Cloud-Native CI/CD with Tekton - Labs

Creating pipelines with custom Kubernetes resources

Session Labs: Revision 1.1 - 07/18/21

Brent Laster - Lead Instructor, Tech Skills Transformations LLC

Important Note: Prior to starting with this document, you should have the ova file for the class (the virtual machine) already loaded into Virtual Box and have ensured that it is startable. See the setup doc tekton-intro-setup.pdf in http://github.com/skilldocs/tekton for instructions and other things to be aware of.

Setup: To do before the first lab

1. For this workshop, files that we need to use are contained in the directory **tekton-intro** in the home directory on the disk. Change into that directory. Open a terminal emulator, and enter the command below:

   ```
   $ cd tekton-intro
   ```

2. To run the examples in this set of labs, we need a Kubernetes instance. Start up a minikube instance on the machine with the command below.

   ```
   $ ./setup.sh
   ```

Lab 1: Working with Tekton Tasks

Purpose: In this lab, we'll see how to run and use a basic Tekton Task.

1. To run the examples in this set of labs, first change into the directory for lab 1.

   ```
   $ cd lab1
   ```

2. Before we do anything else, let's go ahead and open the Tekton dashboard. Open a web browser by clicking on the mouse icon in the upper left and then selecting the **Web Browser** menu item.
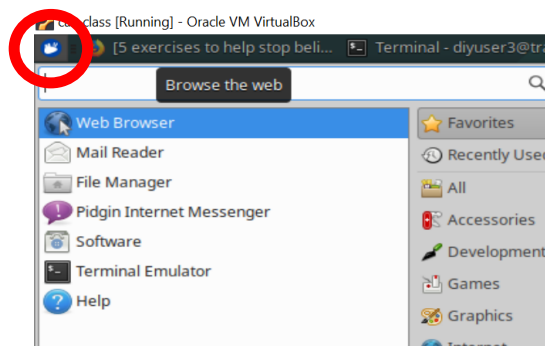
3. The dashboard pod itself should have already been installed on your system via the setup script and should be running. So all we need to do is a port-forward command and then open it up in a browser session. It is suggested to do this in a separate terminal session.

```
$ kubectl --namespace tekton-pipelines port-forward svc/tekton-dashboard 9097:9097 &
```

Then, open a tab in the browser to: http://127.0.0.1:9097

(Note that we are using 127.0.0.1 specifically here instead of localhost due to a problem with port-forward and localhost timing out.)
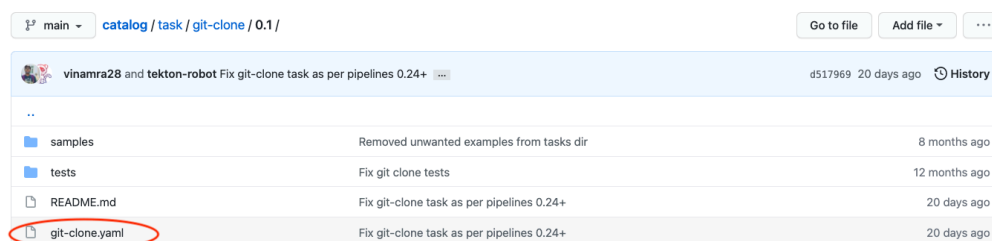
4. For this lab, we are going to see how to run a simple, predefined Task - git-clone. To start with, take a look at the actual Task as it is defined in the Tekton Catalog.

Go to the link for the main page of the task:

https://github.com/tektoncd/catalog/tree/main/task/git-clone/0.1

You can read about the task on there if you like. Note that is intended as a replacement for the former "Pipeline Resource" for a Git repository.

Click on the git-clone.yaml link in the list of files at the top to see the actual task definition.

| ⌥ main ▾	catalog / task / git-clone / 0.1 /		Go to file	Add file ▾	⋯
vinamra28 and tekton-robot Fix git-clone task as per pipelines 0.24+ …			d517969 20 days ago	⏱ History	
..					
📁 samples	Removed unwanted examples from tasks dir			8 months ago	
📁 tests	Fix git clone tests			12 months ago	
🗋 README.md	Fix git-clone task as per pipelines 0.24+			20 days ago	
🗋 git-clone.yaml	Fix git-clone task as per pipelines 0.24+			20 days ago	

(There's a lot of content in there, but if you skip past the possible parameters and environment settings, you can see most of the main functionality at the bottom.)

5. To run this task, we need it installed locally. We can do this easily via the "tkn hub" command. Execute the command below to install the git-clone Task in our tekpipe namespace. Then we'll go ahead and install the kaniko Task for building images that we'll need later as well.

2

```
$ tkn hub -n tekpipe install task git-clone
$ tkn hub -n tekpipe install task kaniko
```

6. You can now get a list of these from the command line. As well, you can look in the dashboard and see these items listed in the Tasks section (selected on the left-hand side). (Make sure to have the tekpipe namespace selected in the upper right.)

```
$ k get -n tekpipe tasks
```
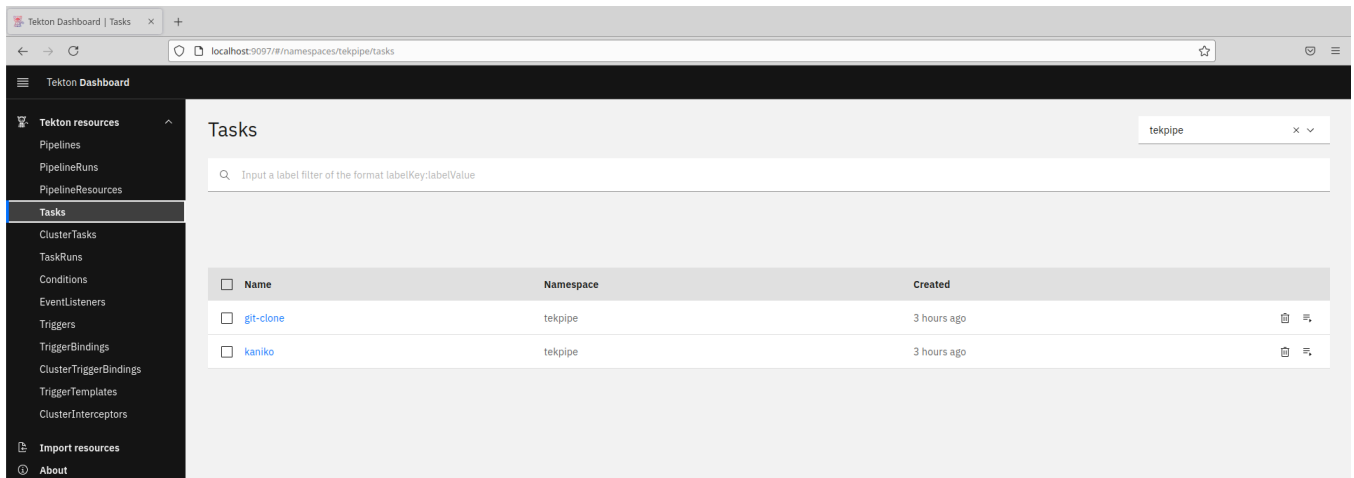

7. In order to run this Task, we will need a persistent volume claim set up. There is already one defined in the tekton-pvc.yaml file in the current directory. Go ahead and apply it into the cluster.

```
$ k apply -n tekpipe -f tekton-pvc.yaml
```

8. Now we can attempt to run our task via the **tkn** command line utility. The fairly long command is below. You can type this in if you want or you can simply run the shell script **tkn-git-clone.sh** that is in the directory. Execute this from a terminal session.

```
$ ./tkn-git-clone.sh
```

OR

```
$ tkn task start git-clone --namespace=tekpipe \
    --param url=https://github.com/skillrepos/tekton-docker-
k8s.git \
    --param revision=main \
    --param deleteExisting=true \
```

```
--workspace name=output,claimName=git-files-pvc \
--use-param-defaults \
--showlog
```

