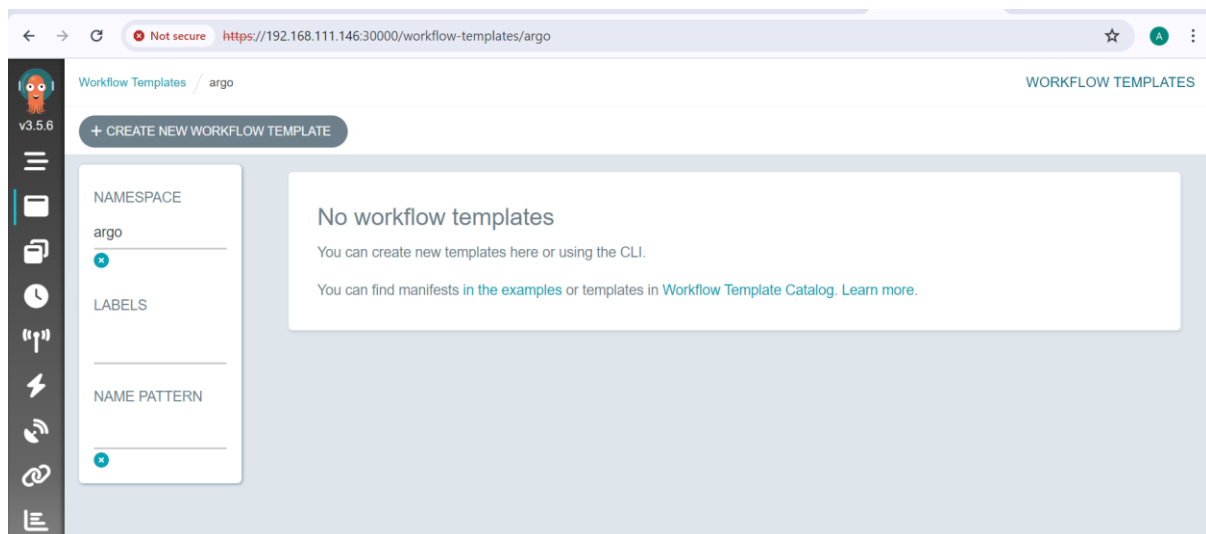
ion has configured **single**

If your organisation has configured **client authentication**, get your token following this instructions from [here](#) and paste in this box:

```
NoNGcHL31av45pMz0lyLbO6IO4iy  
NEK-  
ItWRKHk9QgUxdIPZ3fD6TBMFwnq  
DD_rbLcH8X7NUYixNmO4Biz35A  
GVYyf6aWkCjQSQ3EnX9aoR4PAA  
c8aNK-6r-  
iLBTdEPgPg3oKlzp7YSTBy6f_sam  
Y4OtlSY0Oe9zwEo_0OePfQ
```

Successfully logged-in





## The Workflow

### Definition

The **Workflow** is the most important resource in Argo and serves two important functions:

1. It defines the workflow to be executed.
2. It stores the state of the workflow.

Imagine a Workflow in Argo like a recipe for baking a cake. It's not just a list of ingredients and steps; it's also where you keep track of how far along you are in making the cake.

**Defining the Workflow:** First, the Workflow tells you what you need to do to make the cake. This includes listing the ingredients (like flour, eggs, sugar) and the steps to follow (like mixing, baking).

**Storing the State:** Second, as you follow the recipe and make the cake, the Workflow keeps track of where you are in the process. It remembers if you've mixed the ingredients, put the cake in the oven, or if it's done baking.

So, the Workflow isn't just a set of instructions; it's also like a diary that keeps updating as you make progress. This dual role helps you know what to do next and where you are in completing your task, whether it's baking a cake or running a complex process with Argo.

### Workflow Spec

**Workflow.spec Field:** This is like the blueprint or plan for what the Workflow will do. It's where you list out all the things the Workflow needs to do.

**Templates:** Think of these as different jobs or tasks that the Workflow needs to perform. Each template is like a set of instructions for one specific task.

**Entrypoint:** This is like pointing to the main task or job that the Workflow should start with. It's like saying, "Hey, start with this job first!"

So, in simple terms, the Workflow.spec field is where you outline what needs to be done (like a to-do list), templates are the individual tasks on that list, and the entrypoint is the first task that gets done.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: wonderful-poochenheimer
  namespace: argo
  labels:
    example: 'true'
spec:
  serviceAccountName: training # Added service account
  arguments:
    parameters:
      - name: message
        value: hello argo
  entrypoint: argosay
  templates:
    - name: argosay
      inputs:
        parameters:
          - name: message
            value: '{{workflow.parameters.message}}'
      container:
        name: main
        image: argoproj/argosay:v2
```

```
command:
  - /argosay

args:
  - e\cho
  - '{{inputs.parameters.message}}'

ttlStrategy:
  secondsAfterCompletion: 300

podGC:
  strategy: OnPodCompletion
```

## Template

In Argo Workflows, a template defines a specific task to be completed, often involving running a container. Let's break down the **CONTAINER** template type using an example:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: cowsay-example
  namespace: argo
spec:
  serviceAccountName: training
  entrypoint: cowsay-entrypoint
  templates:
    - name: cowsay-entrypoint
      container:
        image: docker/whalesay
        command: [cowsay]
        args: ["hello from Argo!"]
```

```
ttlStrategy:
  secondsAfterCompletion: 300
podGC:
  strategy: OnPodCompletion
```

- The workflow is named cowsay-example and belongs to the argo namespace.
- It uses the training service account (serviceAccountName: training).
- The entrypoint is defined as cowsay-entrypoint, which is where the cowsay command is executed.
- The cowsay-entrypoint template runs a container using the docker/whalesay image and executes the cowsay command with the argument "hello from Argo!".
- The ttlStrategy specifies that completed pods will be removed after 300 seconds (5 minutes).
- The podGC strategy is set to OnPodCompletion, meaning pods will be removed after they complete.

## Script type

A "script" in an Argo Workflow is like a handy tool that wraps around a container. It's a way to run a script inside a container without having to manage the script file separately. Here's a simple breakdown:

**Convenience Wrapper:** Think of it as a convenient way to execute a script inside a container without worrying about creating or managing separate script files.

**Spec Same as Container:** The specifications (spec) for a script in Argo Workflow are similar to running a container. You still specify the image, command, and arguments just like you do when running a container directly.

**Source Field:** This is where you define the script that you want to run. Instead of creating a script file and copying it into the container, you write the script directly in the YAML file using the `source:` field.

