Runtime Auto Deploy (RAD)

CS263 Runtime Systems Project
Aarti Jivrajani
Daniel Shu

Presentation Overview

Presentation hosted here

- Motivation - What is Kubernetes and Containers? - Introduction to RAD - Layers of Implementation - Components of the pipeline - Status Profiler - Evaluation

Motivation

Cloud has become one of the most popular go-to deployment models - both in academia and industry - primarily because of the flexibility and ease it provides to the developers. Containerized applications are also very appealing because they can freeze the dependencies and maintain consistent runtime environments. But it is also true that deploying an application to the cloud should be just that - deploying that application. Users shouldn't be forced to bootstrap a cloud instance from scratch and also deploy Kubernetes; this would simply deter productivity and defeat the purpose of using the cloud. Bootstrapping, installation, and spinning off new Kubernetes nodes on a cluster is the main problem we attempt to address.

Overview

Our project is called Runtime Auto Deploy (RAD), and the goal is to automate the process of spinning off new Kubernetes clusters on-demand and deploying dockerized applications to Kubernetes on cloud(we used AWS for this project). Our goal is to ensure that the user has to provide minimalistic configuration data and the rest is taken care of by the RAD pipeline. Based on a user-supplied configuration file, RAD bootstraps an environment in the cloud and deploys the user's code.

Architecture and Implementation

Deployment tools

Before designing the RAD pipeline, we wanted to ensure that we chose tools/frameworks which, once set up, required minimum user modification. We explored multiple methods of deploying Kubernetes for our management cluster like - deploying Kubernetes the native way and using minikube. The former method was hard to debug by the user if something went wrong and it required us to install a bunch of dependencies which after trying once, we realized was not very user friendly. We needed a quick way to deploy the smallest Kubernetes node whose only purpose would be to spin off more clusters on demand.

Kind

Kind is a tool for running local Kubernetes clusters using Docker container "nodes". This seemed to be the perfect choice since deploying Kubernetes with kind is straightforward and it

is being actively developed. Active development and support are crucial when it comes to a community that is as large as Kubernetes.

Cluster-API - Cluster Lifecycle management tool

While searching for tools that would allow us to easily manage Kubernetes cluster lifecycle, Cluster-API seemed like the best fit. Cluster-API is a Kubernetes project to bring declarative, Kubernetes-style APIs to cluster creation, configuration, and management. It provides optional, additive functionality on top of core Kubernetes to manage the lifecycle of a Kubernetes cluster.

User Interface

Since the RAD pipeline is user-triggered, we attempted to design a simple API. We also provide a status API that allows a user to poll the service and check the status of the pipeline.

Trigger API (/trigger)

This is a POST call with the request body having the user GitHub repository. This API triggers the pipeline in the background and immediately returns a Trace ID unique to the user request.
We impose the following 2 rules on the structure of this GitHub repository:

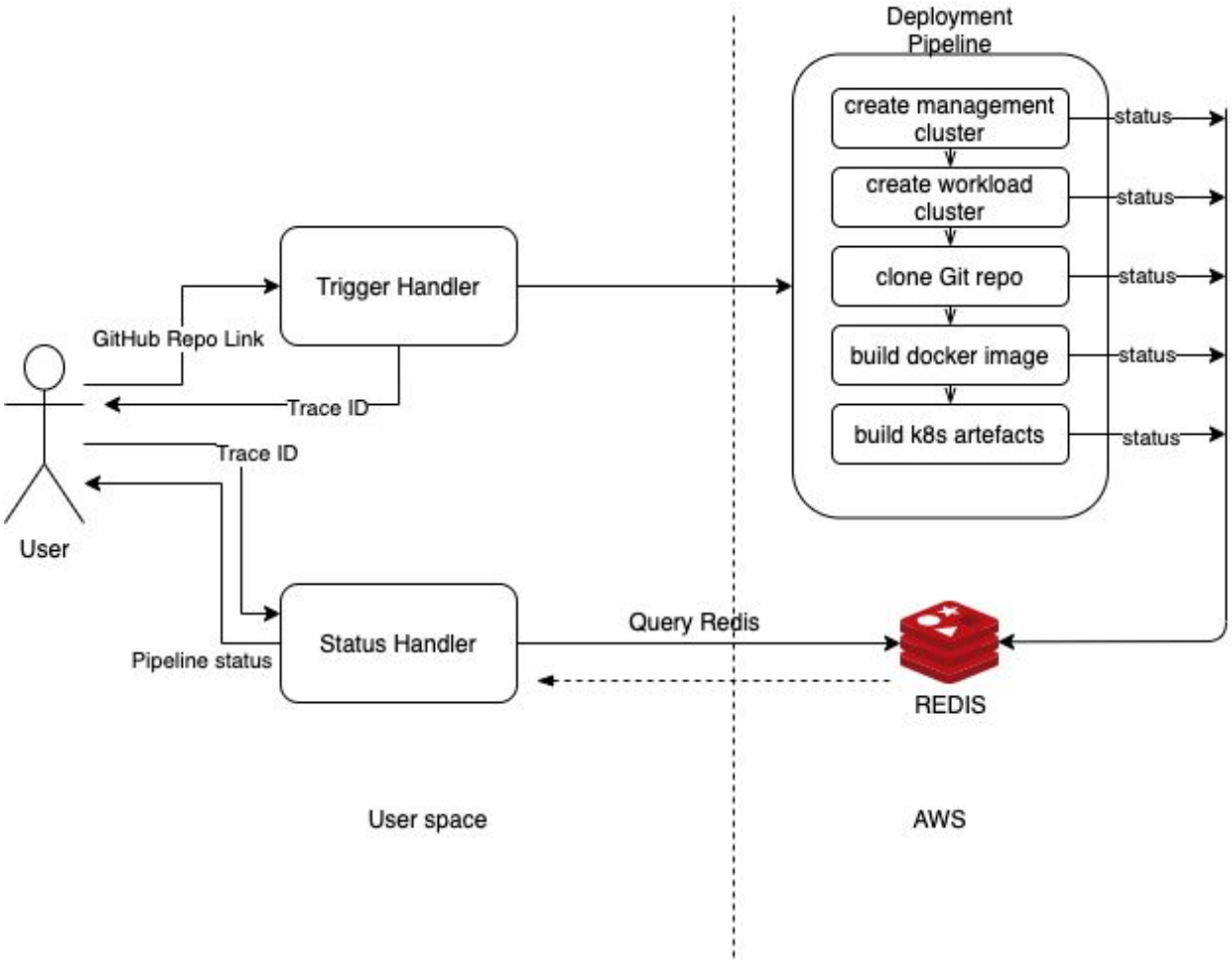
1. There must be at least one Dockerfile in this repository with the name Dockerfile*.
2. There must be a config.json file which has the following 4 fields:
    a. Name of the application
    b. Number of replicas that have to be deployed
    c. Dockerfile of the corresponding application
    d. User's docker registry details or the user can log in in the shell before invoking the pipelines

The trigger API does the following - clone the user GitHub repo, check if the above-mentioned rules are followed by the repository structure, build the docker image, push it to the Docker registry, create Kubernetes deployment and service and finally run the deployment pods. At each of these stages, the timestamp and the status of the stage are stored in Redis.

Status API (/status?traceId=<traceId>)

This is a GET call with traceId of the currently executing pipeline as the query parameter. This API returns the current status of the various stages of the deployment pipeline. This includes phases like "IN PROGRESS", "ERROR", "COMPLETED". Once the pipeline runs, the total time taken to run the pipeline is also returned.

Stages of Deployment



Create AWS instance

For this project, we chose to deploy Kubernetes on AWS. Although, this project could very easily be extended to deploy Kubernetes on GCE, Azure, and other cloud providers which are supported by cluster-API.

Create Kubernetes Management Cluster [script]

We use cluster API in order to spin off a new Kubernetes cluster and deploy the user application. In order to do so, cluster API requires a base Kubernetes cluster. This is called the “Kubernetes management cluster”. In this stage, the freshly created AWS instance is bootstrapped with all the essential libraries and tools - *kubectl(v1.18.0)*, *kind(v0.8.0)*, *docker engine*, *clusterctl(v0.3.5)*, *clusterawsadm(v0.5.3)*. We also initialize this Kubernetes cluster to behave as cluster API’s management cluster.

Create Kubernetes Workload Cluster [script]

Before the Kubernetes workload cluster is created, a bunch of “ceremonial” tasks has to be performed. These include making cluster-API aware of the cloud provider being used and plugging in the authentication credentials of the cloud instance(AWS access and secret key in this case). After this, clusterawsadm and clusterctl are used to bootstrap this new cluster. Once

ready, a CNI(Container network interface) solution is deployed which sits at the heart of Kubernetes networking.