9. This will create a TaskRun object and run the pre-defined git-clone task. You can see it execute in the terminal window or you can go to the Dashboard and look at it there, under the TaskRuns section (on the left). Make sure to have the "tekpipe" namespace selected in the upper right box.
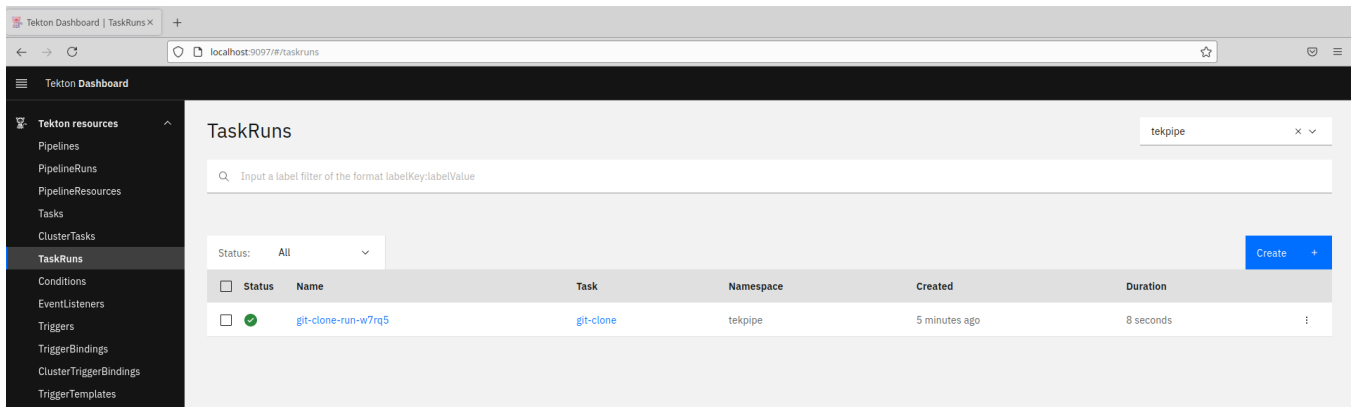
10. You can click on that TaskRun there to get more details, logs, and status. Click on the TaskRun and browse those tabs. This will take a while to startup and execute.
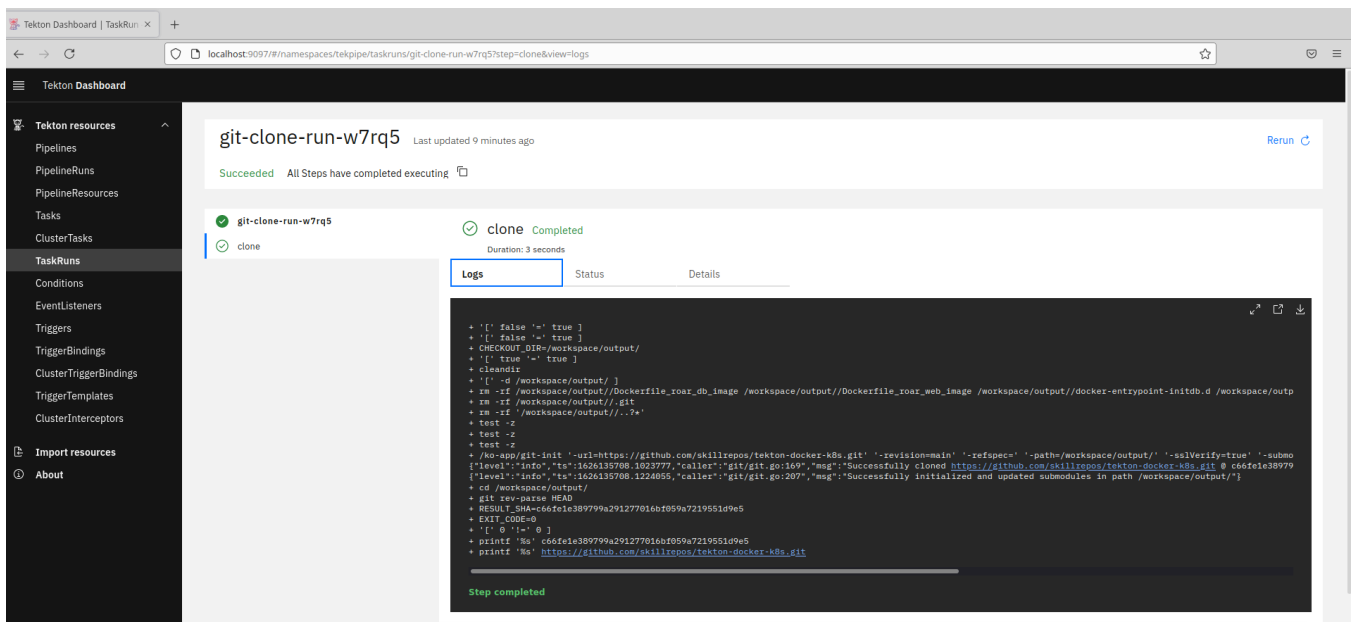
11. Now let's see how this same TaskRun can be done via a manifest file. In the same directory, take a look at the file **taskrun-git-clone.yaml**. Then you can use the "kubectl create" command (NOT apply) to create the TaskRun object and invoke it.

```
$ cat taskrun-git-clone.yaml

$ k create -n tekpipe -f taskrun-git-clone.yaml
```

12. From here, you can look at the new TaskRun and its details, logs, etc. in the dashboard just as you did the previous one.

Lab 2: Creating Tekton Tasks

Purpose: In this lab, we'll dive deeper into how to construct Tekton Tasks

13. To run the examples in this set of labs, first change into the directory for lab2.

```
$ cd ../lab2
```

14. We will construct the two other Tasks for our pipeline section-by-section so you can better understand what makes up a task. To simplify the creation due to time and syntax, we'll use a program called "meld" to add content each time. At each point, you will click on the indicated arrow to merge the content. Then the files should be identical, and you can save your changes and exit meld. Pay attention to the order and names of the files.

15. We'll start with the basic pieces of our first task and add in the parameters (including the workspace) that it needs.

    The parameters include the path to the Kubernetes manifest file, and the locations of our updated image files. The idea is that changes are made to the code, new docker images are built with the changes, and then this task uses the Linux "sed" tool to update the location of the images that are part of the pod and deployment specs in our Kubernetes deployment manifest.

    In a terminal session, run the command below (in the lab2 subdirectory).

    ```
    $ meld tasks.yaml task1.yaml.params
    ```
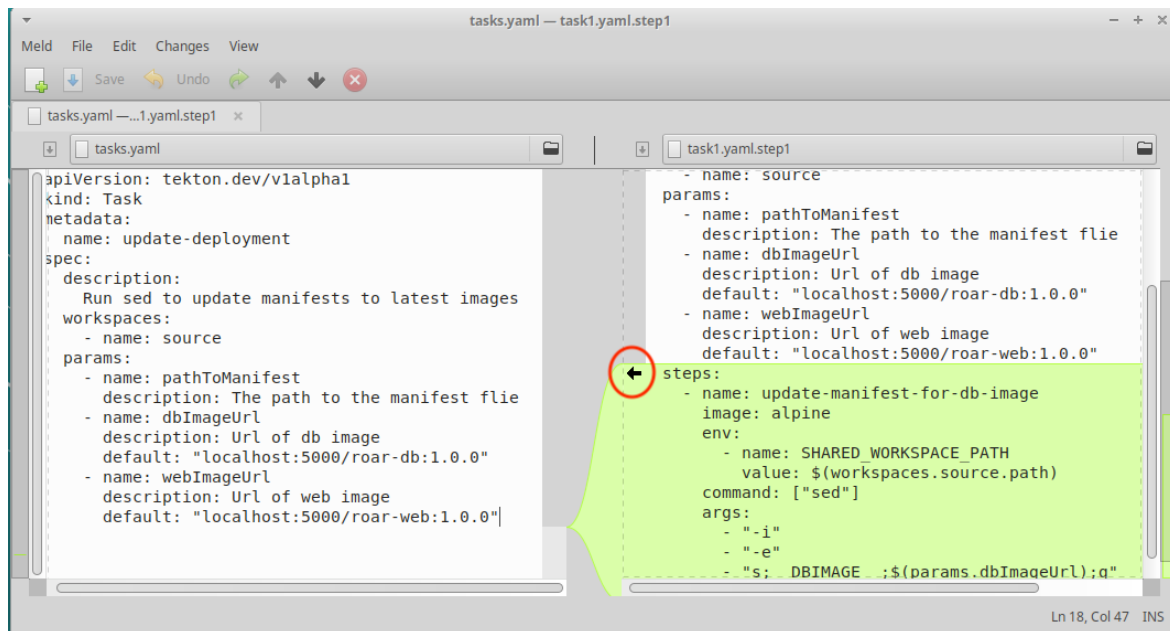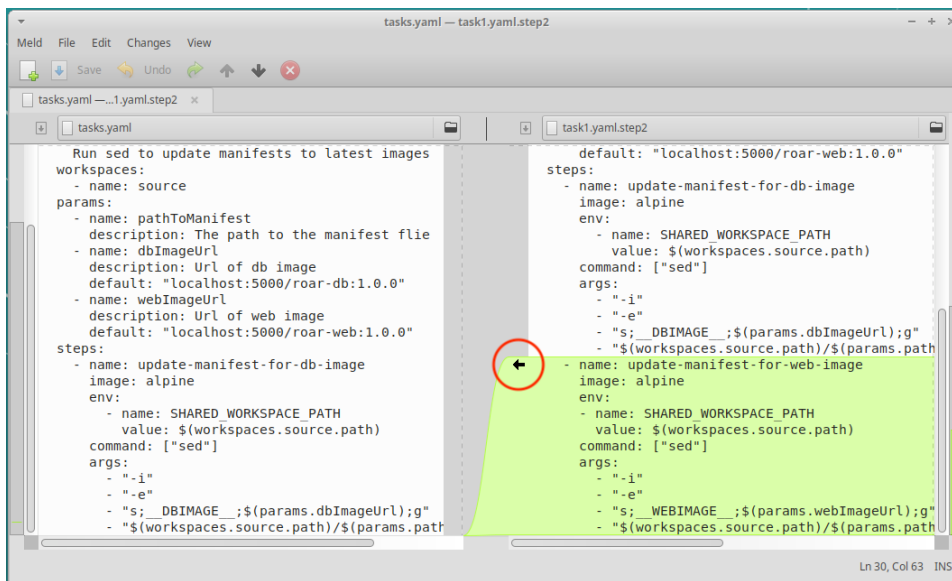