**Execution:** When the workflow runs, Argo will save the script into a file inside the container and execute it. You don't need to handle the script file creation or deletion; Argo manages that for you.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
```

```
name: gen-random-int
namespace: argo

spec:
  serviceAccountName: training
  entrypoint: gen-random-int-entrypoint
  templates:
    - name: gen-random-int-entrypoint
      script:
        image: python:alpine3.6
        command: [python]
        source: |
          import random

          i = random.randint(1, 100)

          print(i)
  ttlStrategy:
    secondsAfterCompletion: 300
  podGC:
    strategy: OnPodCompletion
```

- The workflow is named gen-random-int and belongs to the argo namespace.
- It uses the training service account (serviceAccountName: training).
- The entrypoint is defined as gen-random-int-entrypoint, where the random integer generation script is executed.
- The gen-random-int-entrypoint template runs a container using the python:alpine3.6 image and executes the Python script to generate a random integer between 1 and 100.
- The ttlStrategy specifies that completed pods will be removed after 300 seconds (5 minutes).
- The podGC strategy is set to OnPodCompletion, meaning pods will be removed after they complete.

## Resource Template

"Performing operations on cluster resources directly" means you can use Argo Workflows to

do things like create, modify, or delete resources on your Kubernetes cluster without manually interacting with the cluster. Here's a simple breakdown:

**Get:** You can retrieve information about existing resources.

**Create:** You can make new resources like Pods, ConfigMaps, or Services.

**Apply:** This updates resources if they exist or creates them if they don't.

**Delete:** You can remove resources from the cluster.

**Replace:** This deletes and recreates resources, ensuring they are up-to-date.

**Patch:** You can make changes to existing resources without recreating them entirely.

For example, let's say you want to create a ConfigMap resource using Argo. You'd provide a YAML description of the ConfigMap, specifying its name, data, and any other required details. Argo would then handle creating this ConfigMap on your Kubernetes cluster automatically when the workflow runs.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: k8s-owner-reference-training
  namespace: argo
spec:
  serviceAccountName: training
  entrypoint: k8s-owner-reference
  templates:
    - name: k8s-owner-reference
      resource:
        action: create
        manifest: |
          apiVersion: v1
          kind: ConfigMap
          metadata:
            generateName: owned-eg-
          data:
```

```

    some: value

ttlStrategy:
  secondsAfterCompletion: 300

podGC:
  strategy: OnPodCompletion

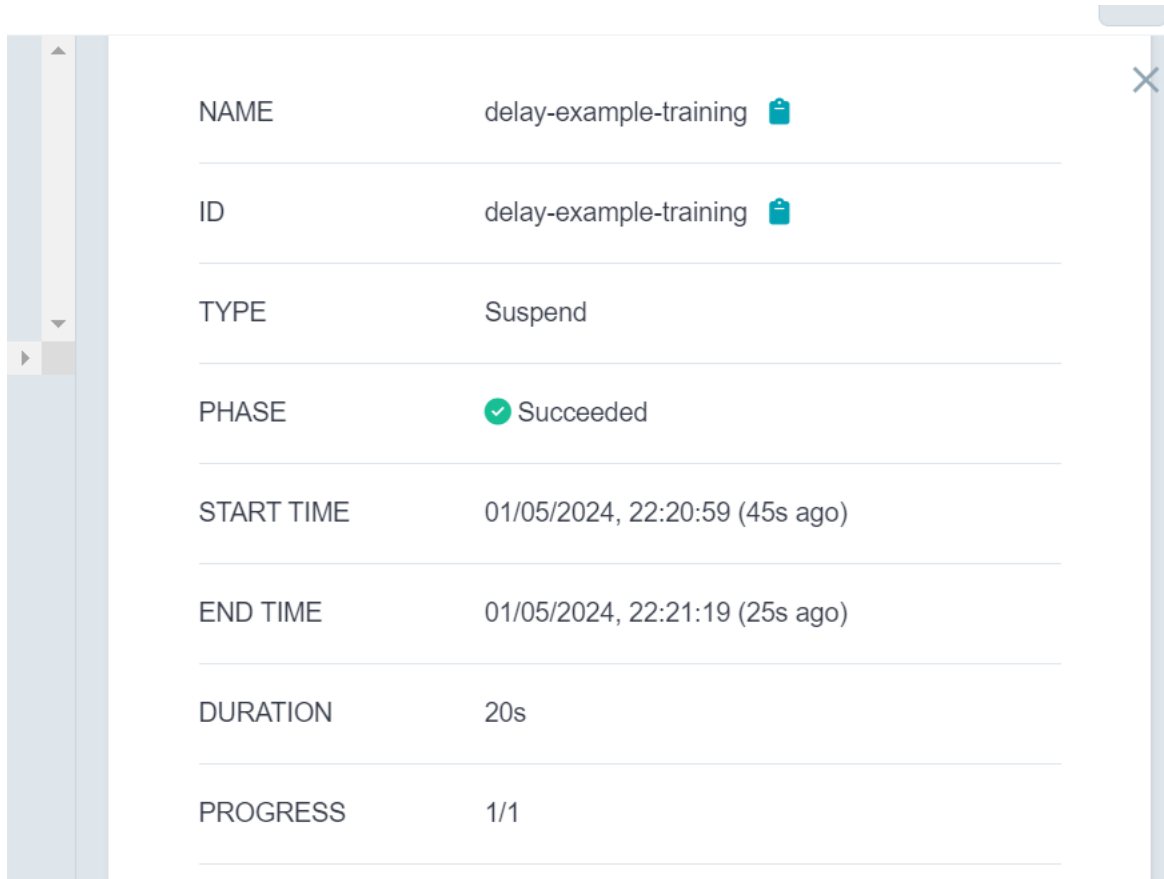
```




TYPE	LAST SEEN	REASON	OBJECT	MESSAGE
✓	1m32s ago	Pulled	Pod/k8s-owner-ref...	Container image "quay.io/argoproj/argoexec:v3.5.6" ahead...
✓	1m32s ago	Created	Pod/k8s-owner-ref...	Created container main
✓	1m32s ago	Started	Pod/k8s-owner-ref...	Started container main
✓	1m34s ago	Scheduled	Pod/k8s-owner-ref...	Successfully assigned argo/k8s-owner-reference-training t...
✓	1m34s ago	Pulled	Pod/k8s-owner-ref...	Container image "quay.io/argoproj/argoexec:v3.5.6" ahead...
✓	1m34s ago	Created	Pod/k8s-owner-ref...	Created container init
✓	1m34s ago	Started	Pod/k8s-owner-ref...	Started container init

- The workflow is named **k8s-owner-reference** and belongs to the **argo** namespace.
- The entrypoint is set to **k8s-owner-reference**, which refers to the template that creates the ConfigMap.
- There is one template defined (**k8s-owner-reference**) that uses the **resource** type to create a resource on the cluster.
- The **action** field is set to **create** to indicate that this template will create a resource.
- The **manifest** field contains the YAML description of the resource to be created. In this case, it's a ConfigMap with a generated name (**generateName**) starting with "owned-eg-" and containing a key-value pair (**some: value**).
- The **ttlStrategy** specifies that completed pods will be removed after 300 seconds (5 minutes).
- The **podGC** strategy is set to **OnPodCompletion**, meaning pods will be removed after they complete.

## Suspend Template

A suspend template in Argo Workflow is like a pause button that temporarily stops the workflow execution. It can be set to suspend for a specific duration or until it's manually resumed.