Bootstrap cloud instance with dependencies [script]

Since RAD's scripts are written in Go, the Go runtime environment and its dependencies first need to be installed on the cloud instance. REDIS is used as a simple key-value store to bookkeep the status of each of the HTTP trigger requests. REDIS is also installed and started in this phase.

Run RAD HTTP Handlers

Finally, the RAD HTTP handlers are run. These are concurrent goroutines running on the bare metal VM. When the trigger is invoked, we perform the following steps:

Clone and Validate Git Repository

The user sends the Git Repo link as part of the POST request body. This repository is cloned and a file named "config.json" is searched for. If this file is not found, the process is terminated. This config file is required to have details of one or more applications that the user wishes to deploy. For each application, the name of the application, the name of its Dockerfile, and the docker repository details are required.

Create Docker Image

Using the Dockerfile details provided by the user, a new docker image is created. This image is created without re-using cached docker layers already present on the machine. This is to ensure that even the most generic layers like OS images are not shared between any of the applications. Lastly, this docker image is pushed to Dockerhub(to the user-provided docker repository).

Create Kubernetes Artefacts

Based on the application name and the docker image created, the following 2 kubernetes artifacts are created on the fly-

1. Kubernetes Deployment(formerly known as a pod): A Kubernetes deployment is the smallest unit of deployment in Kubernetes. Currently, we only support one container per deployment. This can very easily be extended by asking the user to provide a nested json in the configuration file.
2. Kubernetes Service: A Kubernetes service essentially allows the Kubernetes deployments to communicate with each other. It is a high-level abstraction of the underlying networking of Kubernetes.

Runtime Experiments

The time for our rad_workload_cluster.sh is 1740 seconds. The time is constant because we inserted sleeps to ensure that every process finishes running within the script. The sleep times are heuristic and give more than enough time for everything to finish.

To test the throughput of our application, we wrote a python script that will send 20 requests and then keep pinging the status endpoint until it reports that it finished. We created a basic application that publishes messages and subscribes to topics to a NATS server for testing

purposes. This application contains two images that need to be built of size 13MB and 16.4MB. We timed the process from the moment that request was received to when the pod started being built. We did not include the time between the pod being deployed and being ready for use.
We ran 20 trails and achieved the following times in seconds:

> 44.0, 58.0, 85.0, 75.0, 92.0, 71.0, 112.0, 51.0, 84.0, 52.0, 101.0, 41.0, 67.0, 68.0, 63.0, 71.0, 96.0, 61.0, 61.0, 81.0

On average, the pipeline from downloading the repo, building the docker images, and setting up the deployments took 71.7 seconds.

We then tried timing the throughput to deploy a 1.06 GB image. For the sake of time, we only ran 10 trials and achieved the following times in seconds:

> 687.0, 621.0, 748.0, 738.0, 735.0, 1464.0, 1749.0, 1352.0, 1503.0, 1395.0

On average, the pipeline took 1025.7 seconds.

You will notice that the second five trials took much longer than the first five. They were run on separate days, but all on my local computer. Our best guess for this discretion is that depending on how the machine is using its resources, there could be a very large difference. This wide range shows how hard it is to accurately estimate the throughput. The speed would also depend upon the speed of the user's internet because the repositories need to be cloned from Github, and the docker images need to be loaded to Dockerhub. For the smaller repo, the times were more consistent with each other because less data pulled and pushed, but as the size increases, the upload and download times could become more volatile. If the network is particularly busy, then the results could be even slower.

Scope

We've designed the required user configuration in such a way that it is easily extensible to deploy other Kubernetes artifacts like jobs, daemon sets, and stateful sets.
In the current implementation, we don't automate the creation of AWS instances, since a lot of security considerations have to be taken into account. But with cloud providers enhancing the IAM rules and policies, a combination of creating a user account and setting the right IAM roles can allow us(the pipeline owners) to get limited access to the user's clusters to perform the Kubernetes deployment operations.

Steps to Build and Deploy the Project

Here's a wiki that outlines all the steps required to build and deploy the project.
(https://github.com/shudaniel/RuntimeAutoDeploy/wiki/Steps-to-launch-a-new-cluster-using-Cluster-API)

Source Code

(https://github.com/shudaniel/RuntimeAutoDeploy)