    Review the code changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.
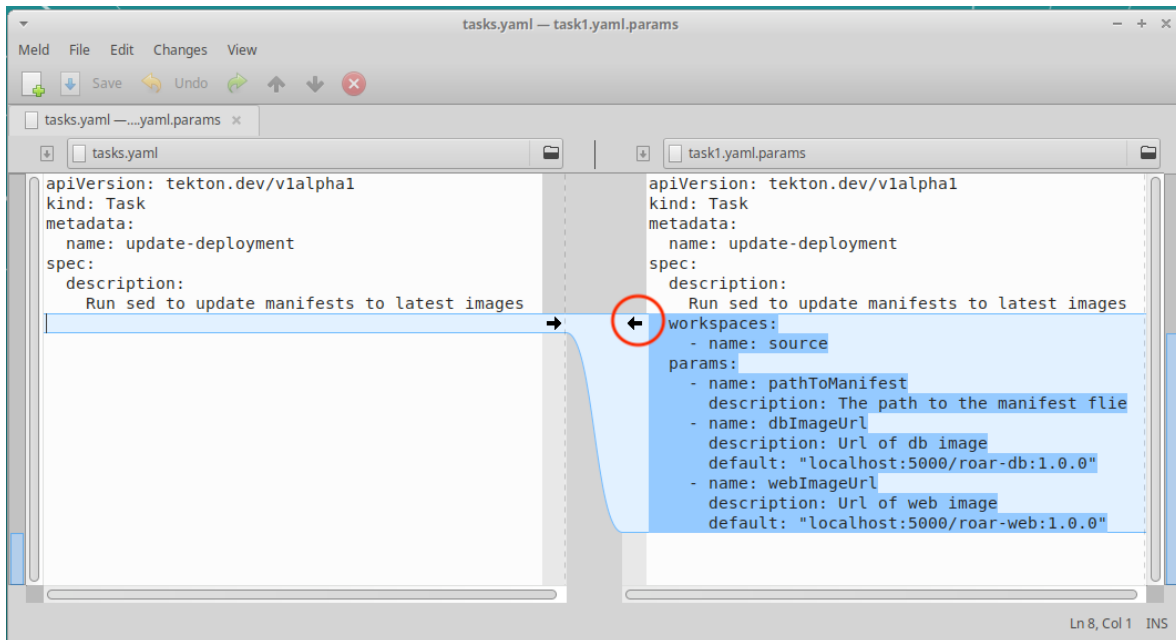
16. Next, we'll add in the first step in the task. This one runs the "sed" command to update the image location for our database image in the Kubernetes deployment manifest.

```
$ meld tasks.yaml task1.yaml.step1
```
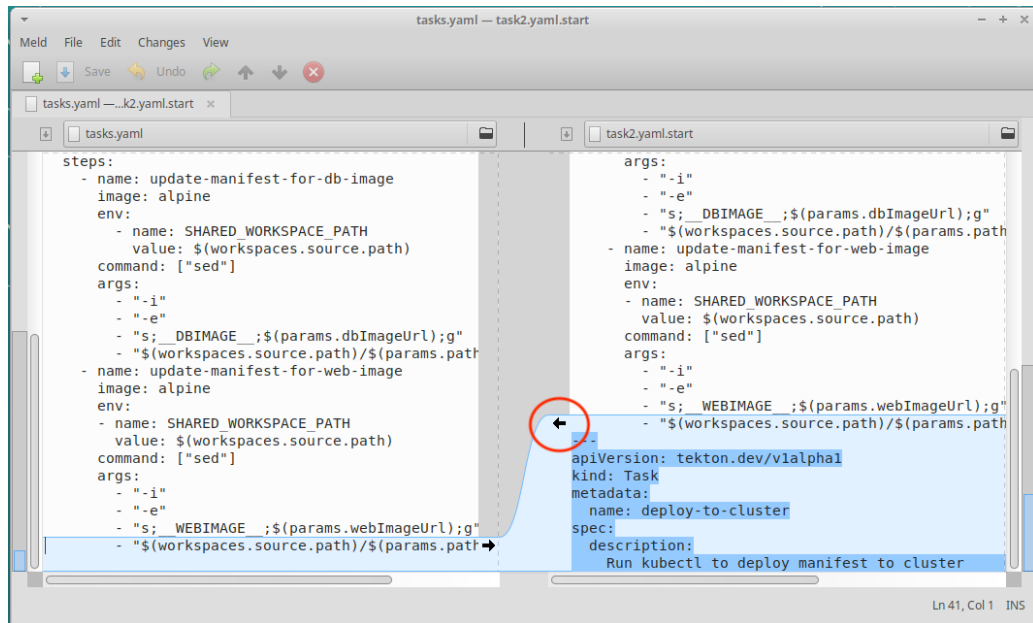
Review the code changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.

17. After that comes the second step. This one runs the "sed" command to update the image location for our web image in the Kubernetes deployment manifest.
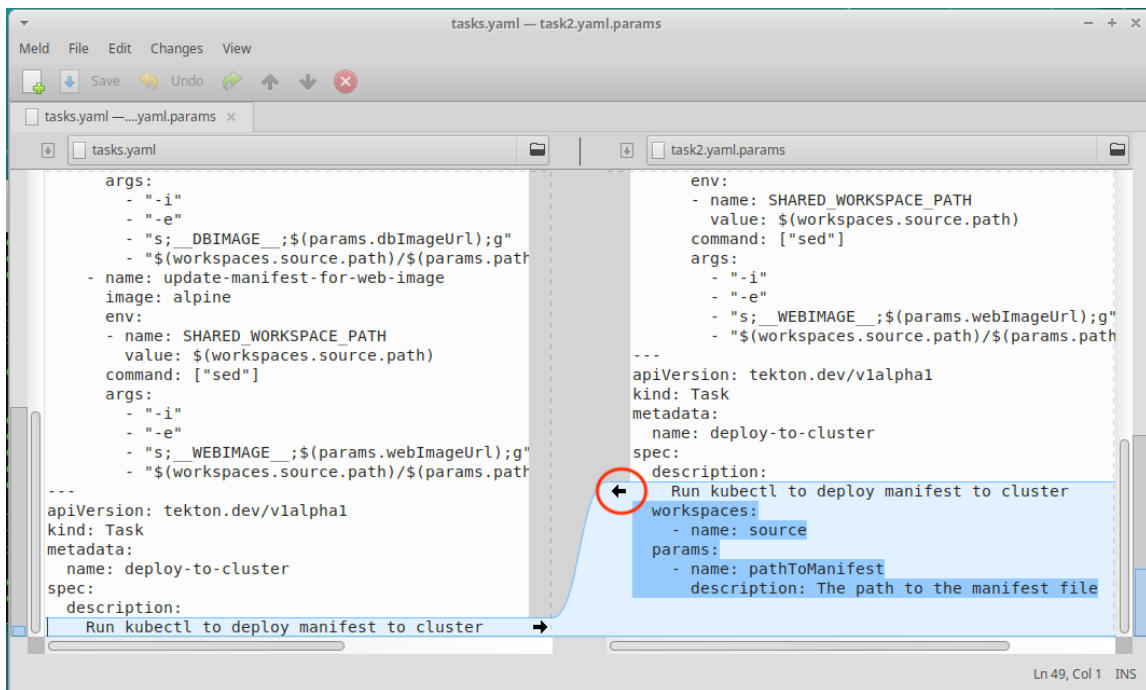
```
$ meld tasks.yaml task1.yaml.step2
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.

18. Our first task is completed. Now we'll add a second task to actually apply the manifest we updated in the previous task to our cluster. We'll do that via running the "kubectl apply" command from an image.