NAME	delay-example-training 
ID	delay-example-training 
TYPE	Suspend
PHASE	 Succeeded
START TIME	01/05/2024, 22:20:59 (45s ago)
END TIME	01/05/2024, 22:21:19 (25s ago)
DURATION	20s
PROGRESS	1/1

```
apiVersion: argoproj.io/v1alpha1
```

```
kind: Workflow
```

```
metadata:
```

```
  name: delay-example-training
```

```
  namespace: argo
```

```
spec:
```

```
  serviceAccountName: training
```

```
  entrypoint: delay
```

```
  templates:
```



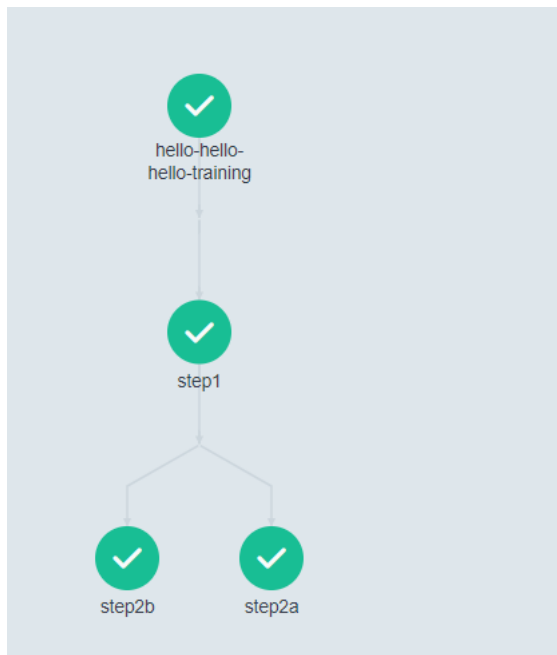
```
- name: delay
  suspend:
    duration: "20s"
ttlStrategy:
  secondsAfterCompletion: 300
podGC:
  strategy: OnPodCompletion
```

- The workflow is named `delay-example` and belongs to the `argo` namespace.
- The entrypoint is set to `delay`, which refers to the suspend template named `delay`.
- There is one template defined (`delay`) that uses the `suspend` type to pause the workflow execution for 20 seconds.
- The `duration` field is set to `"20s"` to specify the duration of the suspension.
- The `ttlStrategy` specifies that completed pods will be removed after 300 seconds (5 minutes).
- The `podGC` strategy is set to `OnPodCompletion`, meaning pods will be removed after they complete.

### Step Template

A "steps" template in Argo Workflow is like a series of instructions where you can define tasks to be performed in a specific order.

```
- name: hello-hello-hello
  steps:
    - name: step1
      template: prepare-data
    - name: step2a
      template: run-data-first-half
    - name: step2b
      template: run-data-second-half
```



```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: hello-hello-hello-training
  namespace: argo
spec:
  serviceAccountName: training
  entrypoint: hello-hello-hello
  templates:
    - name: hello-hello-hello
      steps:
        - name: step1
          template: prepare-data
        - name: step2a
          template: run-data-first-half
        - name: step2b
          template: run-data-second-half
```

```
- name: prepare-data
  container:
    image: busybox
    command: [sh, -c]
    args: ["echo 'Preparing data...'"]
- name: run-data-first-half
  container:
    image: busybox
    command: [sh, -c]
    args: ["echo 'Running first half of data...'"]
- name: run-data-second-half
  container:
    image: busybox
    command: [sh, -c]
    args: ["echo 'Running second half of data...'"]
ttlStrategy:
  secondsAfterCompletion: 300
podGC:
  strategy: OnPodCompletion
```

- The workflow uses the `training` service account (`serviceAccountName: training`) to execute the workflow tasks.
- The steps (`step1`, `step2a`, `step2b`) run containers with different commands to echo messages, simulating different tasks.
- The `entrypoint` is set to `hello-hello-hello`, which refers to the steps template with the same name.
- The `ttlStrategy` specifies that completed pods will be removed after 300 seconds (5 minutes).
- The `podGC` strategy is set to `OnPodCompletion`, meaning pods will be removed after they complete.

## Template DAG

```
- name: diamond
  dag:
    tasks:
      - name: A
        template: echo
      - name: B
        dependencies: [A]
        template: echo
      - name: C
        dependencies: [A]
        template: echo
      - name: D
        dependencies: [B, C]
        template: echo
```

- In a DAG template, tasks are organized based on dependencies. Here's a simple explanation using your example:
- **Name:** This is the name of the DAG template, which you can use to refer to it elsewhere in your workflow.
- **Tasks List:** In the tasks list, you define each task along with its dependencies and the template it uses.
- **Task A:** This is the first task and doesn't have any dependencies (**dependencies: []**). It runs the **echo** template.
- **Task B and Task C:** Both tasks **B** and **C** have a dependency on Task **A** (**dependencies: [A]**). This means they will wait for Task **A** to complete before they start. They run in parallel.
- **Task D:** Task **D** has dependencies on both Task **B** and Task **C** (**dependencies: [B, C]**). This means it will wait for both **B** and **C** to complete before it starts. Task **D** runs after **B** and **C** are completed.



```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: diamond-example-training
  namespace: argo
spec:
  serviceAccountName: training
  entrypoint: diamond
  templates:
    - name: diamond
      dag:
        tasks:
          - name: A
            template: echo
          - name: B
            dependencies: [A]
            template: echo
```

```

- name: C
  dependencies: [A]
  template: echo

- name: D
  dependencies: [B, C]
  template: echo

- name: echo
  container:
    image: busybox
    command: [echo, "Task executed."]

ttlStrategy:
  secondsAfterCompletion: 300

podGC:
  strategy: OnPodCompletion

```

- The workflow uses the **training** service account (**serviceAccountName: training**) to execute the workflow tasks.
- The entrypoint is set to **diamond**, which refers to the DAG template with the same name.
- The **diamond** DAG template defines four tasks (**A**, **B**, **C**, **D**) with their dependencies and the template they use (**echo**).
- Task **A** runs the **echo** template immediately because it has no dependencies (**dependencies: []**).
- Tasks **B** and **C** wait for Task **A** to complete (**dependencies: [A]**) before running. They run in parallel.
- Task **D** waits for both **B** and **C** to complete (**dependencies: [B, C]**) before running.
- The **echo** template is a simple container that echoes "Task executed."
- The **ttlStrategy** specifies that completed pods will be removed after 300 seconds (5 minutes).
- The **podGC** strategy is set to **OnPodCompletion**, meaning pods will be removed after they complete.

### WorkflowTemplates

ClusterWorkflowTemplates are like master blueprints for workflows that you can use across the entire cluster, similar to how a master key works for all the rooms in a building. Once created at the cluster level, they can be used in any namespace within the cluster, making them accessible from anywhere in your system.

WorkflowTemplates are like ready-made instructions for tasks in your cluster. They act as templates that you can reuse whenever you need to perform certain tasks. You can think of them as pre-designed plans that you can quickly use to carry out specific actions within your system.

