Elevating CI/CD with Containers and Embracing Serverless

Vamsi Krishna Yepuri
Dept. of Computer Science
Kent State University
vyepuri@kent.edu

Venkata Kalyan Polamarasetty
Dept. of Computer Science
Kent State University
vpolamar@kent.edu

Henry Moye
Dept. of Computer Science
Kent State University
hmoye@kent.edu

***Abstract*—In today's software world, we're moving from large monolithic application towards using small, independent software parts called microservices. These microservices are packed into containers and run on a cloud native system like Docker, and Kubernetes. However, managing lots of these microservices can be tricky. Manual deployment of these applications takes more time and limits how quickly we can deliver our applications. Existing CI/CD (Continuous Integration/Continuous Deployment) systems often use traditional tools like Jenkins, which run on virtual machines, making things even more complicated. With the number of modular applications are growing up, we end up needing more and more build servers to handle different requirements for each microservice. So, our project takes a different approach. We have implemented the build process in a volatile container as serverless model instead of using multiple build servers. We used modern practices for CI using cloud-native CI with Tekton and GitOps model for CD with ArgoCD. The idea is to make the most out of these cloud-native methods and also showcase a more efficient way to build software without relying on lots of servers. To demonstrate, we have considered a bookstore application – a polyglot microservice application and created pipelines using Tekton and ArgoCD to deploy microservice applications in a cloud-native environment faster and smoother.**

***Index Terms*—Docker; Kubernetes; Containers; Polyglot; DevOps; GitOps; CI/CD; ArgoCD**

1 Introduction

In software development, a monolithic application or system is one where all the components and functions are tightly integrated into a single codebase. This means that the entire application is built as a single unit, and it can be more challenging to update or modify individual parts without affecting the whole system. In contrast, microservices is an architectural approach used in software development to build applications as a collection of small, independent, and loosely coupled services. Because of the modularity, scalability, flexibility, cloud native and easier maintenance, there is shift from the monolithic to microservice based applications. Indeed, there has been a significant ruse in microservices applications in the recent years.

The modularity and independent microservice architecture have paved a way for polyglot microservices, where each individual microservice is developed using different programming languages, frameworks and technologies that are best suited for that particular functionality. However, microservice also come with challenges such as load balancing, service discovery, communication and ensuring data consistency in a distributed system.

In recent years, the rise of containerization has brought about a significant transformation in the world of software development. Containers make the application packaging, deploying in an easier manner. They are very lightweight when compared to virtual machines and have a standardized packaging format where we encapsulate all the application and its dependencies. This packaged format is called container image. When we run this container image, our application will be running inside the container. In virtual machines, sometimes our applications run well in development environment and fails to run in production environments. This is because of inconsistency in the dependencies of the running servers. In case of containers, we can run the application consistently in different environments from development to production. Ziyu Li [1] done a performance comparison of docker container and a virtual machine and finds that the container had demonstrated the superior performance to virtual machine in terms of CPU, hard disk and memory. Leveraging the advantages of containers, microservice applications are also containerized for efficient way of running them. However, containerization itself doesn't solve the microservice based application challenges. To address the challenges of microservice based applications, we have used Kubernetes, a powerful container orchestrator. Kubernetes is installed as a cluster with multiple servers and it runs the multiple instances of our microservices as containers in distributed manner. It tackles all the microservices challenges. It distributes the incoming load to our distributed microservice containers. It also takes care of service discovery and communication between the microservices and also makes the data consistent across all the microservices.

Manual deployments of containerized microservice applications to Kubernetes takes the lot of time and makes our delivery late. In order to reduce the manual deployment, there should be an automated way of deploying the application which takes less time. The automated way of build the application form the code is called continuous integration and the automated way of deploying the application is called continuous delivery. When the developer changes the code, this automated CI/CD pipelines will pull the latest code, build the application to a deployable artifact (container image) and deploy it to the Kubernetes. There are