```
$ meld tasks.yaml task2.yaml.start
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.

19. Now we'll add in the parameters and workspace definitions. There is only one parameter in this task - a path to the updated manifest file to be applied to the cluster.

```
$ meld tasks.yaml task2.yaml.params
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.

20. Finally, we'll add in the step for this task. As mentioned before, this simply runs kubectl apply from an image to update

    ```
    $ meld tasks.yaml task2.yaml.steps
    ```

    Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the tasks file with the change. Then Save your changes and exit meld.
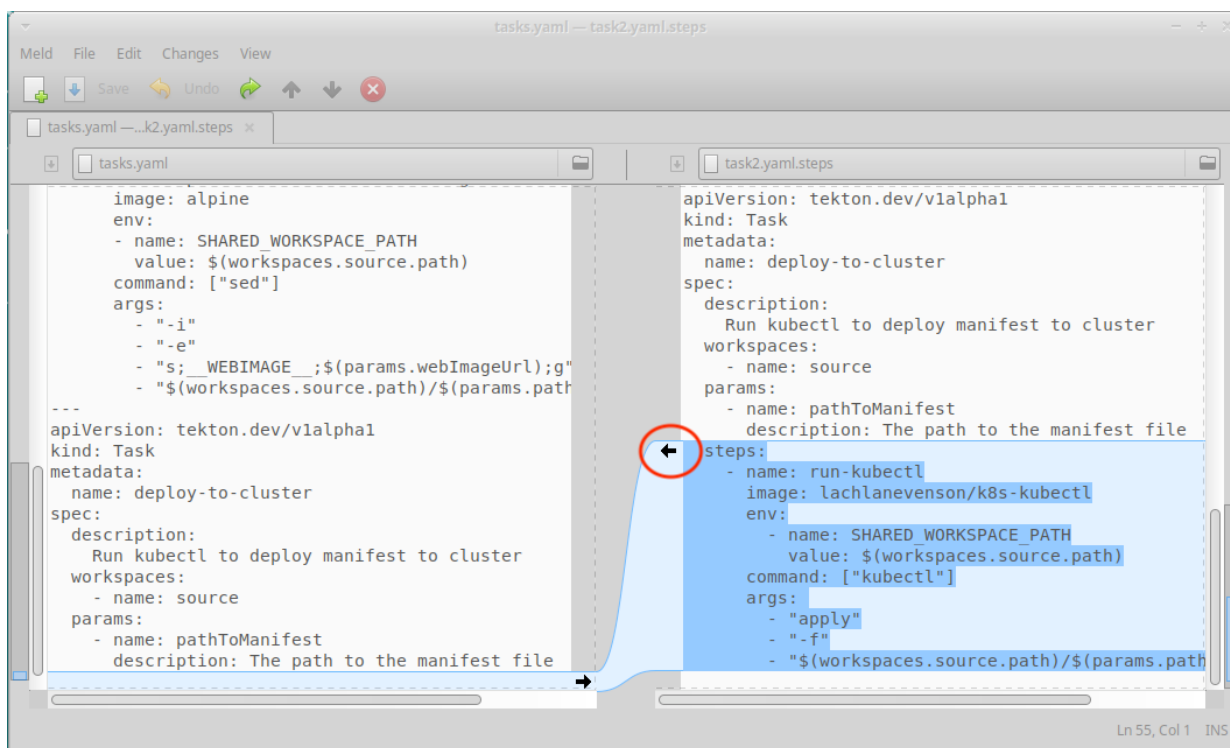
21. Now that we've created these tasks, we'll go ahead and apply them in the namespace in the cluster.

    ```
    $ k apply -f tasks.yaml -n tekpipe
    ```

22. Take a quick look at the set of tasks we have in the namespace.

    ```
    $ k get tasks -n tekpipe
    ```

    You can also look at the list in the Dashboard if you want.

Lab 3: Creating Tekton Pipelines

Purpose: In this lab, we'll see how to create a Tekton Pipeline to orchestrate our tasks and run it.

1. To work with the examples in this set of labs, first change into the directory for lab3.
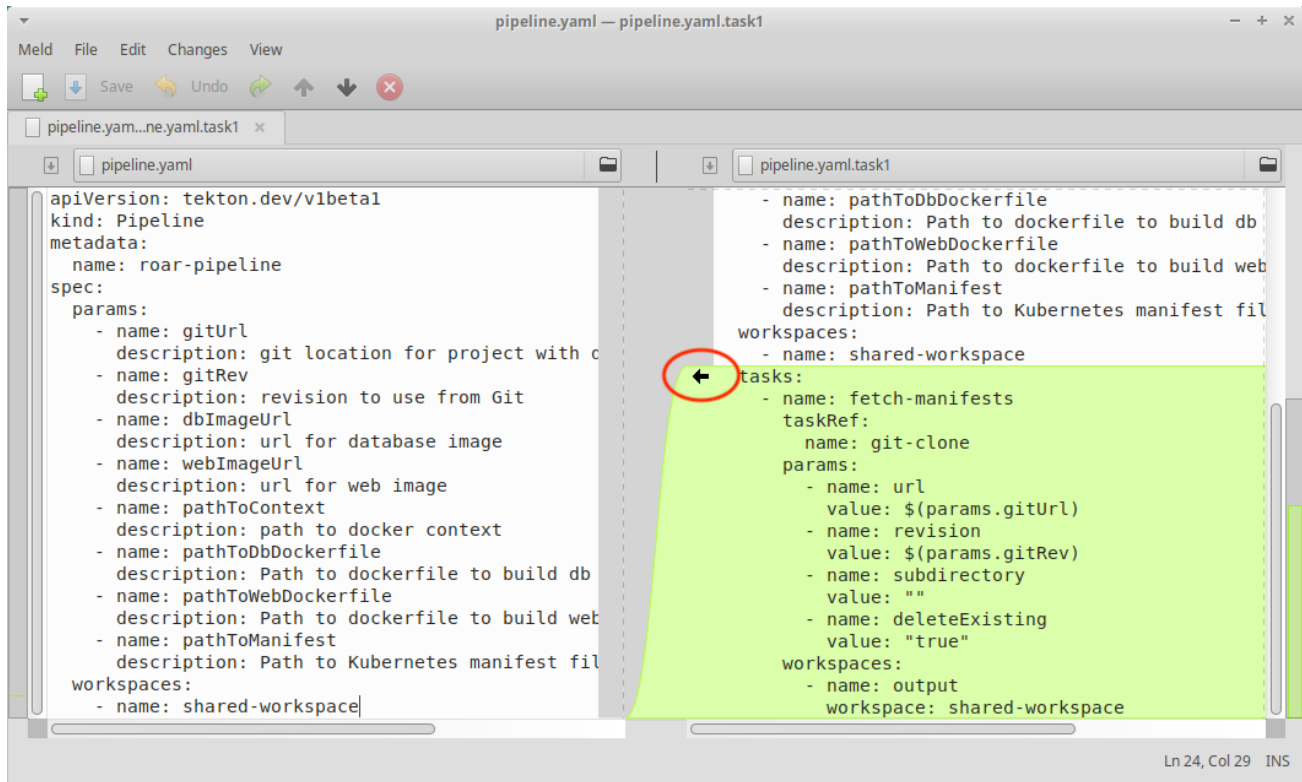
```
$ cd ../lab3
```

2. Because of the amount of code involved, we'll use the same approach as in the previous lab to construct the pipeline spec. We'll start by adding in all of the parameters that the pipeline will need.

   These parameters include:
   a. The Git URL and revision (branch) where the content to build our images is stored.
   b. The locations for our updated database and webapp images after they are rebuilt.
   c. Paths to the Docker files for building the updated images.
   d. Path to the Kubernetes manifest to updated with the new images.

   ```
   $ meld pipeline.yaml pipeline.yaml.params
   ```
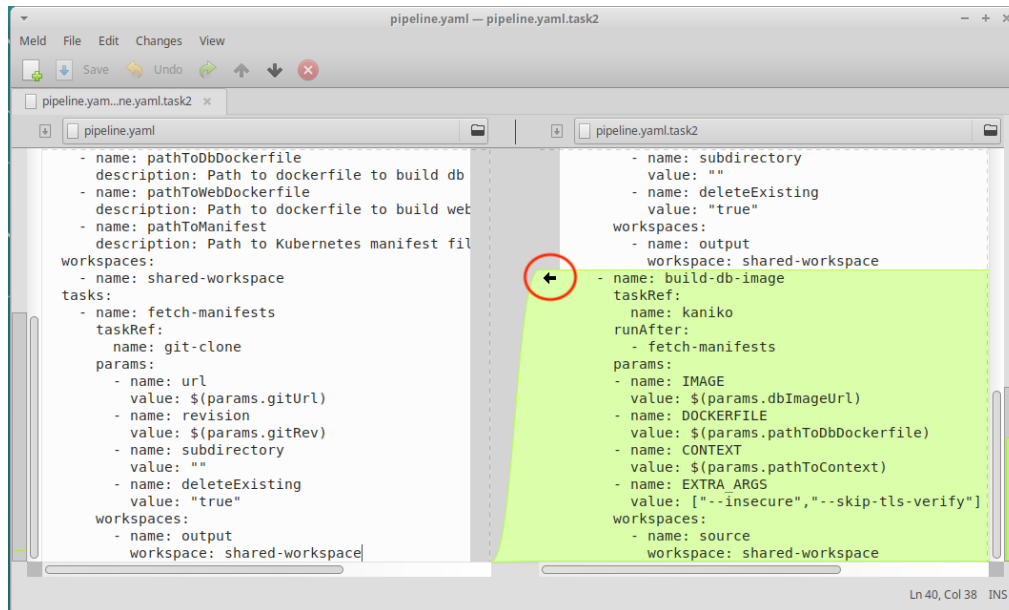
   Review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.
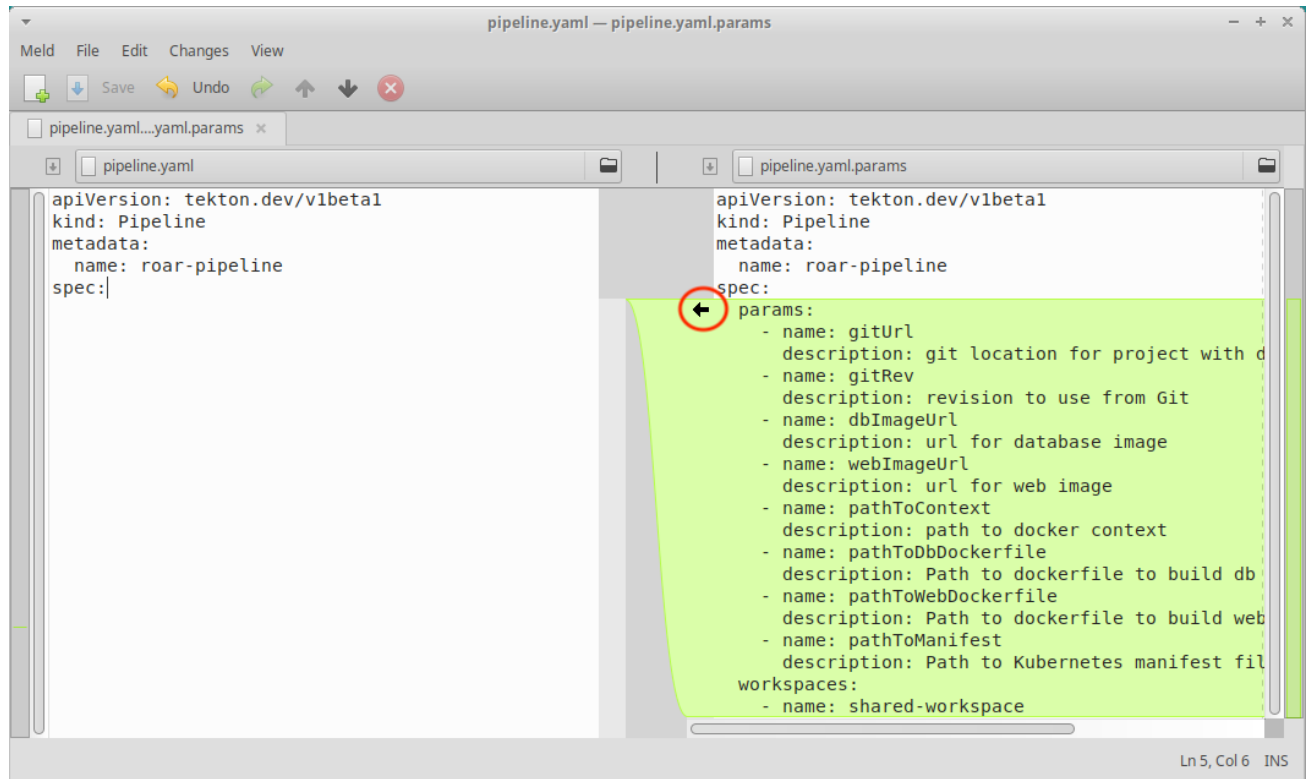
3. Next, we'll add in the call to the first task - fetch-manifests. Notice that the "taskRef" element tells us that this is really a call to the "git-clone" task that we saw in the first lab. We pass in the parameters appropriate to that task here along with the workspace for it to use.
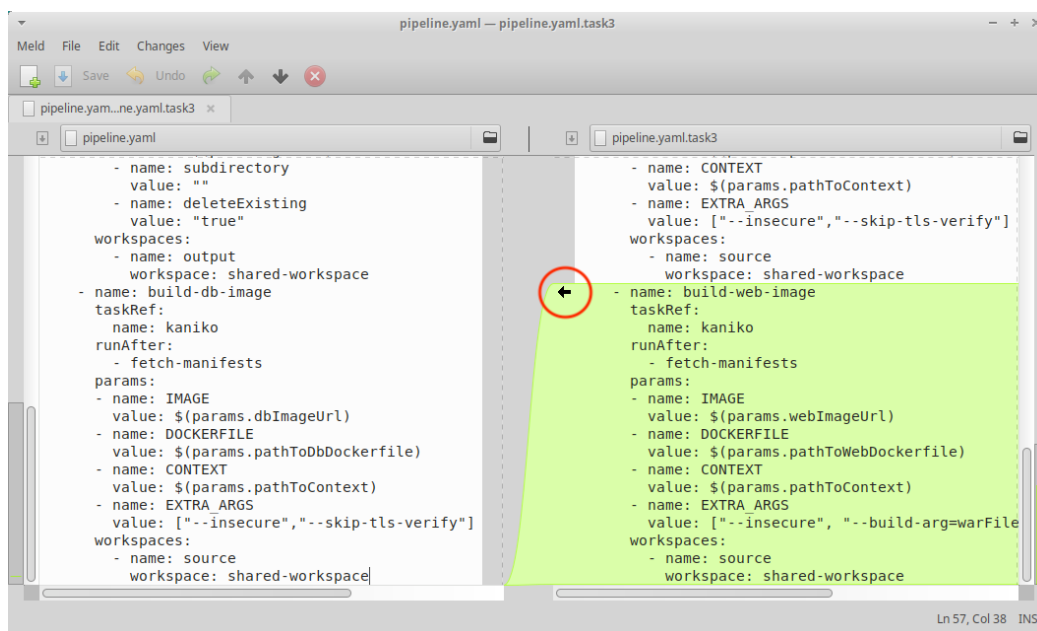
```
$ meld pipeline.yaml pipeline.yaml.task1
```

Review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