### Template vs WorkflowTemplate

Aspect	WorkflowTemplate	template (lowercase)
Definition	Predefined blueprint for entire workflows	Specific task instruction within a Workflow or WorkflowTemplate
Usage	Used to create reusable workflow plans	Used to define individual tasks within workflows
Scope	Cluster-wide, accessible across namespaces	Scoped within a specific Workflow or WorkflowTemplate
Complexity	Represents a collection of tasks and workflows	Represents a single task or step within a workflow
Example	A set of instructions for deploying an application	A step-by-step guide for configuring a database

```

apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: steps-
spec:
  entrypoint: hello      # We reference our first "template" here
  templates:
    - name: hello        # The first "template" in this Workflow, it is referenced by "entrypoint"
      steps:             # The type of this "template" is "steps"
        -- name: hello
          template: whalesay # We reference our second "template" here

```

```

arguments:
  parameters: [{name: message, value: "hello1"}]
- name: whalesay      # The second "template" in this Workflow, it is referenced by "hello"
inputs:
  parameters:
    - name: message
container:      # The type of this "template" is "container"
  image: docker/whalesay
  command: [cowsay]
  args: ["{{inputs.parameters.message}}"]

```

This workflow has two templates:

#### hello Template:

- Defines the "hello" template that references the "whalesay" template.
- Type: "steps" template.
- Contains a step named "hello" that references the "whalesay" template.
- Passes the "message" parameter with the value "hello1" to the "whalesay" template.

#### whalesay Template:

- Defines the "whalesay" template.
- Type: "container" template, indicating it runs a containerized task.
- Specifies input parameters, in this case, a "message" parameter.
- Runs the "cowsay" command inside a Docker container using the "docker/whalesay" image, with the "message" parameter as an argument.

### WorkflowTemplate Spec

A WorkflowTemplate is like a master plan for workflows in your cluster. It includes templates, which are like pre-made task instructions. These templates can be used within the WorkflowTemplate itself and can also be shared and used by other workflows and WorkflowTemplates in your cluster. So, it's like having a blueprint for how workflows should be structured, with reusable instructions that can be used across different parts of your system.



In versions 2.7 and later, WorkflowTemplates support all fields in WorkflowSpec except for the "priority" field. This means that you can use WorkflowTemplates to define workflows with a wide range of configurations, making them more versatile and powerful for managing your workflows in Argo Workflows.

In versions 2.4 to 2.6, WorkflowTemplates were limited to containing only the "templates" and "arguments" fields. They couldn't include other fields like "entrypoint" that are found in a full Workflow definition.

So, if you tried to use a WorkflowTemplate in versions 2.4 to 2.6 that included an "entrypoint" field, it wouldn't be considered a valid WorkflowTemplate because those versions didn't support the "entrypoint" field in WorkflowTemplates.

#### Valid WorkflowTemplates (v 2.7)

```
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate
metadata:
  name: workflow-template-submittable
spec:
  entrypoint: whalesay-template # Fields other than "arguments" and "templates" not
                                supported in v2.4- v2.6
  arguments:
    parameters:
      - name: message
        value: hello world
  templates:
    - name: whalesay-template
      inputs:
        parameters:
          - name: message
      container:
        image: docker/whalesay
        command: [cowsay]
        args: ["{{inputs.parameters.message}}"]
```

Valid for (v 2.4-2.6)

```
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate
metadata:
  name: workflow-template-submittable
spec:
  arguments:
    parameters:
      - name: message
        value: hello world
  templates:
    - name: whalesay-template
      inputs:
        parameters:
          - name: message
      container:
        image: docker/whalesay
        command: [cowsay]
        args: ["{{inputs.parameters.message}}"]
```

#### Adding labels/annotations to Workflows with workflowMetadata (v 2.10 and after)

The workflowMetadata field in WorkflowTemplates allows you to automatically add labels and/or annotations to Workflows that are created from those templates. Here's how it works:

**Labels:** You can specify labels under workflowMetadata to add metadata tags to Workflows. For example, if you have labels like app: my-app and environment: production, specifying them under workflowMetadata in a WorkflowTemplate will automatically apply these labels to all Workflows created from that template.

**Annotations:** Similarly, you can add annotations under workflowMetadata to provide additional information or context to Workflows. Annotations are key-value pairs that can be used for various purposes like tracking, debugging, or monitoring.

```
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate
metadata:
  name: workflow-template-submittable
spec:
  workflowMetadata:
    labels:
      example-label: example-value
  templates:
    - name: hello-world
      inputs:
        parameters:
          - name: message
            default: "Hello, World!"
      steps:
        -- name: print-message
          template: whalesay
          arguments:
            parameters:
              - name: message
                value: "{{inputs.parameters.message}}"
        - name: whalesay
          container:
            image: docker/whalesay
            command: [cowsay]
            args: ["{{inputs.parameters.message}}"]
```

## Working with parameters

Suppose you have a global parameter named message that you want to use in multiple Workflows created from your WorkflowTemplate. Here's how you can set it up:

<https://argo-workflows.readthedocs.io/en/latest/workflow-templates/>

### Cluster Workflow Template

ClusterWorkflowTemplates are cluster scoped WorkflowTemplates. ClusterWorkflowTemplate can be created cluster scoped like ClusterRole and can be accessed across all namespaces in the cluster.

```
apiVersion: argoproj.io/v1alpha1
kind: ClusterWorkflowTemplate
metadata:
  name: cluster-workflow-template-whalesay-template
spec:
  templates:
  - name: whalesay-template
    inputs:
      parameters:
      - name: message
    container:
      image: docker/whalesay
      command: [cowsay]
      args: ["{{inputs.parameters.message}}"]
```

You all will submit this and observe error message :

Error (exit code 1): pods "cluster-workflow-template-whalesay-template-t5tpc" is forbidden: User "system:serviceaccount:argo:default" cannot patch resource "pods" in API group "" in the namespace "argo"

Why we got this error message: Check the SA Roles.

### Referencing other ClusterWorkflowTemplates

Sure, referencing templates from other `ClusterWorkflowTemplates` using `templateRef` with `clusterScope: true` is a powerful feature in Argo Workflows. Here's a simple explanation of how this works:

### 1. **ClusterWorkflowTemplates and TemplateRef:**

- A `ClusterWorkflowTemplate` is a resource in Argo Workflows that defines reusable workflow templates at the cluster level.
- You can reference templates defined in other `ClusterWorkflowTemplates` using the `templateRef` field.
- When referencing a template from another `ClusterWorkflowTemplate`, you specify the name of the template and set `clusterScope: true` to indicate that the reference is at the cluster level.

### 2. **Steps or DAG Templates:**

- To use a referenced template within a workflow, you typically define a steps or DAG template.
- In the steps or DAG template, you specify the `templateRef` with the name of the referenced template and set `clusterScope: true`.

### 3. **Example:**

Let's say you have two `ClusterWorkflowTemplates`: `templateA` and `templateB`.