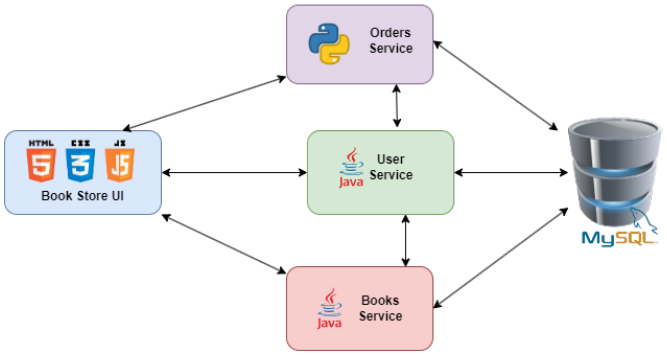

Fig. 1: Bookstore microservices application architecture

different tools through which we can makes our deployments in an automated way. One such popular tool used in the industry is Jenkins. There are different CI/CD solutions provided by the public cloud providers as well. In case of Jenkins, we can install it on a server and run all the build script for building and deploying our applications. For example, if you want to build an application that requires system level library with version v1 and in the same server if you want to build another application that requires the same system level library with version v2, then it's not possible to run both the applications in the same server. In this case Jenkins uses a build agent, where you can create a different server with v2 dependencies and you can use that server building the second application. But in an organization, you are not limited to build one or two applications. With the growth of microservices, the number of applications keeps on growing and assume how many conflicts of dependencies may raise when you have polyglot microservices application with each microservice having its own unique dependency. We need to keep on adding the build servers and lot of maintenance and operations are required for these build servers. This costs a lot to the organization if the build servers were left idle even there was no build happening in the build servers. Starting and stopping build servers only when required is also an additional manual task and reduces the agility of delivering the applications.

In order to overcome the challenges of the server-based CI/CD model, the build servers should be dynamically provisioned, i.e. the severs should run only at the required time instead of all the time. This way of compute is called serverless model. In this project, we employed a framework called Tekton to overcome this challenge. Another challenge here is we are having the CI servers installed on the virtual machines. When our target is to deploy our microservice application on Kubernetes, instead of building the applications on the virtual machines, we can build the same application in containers running on Kubernetes. This is called cloud native CI CD solution. Tekton is a cloud native CI solution with serverless behavior. We'll be using ArgoCD which uses a pull-based mechanism to deploy the applications to Kubernetes. This pull-based mechanism uses git repository to store all its deployment manifests. So, we call using Argo CD as a GitOps model of deploying the application.

Our project objective is to utilize the most out of these cloud-native methods and deliver the applications in an efficient manner. In order to demonstrate this serverless way of implementing CI CD with Tekton and ArgoCD, we have bookstore application. As shown in the Fig. 1, the bookstore application is a polyglot microservices application with four different microservices developed in Java, Python and UI technologies. To deploy this application in Kubernetes, we will write the necessary pipelines with Tekton, ArgoCD and build and deploy these applications in a serverless manner.

The rest of the paper in organized as follows: The next section introduces the related work, project background, recent methodology and system architecture.

2 Literature Review

In the market, there are many CI/CD software solutions available. They are being used to automate the build and deployment process in software development life cycle. These CI/CD tools have wide range of features and capabilities to build and deploy the software products. We will be looking after different CI CD solutions available.

The most popular tool for CI/CD is Jenkins. A. Cepuc et al [2] used Jenkins to deploy the Java based application to Kubernetes. Here the source code is hosted on git repo and whenever the developer makes changes to the repository, the pipeline is trigged in Jenkins and builds the java application into a container image and pushes to Docker hub. This work uses Ansible playbook to deploy the image to Kubernetes. Jenkins is a popular CI/CD tool and works with the plugin ecosystem. Jenkins is completely server based and the build happens entirely in the build agent servers. The drawback here is we need to have multiple build servers in this approach if we have more microservice applications.

C. Singh et al [3] has presented a case study of comparing two popular CI/CD tools Jenkins and GitLab. GitLab can acts as a git repository and can also maintain the product related issues as well as it is a CI/CD solution that uses runners to run the applications. Same a Jenkins build server, GitLab requires runners running on the agent servers. The Gitlab server will issue the instructions to these agents. Based on the instructions received, the agent will perform the build process. While in Jenkins, all the pipeline requires to be run in a same build sever. In case of Gitlab, the pipeline as a code comes into picture, where the pipelines are divided into stages. Each stage is modular and can run each stage on different server. For example, we can run testing on one server and building on one server and deployment using another server. The performance comparison was done by considering deployment time as a metric. Gitlab performed so close in Jenkins when used a dedicated agent server to build, while there was a huge delay in build time when shared GitLab runner is used.