```
pipeline.yaml — pipeline.yaml.task1
Meld   File   Edit   Changes   View
       Save    Undo

pipeline.yam...ne.yaml.task1  ×

pipeline.yaml                                    pipeline.yaml.task1

apiVersion: tekton.dev/v1beta1                        - name: pathToDbDockerfile
kind: Pipeline                                          description: Path to dockerfile to build db
metadata:                                             - name: pathToWebDockerfile
  name: roar-pipeline                                   description: Path to dockerfile to build web
spec:                                                 - name: pathToManifest
  params:                                               description: Path to Kubernetes manifest fil
    - name: gitUrl                                  workspaces:
      description: git location for project with c    - name: shared-workspace
    - name: gitRev                             ←  tasks:
      description: revision to use from Git            - name: fetch-manifests
    - name: dbImageUrl                                  taskRef:
      description: url for database image                 name: git-clone
    - name: webImageUrl                                 params:
      description: url for web image                      - name: url
    - name: pathToContext                                 value: $(params.gitUrl)
      description: path to docker context                - name: revision
    - name: pathToDbDockerfile                            value: $(params.gitRev)
      description: Path to dockerfile to build db        - name: subdirectory
    - name: pathToWebDockerfile                           value: ""
      description: Path to dockerfile to build web       - name: deleteExisting
    - name: pathToManifest                                value: "true"
      description: Path to Kubernetes manifest fil     workspaces:
  workspaces:                                            - name: output
    - name: shared-workspace|                              workspace: shared-workspace