- In `templateA`, you define a template named `stepA`:

```
apiVersion: argoproj.io/v1alpha1
kind: ClusterWorkflowTemplate
metadata:
  name: templateA
spec:
  templates:
    - name: stepA
      container:
        image: busybox
        command: [echo, "Hello from stepA"]
```

- In `templateB`, you want to use `stepA` from `templateA`. You can do this in a steps template:

```
apiVersion: argoproj.io/v1alpha1
kind: ClusterWorkflowTemplate
metadata:
  name: templateB
spec:
  templates:
    - name: steps-template
      steps:
        -- name: use-stepA
          templateRef:
            name: templateA.stepA
            clusterScope: true
```

In this example, `templateB` references `stepA` from `templateA` using `templateRef` with `clusterScope: true`. This allows you to reuse and compose workflows across multiple `ClusterWorkflowTemplates` within your Argo Workflows environment.

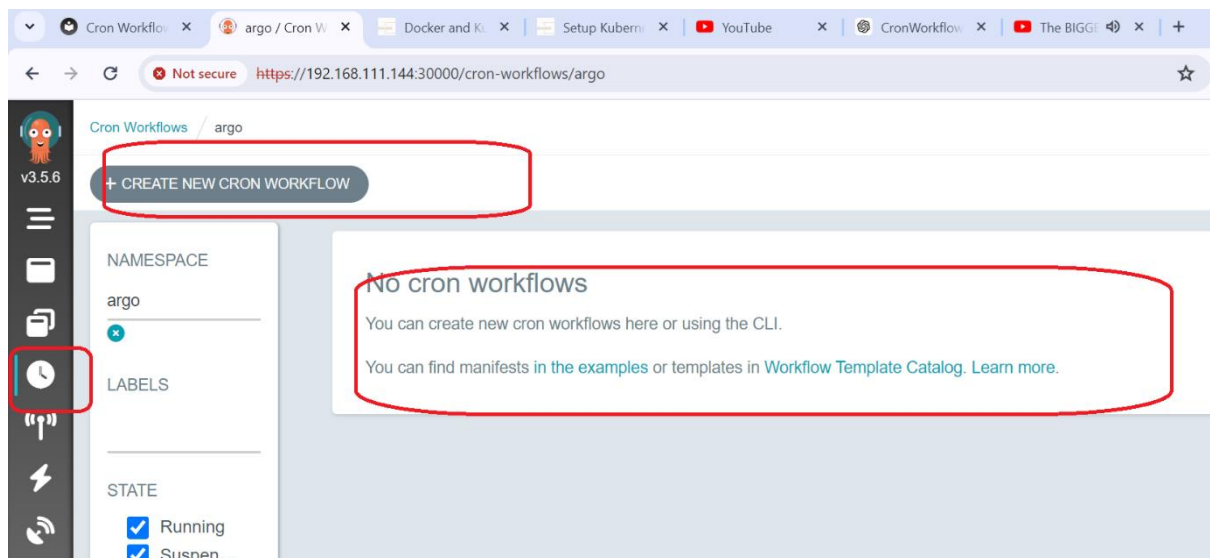
### CronWorkflows

CronJobs: Think of these like timers. They tell the computer to do something at certain times or on a regular schedule, like running a program every hour or every day.

CronWorkflows: Now, imagine combining a workflow (the sequence of tasks) with a timer (like CronJobs). This creates a CronWorkflow, which is a set of tasks that automatically runs based on a schedule you set. It's like having a recipe that the computer follows at specific times without you having to start it manually each time.

```
apiVersion: argoproj.io/v1alpha1
kind: CronWorkflow
metadata:
  name: test-cron-wf
spec:
  schedule: "* * * * *
```

```
concurrencyPolicy: "Replace"
startingDeadlineSeconds: 0
workflowSpec:
  entrypoint: whalesay
  serviceAccountName: training
  shutdown: Terminate
  templates:
    - name: whalesay
      container:
        image: alpine:3.6
        command: ["sh", "-c"]
        args: ["date; sleep 90"]
```



### Timezone parameter

timezone: Asia/Kolkata sets the timezone for the CronWorkflow to India Standard Time (IST), which is represented as "Asia/Kolkata" in the IANA timezone database.

```
apiVersion: argoproj.io/v1alpha1
kind: CronWorkflow
```

```
metadata:
  name: test-cron-wf
spec:
  schedule: "* * * * *"
  concurrencyPolicy: "Replace"
  startingDeadlineSeconds: 0
  workflowSpec:
    entrypoint: whalesay
    serviceAccountName: training
    shutdown: Terminate
    timezone: Asia/Kolkata
    templates:
      - name: whalesay
        container:
          image: alpine:3.6
          command: ["sh", "-c"]
```

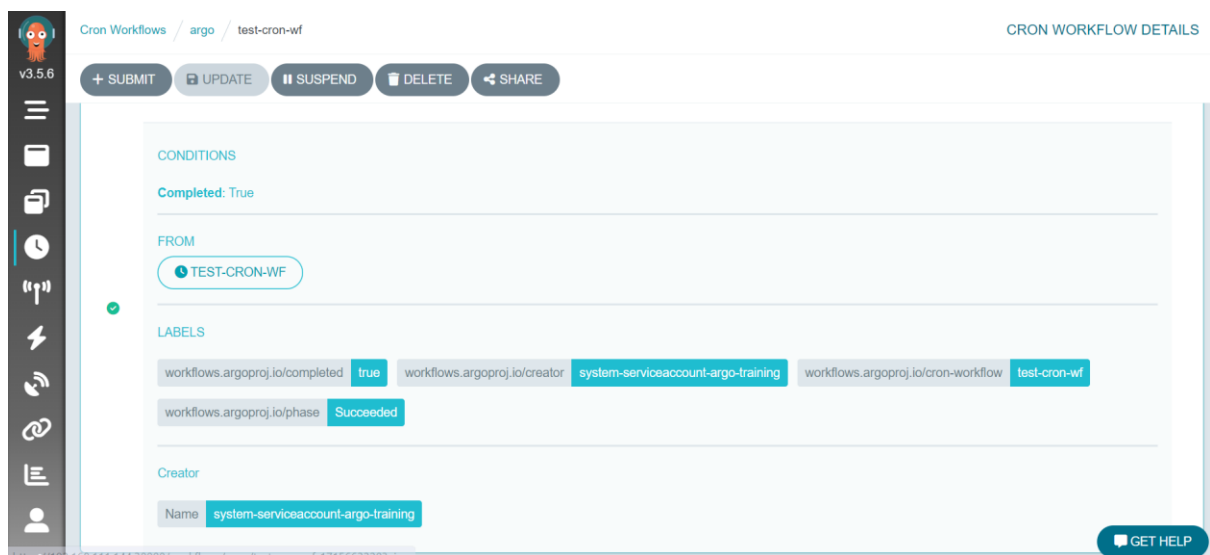
#### Add other parameters in cronjob workflow

- suspend: false specifies that the workflow scheduling will not be suspended.
- concurrencyPolicy: Allow sets the policy to allow multiple workflows to run at the same time.
- startingDeadlineSeconds: 0 indicates that a missed workflow will be run immediately after the last successful run.
- successfulJobsHistoryLimit: 3 sets the number of successful workflows to be persisted.
- failedJobsHistoryLimit: 1 sets the number of failed workflows to be persisted.

```
apiVersion: argoproj.io/v1alpha1
kind: CronWorkflow
metadata:
  name: test-cron-wf-1
spec:
  schedule: "* * * * *"
```



```
suspend: false
concurrencyPolicy: Allow
startingDeadlineSeconds: 0
successfulJobsHistoryLimit: 3
failedJobsHistoryLimit: 1
workflowSpec:
  entrypoint: whalesay
  serviceAccountName: training
  shutdown: Terminate
  timezone: Asia/Kolkata
  templates:
  - name: whalesay
    container:
      image: alpine:3.6
      command: ["sh", "-c"]
      args: ["date; sleep 90"]
```



