Kubernetes cluster ad-build

Page about kubernetes clusters **ad-build** and **ad-build-dev**

Background information

1. Runs on hardware in s3df

```
Allocatable:
  cpu:                64
  ephemeral-storage:  152933498761
  hugepages-1Gi:      0
  hugepages-2Mi:      2816Mi
  memory:             259679512Ki
  pods:               220
System Info:
  Machine ID:                 92faa81e90af4e65ba73d3007e42519e
  System UUID:                ce9ba000-5727-11ed-8000-3cecefd8e38e
  Boot ID:                    96386228-b4ab-4836-b764-b22d4dfc0cda
  Kernel Version:             4.18.0-372.32.1.el8_6.x86_64
  OS Image:                   Red Hat Enterprise Linux 8.6 (Ootpa)
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.6.31
  Kubelet Version:            v1.28.8
  Kube-Proxy Version:         v1.28.8
```

2. **ad-build-dev** is used for build system development, while **ad-build** is the production build system for users.

How to access

ad-build-dev cluster: https://k8s.slac.stanford.edu/ad-build-dev

ad-build cluster: https://k8s.slac.stanford.edu/ad-build

1. After following the commands in those links, you will need to request access to the cluster from Claudio.
2. Recommended to install k9s tool https://k9scli.io/topics/install/

Other Notes

1. Kubernetes cluster is intended to be just for build system.
2. Can use local machine or nodes in the kubernetes cluster to create docker images (Don't need access to s3df/afs filesystem if all modules /dependencies are uploaded to GitHub, as they should be)
3. Docker can be ran on local machine, but not on s3df, it is intended to use apptainer instead, if build enviornment wants to be passed around. So when this build system is finished, developers/users won't use docker directly, instead will use apptainer and pull docker images from artifact storage (if needed)

Current status

1. We have a self-hosted runner image built: pnispero/gh-runner-image - Docker Image | Docker Hub (Temporary location)
2. how to deploy cbs on ad-build-dev cluster
    a. kubectl create configmap env-config-map --from-file=env-config-map.yaml
    b. last step: kubectl apply -f deployment.yaml
    c. Solution: a vault operator will be in place and we won't need these secrets

Current Tasks

1. **work on cli** - refer to CLI Tool
2. backend proposal?

a. Figure out what we need to pass in from the runner to the backend to start the build container and how this system can handle large number of requests
b. Possible things we need to start the build container
    i. repo name
    ii. organization
    iii. branch
    iv. user
    v. filepath to where the repo is checked out
c. Think about where we are going to store the builds, where the build container is doing the builds, we may want to seperate by user like /sdf/group/ad/user1/component_name/branch
    i. make script that maybe the runners can invoke (to pass in the vars)
d. Then we also want to think about where to keep the 'scripts' for building somewhere on s3df, so we don't have to bake the scripts into the images. maybe /sdf/group/ad/eed/ad-build/build-scripts/
    i. Possible scripts we need (this can be one or multiple scripts):
    ii. script to start the make or buildInstructions and enter the right filepath
    iii. script to log what part of the build system workflow are we in (Used for build system developers to debug failed builds due to build system)
e. Create a deployment file for the build containers (may be minimal)
    i. needs the volume mounted
    ii. the image
    iii. the name of the container (repo name + branch + unique id (automatically made))
f. jerry is working on trying to get x86/amd64 architecture to build not arm, then saw on s3df there are hundreds of packages installed, we may or may not want to install a good amount of them onto our containers.