                                                                        Ln 24, Col 29   INS
```

4. Now comes the call to the second task - build-db-image. Notice that the "taskRef" element tells us that this is really a call to the "kaniko" task. We pass in the parameters appropriate to that task here along with the workspace for it to use.

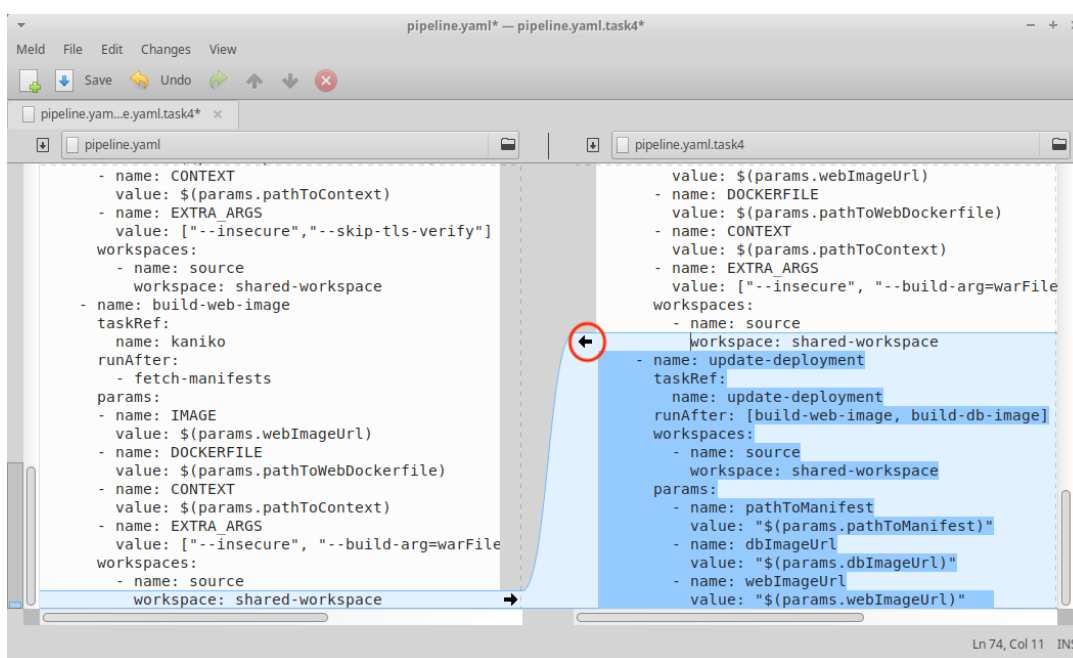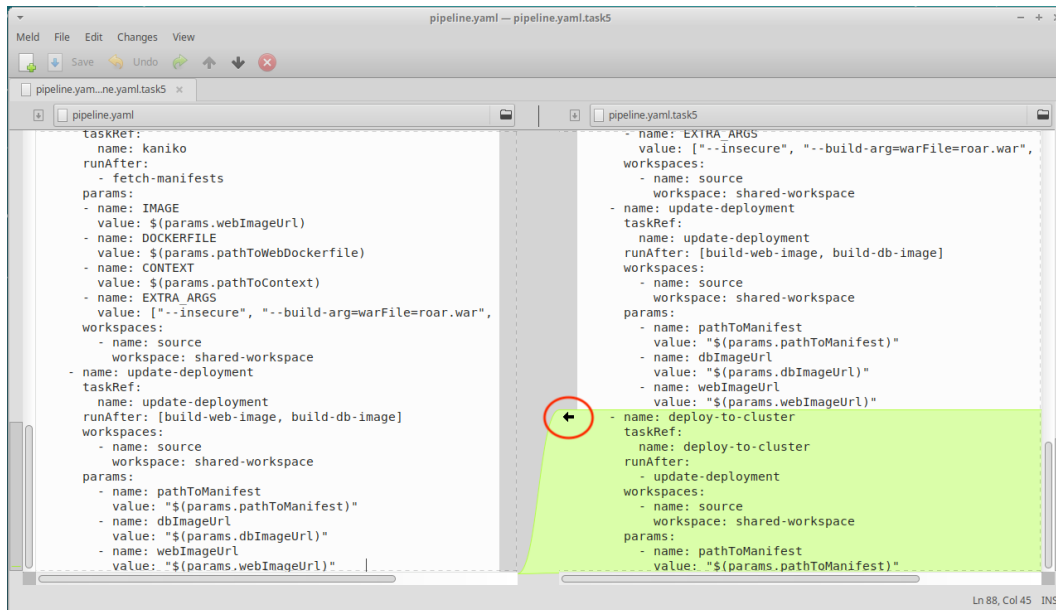 $ meld pipeline.yaml pipeline.yaml.task2

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

5. Add in the call to the third task - build-web-image. Notice that the "taskRef" element tells us that this is really a call to the "kaniko" task. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task3
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

13

6. Add in the call to the fourth task - the one to update the deployment manifest with the updated image information. Notice that the "taskRef" element points to one of the tasks we created earlier. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task4
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

7. Finally, we'll add the call to the fifth task - the one to update the cluster based on our updated deployment manifest with the updated image information. Notice that the "taskRef" element points to one of the tasks we created earlier. We pass in the parameters appropriate to that task here along with the workspace for it to use.

```
$ meld pipeline.yaml pipeline.yaml.task5
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

8. Our Pipeline spec is complete. Go ahead and instantiate it in the cluster and then open up the Tekton dashboard to note the roar-pipeline object is present in the tekpipe namespace.

```
$ k apply -f pipeline.yaml -n tekpipe
```
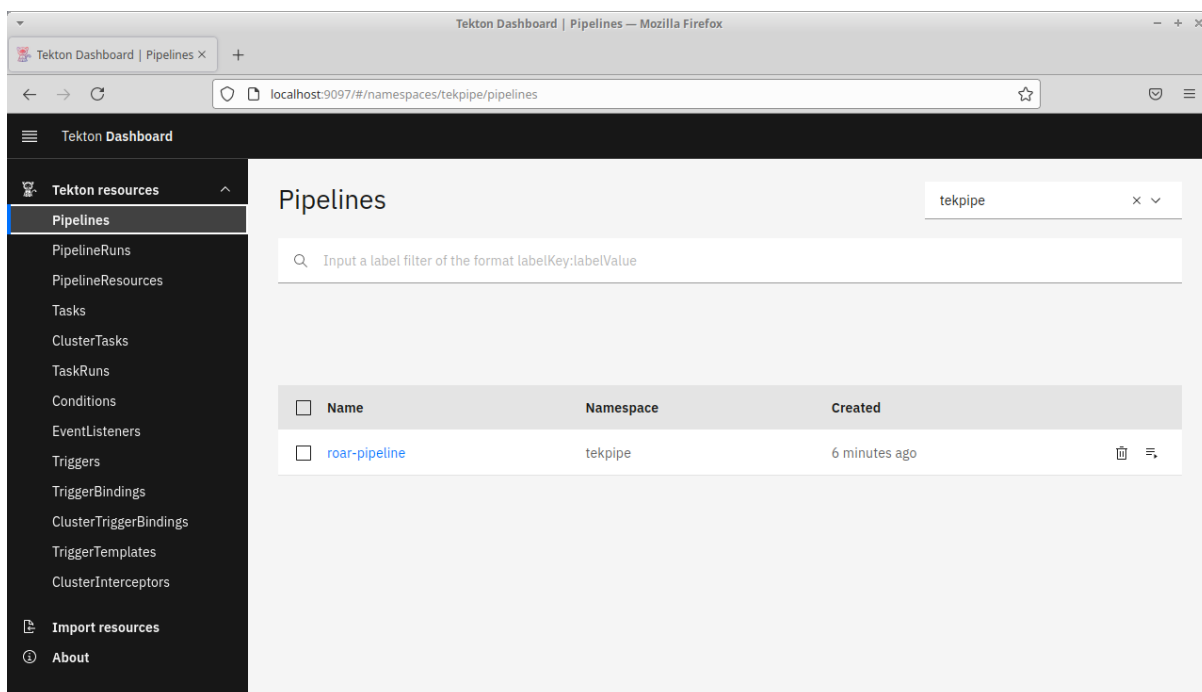
9. Now that we have a pipeline, we need some way to run it. To do this, we need a PipelineRun object - similar to the TaskRun object we used in Lab 1. The PipelineRun needs to ultimately render a Pipeline object with the necessary parameters to pass into the Pipeline. There is already a spec for a PipelineRun object in this directory. Take a look at it via the command below. Note that it has a PipelineRef to the Pipeline we just created. And it specifies parameter values needed for the Pipeline to run.

```
$ cat pipelinerun.yaml
```