	NAME	NAMESPACE	STARTED	FINISHED	DURATION	PROGRESS	MESSAGE	DETAILS	ARCHIVED
✓	test-cron-wf-1715663280	argo	8s ago	8s ago	0s	1/1	-	SHOW ▾	false
✓	test-cron-wf-1715663220	argo	1m8s ago	1m8s ago	0s	1/1	-	SHOW ▾	false

[GET HELP](#)

Imagine you have a CronWorkflow that's supposed to run every minute, starting at 12:00 PM. However, if the controller managing this workflow crashes at some point, there's a chance that the scheduled run could be missed. Here's where the `startingDeadlineSeconds` parameter comes into play.

- `StartingDeadlineSeconds`: This parameter sets a time limit after which a missed run of the CronWorkflow will still be executed. For example, if you set `startingDeadlineSeconds` to 65 seconds and the controller crashes between 12:05:55 and 12:06:05 (65 seconds after the last scheduled run at 12:05:00), the missed run at 12:06:00 will still be executed when the controller restarts at 12:06:05.

In simpler terms, `startingDeadlineSeconds` ensures that even if the controller crashes and misses a scheduled run, it will catch up and execute the missed run within the specified time limit after the last successful run.

### Command line options for cronworkflow

```
argo cron create argo-cronjob.yaml
argo cron list -n argo
argo cron get test-cron-wf-1 -n argo
argo cron suspend test-cron-wf-1 -n argo
argo cron list -n argo
```

yaml file

```
apiVersion: argoproj.io/v1alpha1
kind: CronWorkflow
metadata:
  name: test-cron-wf-1
  namespace: argo
spec:
```

```
schedule: "* * * * *"
suspend: false
concurrencyPolicy: Allow
startingDeadlineSeconds: 0
successfulJobsHistoryLimit: 3
failedJobsHistoryLimit: 1
workflowSpec:
  entrypoint: whalesay
  serviceAccountName: training
  shutdown: Terminate
  templates:
    - name: whalesay
      container:
        image: alpine:3.6
        command: ["sh", "-c"]
        args: ["date; sleep 90"]
```

## Template Types

An HTTP Template in Argo Workflows is a type of template that allows you to execute HTTP requests as part of your workflow. Here's a detail about how HTTP Templates work and what they're used for:

1. **HTTP Request Execution:** An HTTP Template is designed to interact with external services or APIs by sending HTTP requests. This can include GET, POST, PUT, DELETE, or any other HTTP method supported by the service you're communicating with.
2. **Usage in Workflows:** You can use an HTTP Template within your workflow to perform actions such as:
  - Retrieving data from a REST API.
  - Sending data to a remote server or service.
  - Triggering actions on external systems.

- Performing CRUD (Create, Read, Update, Delete) operations on resources.
3. **Configuration:** When using an HTTP Template, you typically specify the HTTP method (GET, POST, etc.), the URL endpoint to which the request will be sent, any headers or parameters required by the API, and the payload (if applicable) to be sent with the request.
  4. **Response Handling:** After sending the HTTP request, the HTTP Template can handle the response from the server. This may involve parsing the response body, extracting relevant data, and making decisions or performing further actions based on the response.
  5. **Integration:** HTTP Templates are often used to integrate workflows with external systems, cloud services, or APIs. For example, you might use an HTTP Template to fetch data from a database, trigger a deployment on a cloud platform, or interact with a third-party service for authentication or data processing.

## Argo Agent

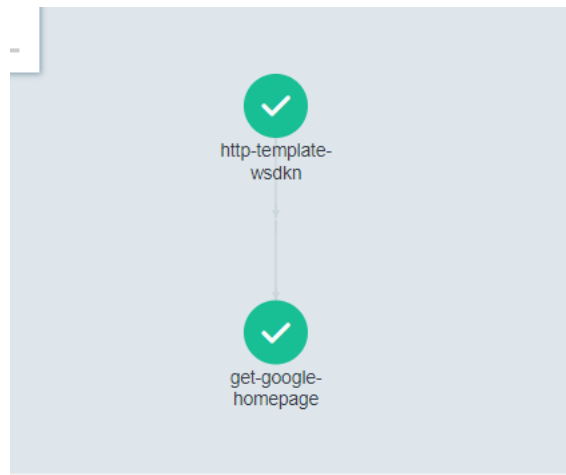
HTTP Templates in Argo Workflows use the Argo Agent to execute HTTP requests. The Argo Agent works independently of the main controller and handles these requests.

When a Workflow needs the Argo Agent, a WorkflowTaskSet Custom Resource Definition (CRD) is created. This CRD helps the Agent and the Workflow Controller communicate effectively.

To Use the Argo Agent, must setup RBAC in Arog workflow. This involves adding a role for the agent in RBAC configuration

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: http-template-
  namespace: argo
spec:
  entrypoint: main
  serviceAccountName: training # Add the service account here for triggering the Workflow
  templates:
    - name: main
```

```
steps:
  -- name: get-google-homepage
    template: http
    arguments:
      parameters: [{name: url, value: "https://www.google.com"}]
- name: http
  inputs:
    parameters:
      - name: url
  http:
    timeoutSeconds: 20 # Default 30
    url: "{{inputs.parameters.url}}"
    method: "GET" # Default GET
    headers:
      - name: "x-header-name"
        value: "test-value"
    # Template will succeed if evaluated to true, otherwise will fail
    # Available variables:
    # request.body: string, the request body
    # request.headers: map[string][]string, the request headers
    # response.url: string, the request url
    # response.method: string, the request method
    # response.statusCode: int, the response status code
    # response.body: string, the response body
    # response.headers: map[string][]string, the response headers
    successCondition: "response.body contains \"google\"" # available since v3.3
    body: "test body" # Change request body
```



Examples:

```
# Submit multiple workflows from files:
argo submit my-wf.yaml

# Submit and wait for completion:
argo submit--wait my-wf.yaml

# Submit and watch until completion:
argo submit--watch my-wf.yaml

# Submit and tail logs until completion:
argo submit--log my-wf.yaml

# Submit a single workflow from an existing resource
argo submit--from cronwf/my-cron-wf
```

### Container Set Template

A container set template in Argo Workflows lets you run multiple containers together inside a single pod. This means all these containers will run on the same server (or node) in your Kubernetes cluster.

Since these containers share the same pod, you can use simple and efficient storage options like empty-dir volumes. These volumes are quick to set up and don't require persistent storage like persistent volume claims (PVCs). Empty-dir volumes are temporary storage that exists only for the duration of the pod, making them cheap and fast for sharing data between the containers in your workflow steps.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: container-set-template-
  namespace: argo
spec:
  entrypoint: main
  serviceAccountName: training
  templates:
    - name: main
      volumes:
        - name: workspace
          emptyDir: { }
      containerSet:
        volumeMounts:
          - mountPath: /workspace
            name: workspace
        containers:
          - name: a
            image: argoproj/argosay:v2
            command: [sh,-c]
            args: ["echo 'a: hello world' >> /workspace/message"]
          - name: b
            image: argoproj/argosay:v2
            command: [sh,-c]
            args: ["echo 'b: hello world' >> /workspace/message"]
          - name: main
            image: argoproj/argosay:v2
            command: [sh,-c]
```

```
args: ["echo 'main: hello world' >> /workspace/message"]

dependencies:

  - a

  - b

outputs:

parameters:

  - name: message

valueFrom:

  path: /workspace/message
```