3. **Build Workflow example**
    i. Build System Backend Flow - LCLSControls - SLAC Confluence (stanford.edu)
4. Create a simple hello world project,
    a. DONE Add to Jira - we can use this as the test project, upload it to github, add it to the component db
    b. try this with the mongodb for us to view - https://www.mongodb.com/products/tools/compass
    c. create a basic build environment with a 'build.sh' script copied over. Push this image, and add its url to hello world component. This build environment can be used by both developers and the build system.
    d. DONE **-** use the basic.yml workflow from the BuildSystem/ (which will be the workflow all 400 something repos under ad will have)
        i. See we can move the workflow to BuildSystem repo and all other repos can call it from there, like 'actions/checkout' can do like 'adbuild/build' this way any updates we roll out any repo can easily receive it
    e. the workflow should eventually do a GET request to the db, to get the build environment image,
    f. then spin up that new build environment image, 3 options to get the actual repo itself onto the new image
        i. (ideal) Runner does an actions/checkout onto a certain directory on s3df which will be accessible to all pods
        ii. Runner does an actions/checkout and passes in the actual repo through a 'kubectl cp',
        iii. or probably pass in the url to the repo which it can then clone - issue with git authorization
    g. Then signal to the new environment container to do a 'make', then we have 3 options
        i. Have the environment container copy over a 'build.sh' script which does not get invoked when the container is spun up, but when it is explicitly called with 'kubectl exec'
        ii. when you do 'kubectl exec' it can wait until the make is finished, then report back to actions that the build finished
        iii. it can signal the make, but report back to actions that the build continued at a certain container.
    h. CAVEATS: we can assume that if there are no additional instructions on the component entry in the db, then we just do a vanilla make. (Which is what most apps here do to build, at least for the iocs its true)
    i. Then we also want the simple project packed up as a package (src code and executable).
    j. Make sure we get the build output available to users, either to github actions, or point them to the container output (we may make a cli command for it?)
5. **Figure out the authentication automation for the runners.** (at the moment i get the blob of config from https://k8s.slac.stanford.edu/ad-build-dev: new solution: we are going to have only a couple 'orchestrator' containers that will do the kubectl commands, so only those need to be authenticated)
6. Get the build system container running on the kluster Deploying Self-Hosted GitHub Actions Runners with Docker | TestDriven.io (Altered to fit our situation)
    a. Lets do it vanilla first (running build system container)
        i. Create the image using base image: Package actions-runner (github.com)
            1. push the docker image to a registry so anyone can pull it
                a. From where the dockerfile is
                b. 'docker build --tag pnispero/gh-runner-image:latest .'
                c. This step may change (make a docker account, then create a access token, which will allow you to login on your shell)
                d. 'docker push pnispero/gh-runner-image:latest'
                e. Output: pnispero/gh-runner-image - Docker Image | Docker Hub
            2. Dockerfile (Here temporarily, these are the only 2 files you need to get this to work)

```
# base
FROM ubuntu:22.04

# set the github runner version
ARG RUNNER_VERSION="2.316.0"

# update the base packages and add a non-sudo user
RUN apt-get update -y && apt-get upgrade -y && useradd -m docker

# install python and the packages the your code depends on along with jq so we can
parse JSON
# add additional packages as necessary
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
    curl jq build-essential libssl-dev libffi-dev python3 python3-venv python3-dev
python3-pip

# cd into the user directory, download and unzip the github actions runner
RUN cd /home/docker && mkdir actions-runner && cd actions-runner \
    && curl -O -L https://github.com/actions/runner/releases/download
/v${RUNNER_VERSION}/actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz \
    && tar xzf ./actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz

# install some additional dependencies
RUN chown -R docker ~docker && /home/docker/actions-runner/bin/installdependencies.sh

# copy over the start.sh script
COPY start.sh start.sh

# make the script executable
RUN chmod +x start.sh

# since the config and run script for actions are not allowed to be run by root,
# set the user to "docker" so all subsequent commands are run as the docker user
USER docker

# set the entrypoint to the start.sh script
ENTRYPOINT ["./start.sh"]
```

start.sh

```
#!/bin/bash

ORGANIZATION=$ORGANIZATION
ACCESS_TOKEN=$ACCESS_TOKEN

# Generate organization registration token
REG_TOKEN=$(curl -L \
  -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer ${ACCESS_TOKEN}" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/orgs/${ORGANIZATION}/actions/runners/registration-token |
jq .token --raw-output)

cd /home/docker/actions-runner

./config.sh --url https://github.com/${ORGANIZATION} --token ${REG_TOKEN}

cleanup() {
    echo "Removing runner..."
    ./config.sh remove --unattended --token ${REG_TOKEN}
}

trap 'cleanup; exit 130' INT
trap 'cleanup; exit 143' TERM

./run.sh & wait $!
```

ii. do 'docker image ls' to ensure its there
iii. Then you must be an organization administrator, and make a personal access token with the "admin:org" and "repo" scope to create a registration token for an organization (REST API endpoints for self-hosted runners - GitHub Docs)
iv. Copy the token, and use it in the next step
v. Run the docker image

```
docker run \
  --env ORGANIZATION=<ORG> \
  --env ACCESS_TOKEN=<PERSONAL-TOKEN> \
  --name runner1 \
  runner-image
```

Replace <ORG> with the organization name
Replace <PERSONAL-TOKEN> with the token you created above

vi. And now your runner should be registered and running
vii. When done testing make sure to 'ctrl+c' and 'stop' and 'remove' the container

b. Start the image using kubectl for our ad-build kubernetes cluster you created above

i.
```
# Start the image with environment variables
kubectl run gh-runner1 --image=pnispero/gh-runner-image --env="ORGANIZATION=<ORG>" --env="
ACCESS_TOKEN=<PERSONAL-TOKEN>"
```

Replace <ORG> with the organization name
Replace <PERSONAL-TOKEN> with the token you created above

c. REMEMBER IF STOPPING THE CONTAINER, give it a grace period so it has some time to remove itself and from the organization

```
kubectl delete --grace-period=15 pod gh-runner1
```

Sample request - but refer to the api docs (https://accel-webapp-dev.slac.stanford.edu/api-doc/?urls.primaryName=Core%20Build%20System)

```
# gets component list
curl -X 'GET' \
  'https://accel-webapp-dev.slac.stanford.edu/api/cbs/v1/component' \
  -H 'accept: application/json'
```

Other Basic

Deployment of an image (running container) ex: Using kubectl to Create a Deployment | Kubernetes

```
pnispero@PC100942:~$ kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-
bootcamp:v1
deployment.apps/kubernetes-bootcamp created
pnispero@PC100942:~$ kubectl get deployments
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp   1/1     1            1           6s
pnispero@PC100942:~$ kubectl delete deployment kubernetes-bootcamp
deployment.apps "kubernetes-bootcamp" deleted
pnispero@PC100942:~$ kubectl get deployments
No resources found in default namespace.
pnispero@PC100942:~$
```