10. As well, we need a Service Account to run our Pipeline under as well as the corresponding Role and RoleBinding. In this directory, there is a file with the necessary specs in it - pipeline-rbac.yaml. Take a look at that file and then apply it to set up the needed rbac pieces for running the pipeline. Notice in the RBAC settings we are setting up the RoleBinding to be active in the target namespace for our app - roar.

```
$ cat pipeline-rbac.yaml
$ k apply -f pipeline-rbac.yaml
```

11. Now apply the PipelineRun manifest. After you apply this , you can switch back to the dashboard and see the PipelineRun. You can also see the Logs and Status by clicking on those tabs. Note this will take quite a while to run. You can also see the pods that were spun up for the different tasks by looking in the tekpipe namespace.

```
$ k apply -n tekpipe -f pipelinerun.yaml

$ k get pods -n tekpipe
```

12. You can see the resulting deployment from this in the roar namespace. You can also find the service and open up the URL to see the application that was deployed.

```
$ k get all -n roar

$ k get svc -n roar | grep web

Find the NodePort in the PORT(S) column after "8089:"

Open in browser - http://localhost:<NodePort>/roar/
```

<div align="center">END OF LAB</div>

Lab 4: Working with Triggers

**Purpose:** In this lab, we'll explore how to drive Pipelines via Tekton Triggers.

1. To work with the examples in this set of labs, first change into the directory for lab 4.

```
$ cd ../lab4
```

2. In order to implement CI, we need to be able to trigger the pipeline to run based on some event. We can do that with the add-on Tekton Triggers component. This is already installed on your system, so we just need to construct the yaml spec for it. We'll start with a spec that defines a template and parameters for a PipelineRun. First we'll add the parameters that we need. As before, we'll use the meld approach to add sections at a time. First, we'll add in the parameters for the Git repository, revision and the service account to run the triggers under.

```
$ meld triggers.yaml triggers.yaml.params
```
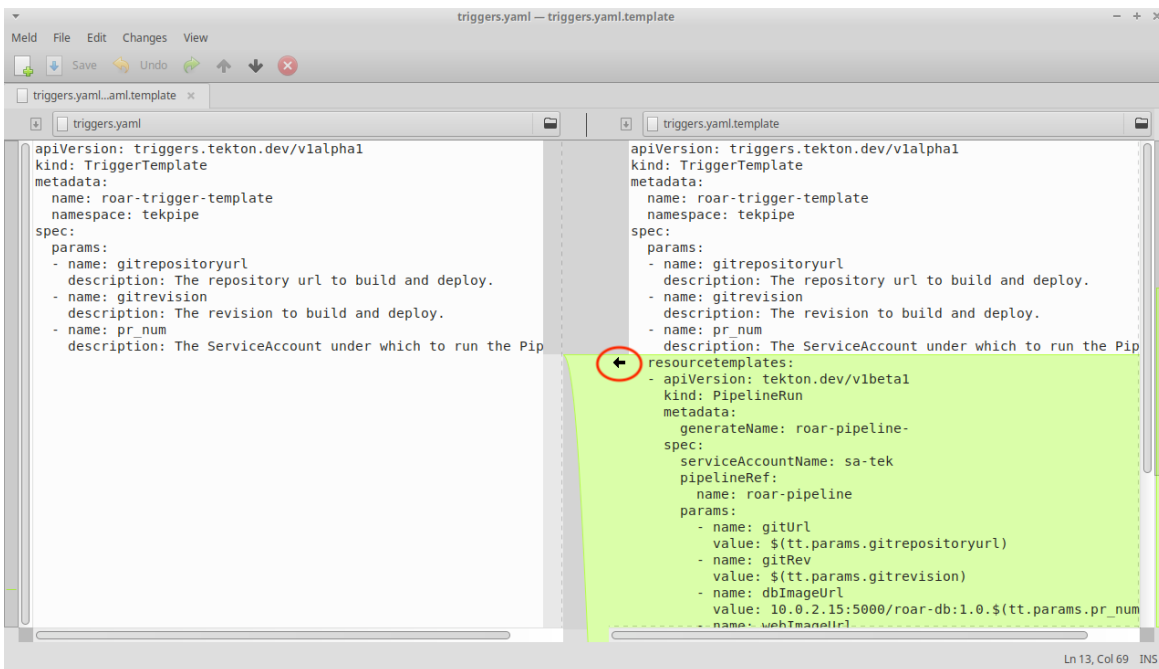
Review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.
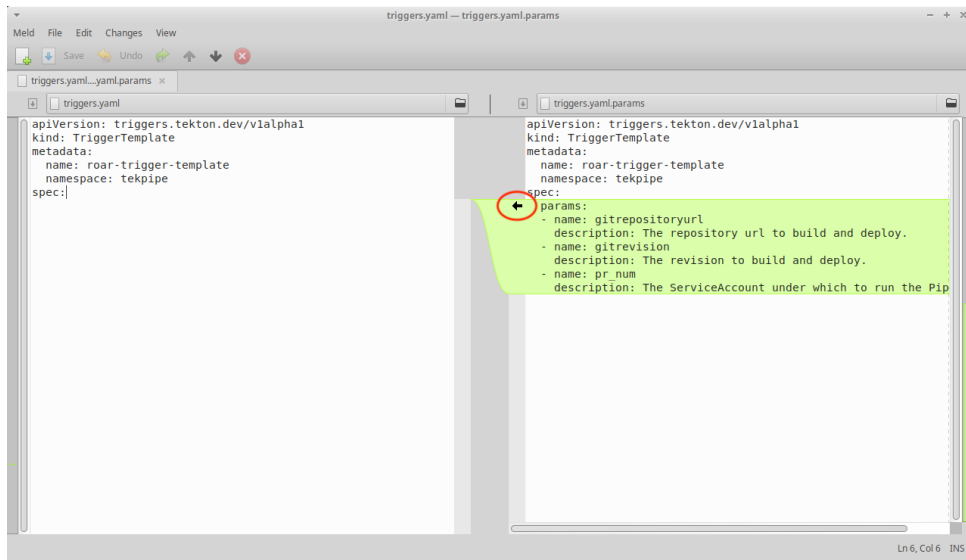
3. Next, we'll add in the template for the PipelineRun. When the trigger is activated and values passed into it, it will instantiate the template with the appropriate values and produce a PipelineRun object. This will then cause the Pipeline to run. Run the command below to add the PipelineRun template into the Triggers spec.

```
$ meld triggers.yaml triggers.yaml.template
```
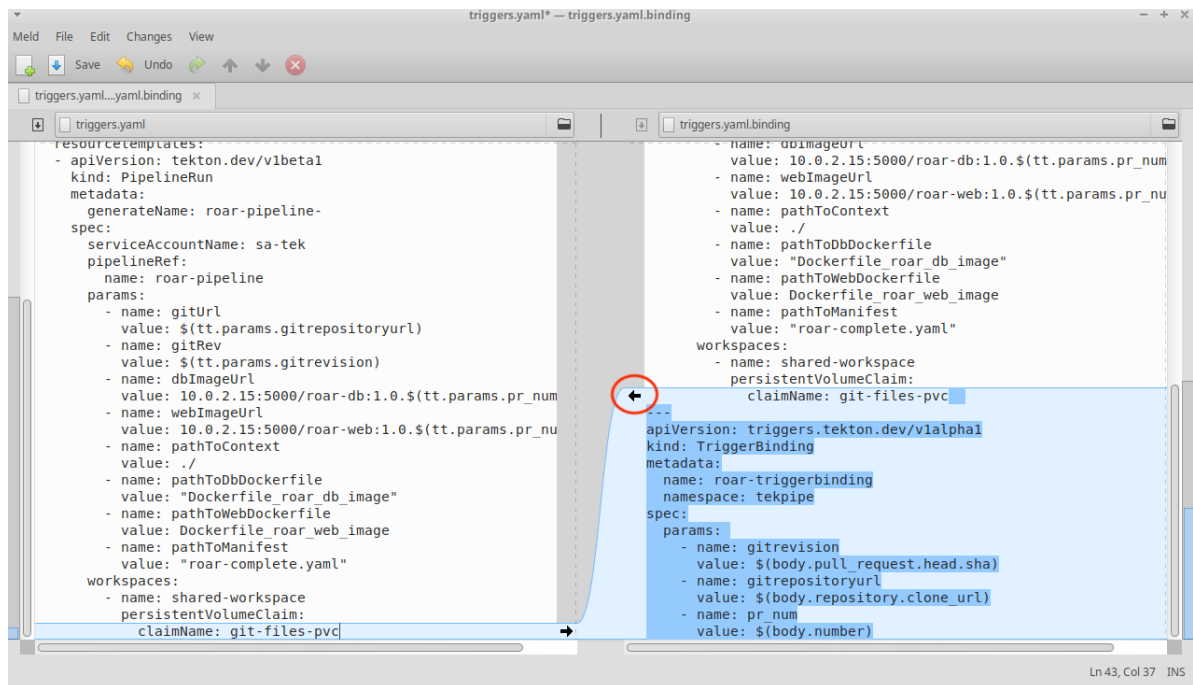
Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

4. Next, we'll add in the template for the Trigger Binding. This is used to bind values pulled from data from an event (such as a Pull Request) to parameters for the Trigger Template.

```
$ meld triggers.yaml triggers.yaml.binding
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.

5. Next, we'll add in the Event Listener. For this process, a service is started up that listens for particular events. When one is detected, the EventListener Trigger Binding. This is used to bind values pulled from data from an event (such as a Pull Request) to parameters for the Trigger Binding to eventually drive a PipelineRun object from the Trigger Template.