- Name and Namespace: The workflow is named with a generated name starting with "container-set-template-" and is placed in the "argo" namespace.
- Entrypoint: The entrypoint of the workflow is set to "main."
- Service Account: The workflow uses the service account named "training."
- Templates:
  - Main Template:
    - Volumes: Defines an emptyDir volume named "workspace" for sharing data between containers.
    - Container Set:
      - Volume Mounts: Mounts the "workspace" volume inside each container.
      - Containers:
        - Container "a" writes "a: hello world" to a file in the workspace.
        - Container "b" writes "b: hello world" to the same file in the workspace.
        - Container "main" depends on "a" and "b" and writes "main: hello world" to the file in the workspace.
      - Outputs: Defines an output parameter "message" that captures the content of the file written by the containers.



- This workflow demonstrates using a container set template to run multiple containers in the same pod, share data between them using an emptyDir volume, and capture output for further processing.

### The Emissary Executor

- The Emissary Executor in Argo Workflows is a specialized execution engine designed to handle complex workflows efficiently. Here's a detailed explanation of the Emissary Executor:
  - **Workflow Execution:**
    - The Emissary Executor is responsible for executing workflows defined in Argo Workflows.
    - It manages the scheduling, execution, and coordination of workflow tasks and containers.
  - **Task Parallelism:**
    - Emissary supports parallel execution of tasks within a workflow.
    - It can handle multiple tasks running simultaneously, optimizing resource utilization and workflow performance.
  - **Dependency Management:**
    - Emissary manages dependencies between tasks in a workflow.
    - It ensures that tasks are executed in the correct order based on their dependencies, enabling complex workflow structures like DAGs (Directed Acyclic Graphs).
  - **Resource Management:**
    - The executor manages resource allocation and utilization for workflow tasks and containers.
    - It handles resource requests and limits specified in the workflow YAML, such as CPU, memory, and storage requirements.

```
apiVersion: argoproj.io/v1alpha1
```

```
kind: Workflow
```

```
metadata:
```

```
  generateName: diamond-dag-
```

```
  namespace: argo
```

```
spec:
  entrypoint: main
  serviceAccountName: training
  templates:
    - name: main
      dag:
        tasks:
          - name: a
            template: task-a
            resources:
              limits:
                cpu: "1000m" # 1000 millicores = 1 CPU
          - name: b
            dependencies: [a]
            template: task-b
            resources:
              limits:
                cpu: "2000m" # 2000 millicores = 2 CPUs
          - name: c
            dependencies: [a]
            template: task-c
            resources:
              limits:
                cpu: "1000m" # 1000 millicores = 1 CPU
          - name: d
            dependencies: [b, c]
            template: task-d
            resources:
              limits:
```

```
    cpu: "1000m" # 1000 millicores = 1 CPU
- name: task-a
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'a: hello world'"]
- name: task-b
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'b: hello world'"]
- name: task-c
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'c: hello world'"]
- name: task-d
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'd: hello world'"]
```

### Lopsided requests

- **Workflow Overview:** This workflow consists of two tasks, "a" and "b", which are executed in a sequence.
- **Task "a":**
  - **Resources Needed:** Task "a" requires a small amount of computing power (0.1 CPU) and memory (1 MiB).
  - **Execution Time:** It runs for 2 minutes.

- **What It Does:** It prints "a: hello world" and waits for 5 seconds.
- **Task "b":**
  - **Dependencies:** Task "b" depends on task "a", which means it starts only after task "a" completes.
  - **Resources Needed:** Task "b" requires a lot of computing power (8 CPUs), a large amount of memory (100 GiB), and a small portion of a GPU (0.2 GPU).
  - **Execution Time:** It runs for 1 minute.
  - **What It Does:** It prints "b: hello world" and waits for 2 seconds.
- In simple terms, this workflow does two things:
- Task "a" prints a message and waits for a short time.
- Task "b" prints another message after task "a" is done, but it needs a lot more computing resources and memory to run.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: lopsided-requests-
  namespace: argo
spec:
  entrypoint: main
  serviceAccountName: training
  templates:
    - name: main
      dag:
        tasks:
          - name: a
            template: task-a
            resources:
              limits:
                cpu: "100m" # 100 millicores = 0.1 CPU
```

```
    memory: "1Mi" # 1 Mebibyte
  retryStrategy:
    limit: 3
    retryPolicy: Always
    duration: "2m" # 2 minutes
- name: b
  dependencies: [a]
  template: task-b
  resources:
    limits:
      cpu: "8000m" # 8000 millicores = 8 CPUs
      memory: "100Gi" # 100 Gibibytes
      nvidia.com/gpu: "200m" # 200 millicores = 0.2 GPU
  retryStrategy:
    limit: 1
    retryPolicy: Never
    duration: "1m" # 1 minute
- name: task-a
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'a: hello world' && sleep 5"]
- name: task-b
  container:
    image: argoproj/argosay:v2
    command: [sh,-c]
    args: ["echo 'b: hello world' && sleep 2"]
```

## ContainerSet Retry

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: containerset-with-retrystrategy
  namespace: argo
  annotations:
    workflows.argoproj.io/description: |
      This workflow creates a container set with a retryStrategy.
spec:
  entrypoint: containerset-retrystrategy-example
  serviceAccountName: training
  templates:
    - name: containerset-retrystrategy-example
      containerSet:
        retryStrategy:
          retries: "10" # if fails, retry at most ten times
          duration: 30s # retry for at most 30s
        containers:
          # this container completes successfully, so it won't be retried.
          - name: success
            image: python:alpine3.6
            command:
              - python
              --c
            args:
              - |
                print("hi")

            # if fails, it will retry at most ten times.
```

```
- name: fail-retry
  image: python:alpine3.6
  command: ["python", "-c"]
  # fail with a 66% probability
  args: ["import random; import sys; exit_code = random.choice([0, 1, 1]);
  sys.exit(exit_code)"]
```

## Artifacts

Imagine you have a set of instructions (templates) to do different tasks, like baking a cake or painting a picture. These instructions can take in specific inputs, like the type of cake you want or the colors you want to use.

Now, workflows are like following these instructions step by step. You provide the necessary inputs (parameters) at each step, making sure everything runs smoothly.

For example, if your cake recipe template asks for the type of frosting (a parameter), you would provide vanilla or chocolate when you follow the recipe (run the workflow).

In the same way, when you use a DAGTemplate (a type of workflow), you're organizing these tasks in a specific order, like first baking the cake and then decorating it. The examples given are just like different ways of customizing your cake or painting based on what you want, making sure everything is clear and straightforward.

## Input Parameters

- **Workflow:** It's like a master plan or a checklist for a series of tasks. When you run a workflow, you can give it arguments, which are like special instructions or details for how to do those tasks.
- **Template:** Think of a template as a mini-plan for a specific task within the master plan (workflow). Templates define inputs, which are like blanks that need to be filled in when you use the template. These inputs are provided by whoever calls the template, whether it's a step within the workflow, another template, or even the main workflow itself.

```
metadata:
  name: wonderful-rhino
  namespace: argo
  labels:
    example: 'true'
spec:
  arguments:
```

```
parameters:
  - name: message
    value: hello argo
entrypoint: argosay
serviceAccountName: argo
templates:
  - name: argosay
    inputs:
      parameters:
        - name: message
          value: '{{workflow.parameters.message}}'
    container:
      name: main
      image: argoproj/argosay:v2
      command:
        - /argosay
      args:
        - echo
        - '{{inputs.parameters.message}}'
    ttlStrategy:
      secondsAfterCompletion: 300
    podGC:
      strategy: OnPodCompletion
```

Run from command line with inputs

```
argo submit-n argo inputwf.yaml-p 'workflow-param-1="abcd"' --watch
```

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: example-
  namespace: argo
spec:
  entrypoint: main
  serviceAccountName: training
  arguments:
    parameters:
      - name: workflow-param-1
  templates:
    - name: main
      dag:
```



```
tasks:
- name: step-A
  template: step-template-a
  arguments:
    parameters:
      - name: template-param-1
        value: "{{workflow.parameters.workflow-param-1}}"

- name: step-template-a
  inputs:
    parameters:
      - name: template-param-1
  script:
    image: alpine
    command: [/bin/sh]
    source: |
      echo "{{inputs.parameters.template-param-1}}"
```

DAG: Previous Step Outputs As Inputs

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: artifact-passing-
  namespace: argo
spec:
  entrypoint: artifact-example
  serviceAccount: training
  templates:
    - name: artifact-example
      steps:
        - name: generate-artifact
          template: whalesay
        - name: consume-artifact
          template: print-message
          arguments:
            artifacts:
              # bind message to the hello-art artifact
              # generated by the generate-artifact step
              - name: message
                from: "{{steps.generate-artifact.outputs.artifacts.hello-art}}"

    - name: whalesay
      container:
        image: docker/whalesay:latest
```

```
command: [sh, -c]
args: ["cowsay hello world | tee /tmp/hello_world.txt"]
outputs:
  artifacts:
    # generate hello-art artifact from /tmp/hello_world.txt
    # artifacts can be directories as well as files
    - name: hello-art
      path: /tmp/hello_world.txt

- name: print-message
  inputs:
    artifacts:
      # unpack the message input artifact
      # and put it at /tmp/message
      - name: message
        path: /tmp/message
  container:
    image: alpine:latest
    command: [sh, -c]
    args: ["cat /tmp/message"]
```

### Key-Only Artifacts

Using key-only artifacts means you're only specifying the essential information about the artifact, like its name or key, without detailing the specific location (bucket) or any secret credentials. This simplifies your workflow specifications by letting the artifact repository's default settings handle the actual storage and access details.

Here's a simplified explanation of why key-only artifacts are beneficial:

- **Reduced Workflow Size:** By omitting detailed artifact location information, your workflows become smaller, which can lead to improved performance.
- **User-Owned Artifact Repository Setup:** Key-only artifacts allow you to manage your artifact repository settings separately, making it easier to configure and maintain.
- **Decoupled Configuration:** Since key-only artifacts rely on the artifact repository's default setup, you can change your repository settings without needing to modify your workflows or templates, providing flexibility and easier maintenance.

### Configuring Artifactory

To use Argo workflows with artifacts, you need to set up an artifact repository. This repository stores and manages the files your workflows use. Argo works with various S3-compatible repositories like those from AWS, Google Cloud Storage (GCS), and MinIO.

Here's a simple breakdown of what this means:

Mail us: [info@openwriteup.com](mailto:info@openwriteup.com)

1. **Purpose of Artifact Repository:** The artifact repository holds the files needed for your workflows, like data, scripts, or configuration files.
2. **Supported Repositories:** Argo can work with S3-compatible repositories, which include popular services like AWS S3 and Google Cloud Storage, as well as self-hosted solutions like MinIO.
3. **Configuration:** Configuring the artifact repository involves setting up credentials, defining storage locations (buckets), and ensuring access permissions for Argo workflows.
4. **Usage in Workflows:** Once configured, you can reference and use artifacts from the repository in your Argo workflows, making it easy to manage and access necessary resources during workflow execution.

Setup the S3

```
$ export mybucket=bucket249
$ cat > policy.json <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:DeleteObject"
      ],
      "Resource": "arn:aws:s3:::$mybucket/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": "arn:aws:s3:::$mybucket"
    }
  ]
}
EOF
```

```
$ aws s3 mb s3://$mybucket [--region xxx]
$ aws iam create-user --user-name $mybucket-user
```

```
$ aws iam put-user-policy --user-name $mybucket-user --policy-name $mybucket-policy --
policy-document file://policy.json
$ aws iam create-access-key --user-name $mybucket-user > access-key.json
```

Modify the configmap

```
kubectl edit configmap workflow-controller-configmap -n argo # assumes
argo was installed in the argo namespace
```

```
...
data:
  artifactRepository: |
    s3:
      bucket: my-bucket
      keyFormat: prefix/in/bucket #optional
      endpoint: my-minio-endpoint.default:9000 #AWS => s3.amazonaws.com; GCS =>
storage.googleapis.com
      insecure: true #omit for S3/GCS. Needed when minio runs without TLS
      accessKeySecret: #omit if accessing via AWS IAM
        name: my-minio-cred
        key: accessKey
      secretKeySecret: #omit if accessing via AWS IAM
        name: my-minio-cred
        key: secretKey
      useSDKCreds: true #tells argo to use AWS SDK's default provider chain, enable for
things like IRSA support
```

Example

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  artifactRepository: |
    s3:
      bucket: amitow23
      #keyFormat: prefix/in/bucket # optional
```

```
    endpoint: s3.amazonaws.com    # AWS => s3.amazonaws.com; GCS =>
storage.googleapis.com

    insecure: true                # omit for S3/GCS. Needed when MinIO runs without TLS

    accessKeySecret:
      name: my-minio-cred
      key: accessKey

    secretKeySecret:
      name: my-minio-cred
      key: secretKey

    useSDKCreds: true

kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"ConfigMap","metadata":{"annotations":{},"name":"workflow-
controller-configmap","namespace":"argo"}}
  creationTimestamp: "2024-05-11T12:26:10Z"
  name: workflow-controller-configmap
  namespace: argo
  resourceVersion: "120715"
  uid: 191a6596-d742-496d-a08f-85776b4872b7
```