```
$ meld triggers.yaml triggers.yaml.el
```

Scroll down if needed. Then review the changes. Then click on the arrow circled in red in the figure. This will update the pipeline file with the change. Then Save your changes and exit meld.
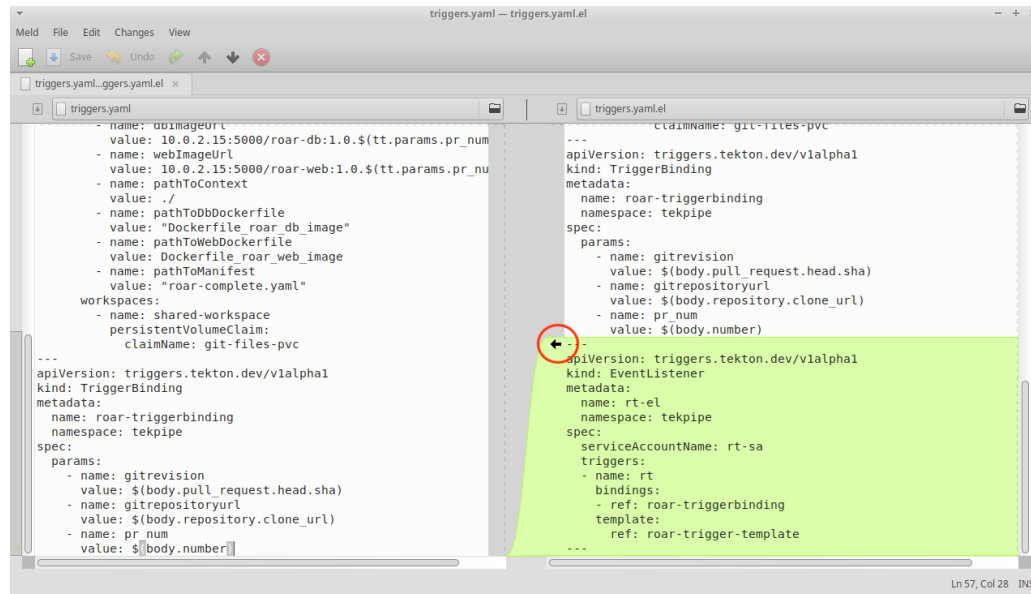
6. Now that we have the trigger pieces all setup in our spec. Let's go ahead and apply this into our namespace.

```
$ k apply -n tekpipe -f triggers.yaml
```

7. After a few moments, take a look at the services in the namespace. You'll see one there for the EventListener waiting for an event to kick off a PipelineRun - the CI process.

```
$ k get svc -n tekpipe
```

8. To be setup for the triggers to run, we need to have certain RBAC objects created. Take a look at the RBAC spec for the Trigger objects and then apply it into the cluster.

```
$ cat triggers-rbac.yaml
$ k apply -n tekpipe -f triggers-rbac.yaml
```

9. So now we are setup to pull data if a GitHub Pull Request sends a payload to us. Since we do not have a port open to the external web and GitHub, we'll need to simulate this instead. First, we'll do a simple port-forward on the service to the local machine. Open a new terminal session and run the command below:

```
$ k port-forward -n tekpipe svc/el-rt-el 8080
```

10. Next, we'll run a simple script in this directory that simulates sending the payload from a GitHub Pull Request. Run that script now.

```
$ ./pr-sim.sh
```

You should see output similar to the following and a new PipelineRun should be kicked off:

```
* Rebuilt URL to: http://localhost:8080/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.58.0
> Accept: */*
> X-GitHub-Event: pull_request
> X-Hub-Signature: sha1=ba0cdc263b3492a74b601d240c27efe81c4720cb
> Content-Type: application/json
> Content-Length: 193
>
* upload completely sent off: 193 out of 193 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json
< Date: Sat, 17 Jul 2021 17:13:38 GMT
< Content-Length: 155
<
{"eventListener":"rt-
el","namespace":"tekpipe","eventListenerUID":"4d143fbb-9c5b-45f6-939c-
d4b96a3f0e5f","eventID":"7489a6fb-32e4-4c71-9f46-359802a1a2cc"}
* Connection #0 to host localhost left intact
```

11. As before, you can see the various activities happening in the dashboard. And you can also see the pods being spun up in the tekpipe namespace. You can even see the containers running that correspond to the steps with the second command below.

```
$ k get pods -n tekpipe
```

```
$ kubectl get pods --all-namespaces -o=jsonpath='{range
  .items[*]}{"\n"}{.metadata.name}{":\t"}{range
  .spec.containers[*]}{.image}{", "}{end}{end}' | sort
```
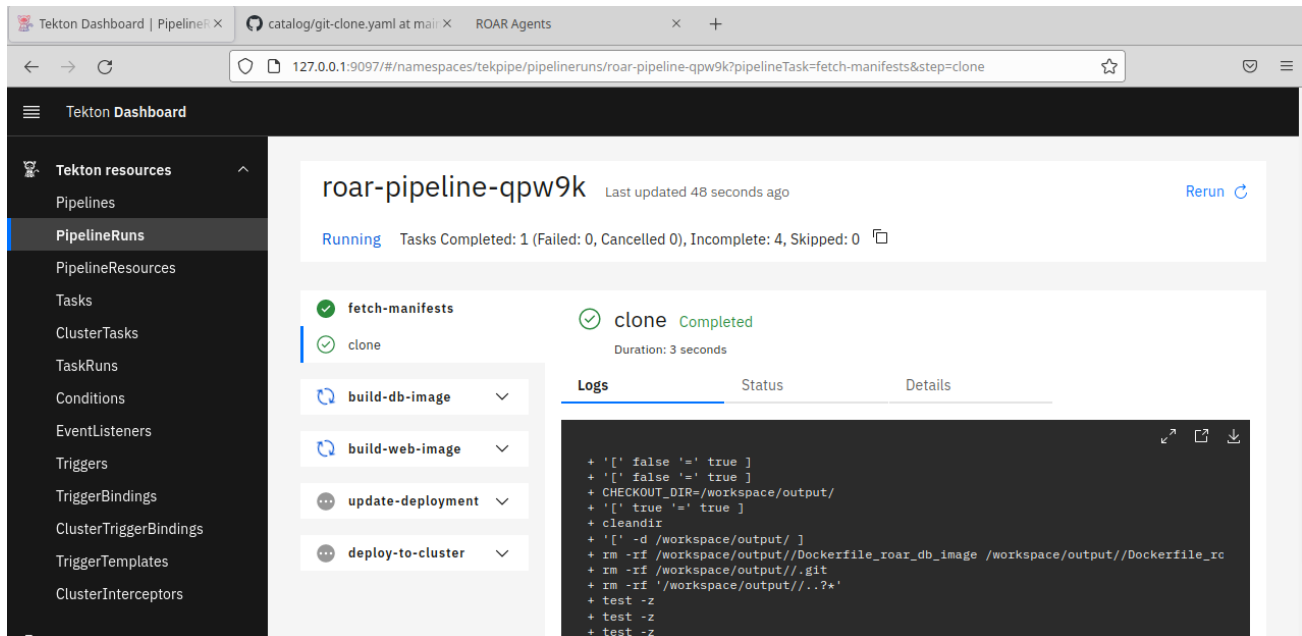
12. Once the PipelineRun completes, you can get the NodePort of the Service in the roar namespace and then open up a web browser to see the app that was deployed.

```
$ k get svc -n roar
```

Note the number after "8089:" in the PORT(S) column. Then open a web browser to

```
        http:localhost:<port number>/roar/
```

and you should see a running version of our app like below.

END OF LAB