C. Aparo et al [4] used the GitLab pipelines to test the java application in a build pipelines. There are different stages in the GitLab pipeline that are used to test the java application. All Static analysis, vulnerability scanning and dynamic application security scanning for the java application is performed by the GitLab pipelines. Each kind of security test is a stage defined the pipeline yaml file. H. Teppan et al [5] mentioned in the survey that they combined GitLab CI/CD pipelines with ArgoCD to deliver the applications on Kubernetes. The author also stated that GitLab pipelines with ArgoCD is an alternative to use Tekton with ArgoCD, to run on Kubernetes. Here the author utilized containers to build each stage like test, build, push i.e., GitLab runners used a docker executor [6] where each stage will be performed in a dynamically provisioned container. This is very

close to what we are trying to achieve. This is same as the serverless CI/CD that we are trying to achieve, but the GitLab CI/CD is not running on Kubernetes and this is not Kubernetes native approach of CI/CD.

A. Decan et al [7] presented the overview of GitHub actions. GitHub actions aka GHA is released in 2019 and gained a huge popularity. This study emphasizes on how the GitHub actions are being used in the public repositories. GHA is great for opensource projects, because we no need to maintain and pay for a separate build server. The build server will be along with your code and can be utilized to build and deploy to the targets. The author presented the workflow file, where we mentioned the build steps in the form of jobs. Each job can be a build step and we can mention which container image to use to run the build. This also achieves the serverless CI/CD model which we are looking for. While executing the workflow/pipeline for each job, the mentioned container will spin up and your job will run inside that container. For example, test job should can on ubuntu container and the build job can be run on centos container. Based on our requirement, we can design our workflow. Even though this has serverless compute model for CI/CD, the major disadvantage here is its tight encapsulation with GitHub. GHA actions can only be used if you are using GitHub. If you are some other git hosted repositories like GitLab or Gogs, you cannot use the GHA for this. Moreover, the entire build process runs on the GitHub cloud and may not be suitable if you are looking for a private cloud or on prem solutions.

M Kushtov [8] has done a thesis on implementing the serverless CI/CD pipeline based on Google Cloud Platform. The author utilized Cloud Build a platform-based service offered by GCP. In order to deploy an application, the author uses a yaml file to run the pipeline. This pipeline consists of build steps, where we can mention the container image name where we need to run our scripts. This is similar to GitHub actions and GitLab runner with docker. Whenever we run a pipeline, each step will be run in a dynamically provisioned container. These containers are dynamically provisioned in Google Cloud servers. Here, it achieved the serverless capability, but this cannot be utilized on private cloud. There is a vendor lock in and these pipelines are not portable. If you want change from one cloud provider to another or even to a on premise platform, you have to refactor your pipelines. The same with different cloud providers like AWS and Azure. They have their own CI/CD solutions, but it can be good for that particular public cloud, and are not portable and cannot to run anywhere in future if we want to run.

S. Gupta et al [9] and Ramadoni et al [10] used GitOps based delivery model. They demonstrated using of ArgoCD for deployment which is GitOps model of deploying the applications. In CI/CD, CD stands for continuous deployment. In an example of deploying an image to Kubernetes, once the deployable container image is built, we will be utilizing the Kubernetes credentials to deploying the image to the cluster. Here this approach is a push-based approach. The build servers initiate the deployment to the Kubernetes server. Here there are two disadvantages with push-based deployment. First one, considering an example where we have deployed the latest version of the application using regular CI/CD push-based deployment. Assume if we have some bugs in the latest version and we have to roll back to the previous version. We have to rerun the CI/CD pipeline with old version to deploy the old image. Second, in terms of security, where some user login to the production server and changes the version of the application manually. There is no way to know who has done so, which version of application that we desired is running or not. In order to overcome these two disadvantages, GitOps model can be used. In GitOps model, git repository will be single source of truth. All the Kubernetes deployment manifests will be placed in the git repository and whatever is provided in the git repository, the same will be deployed on the Kubernetes cluster. If you have nginx:3.2 version docker image to run on your cluster, the Argo CD will pull these and runs the nginx 3.2 container. If someone manually tries to change the version of the container in Kubernetes, it will not be in sync with the git repository and can be easily monitored. The only way we can change the version of nginx is to change the deployment manifest in the git repository. This will be logged and can be easily used to trace, who have changed to version of the application. ArgoCD is one such operator that uses this pull based mechanism which always keeps the Kubernetes deployments in sync with git repository. This is good CD model, that is used in our implementation for deploying the manifests in Kubernetes. Even though this is not a complete CI/CD model, we can integrate this model with Tekton CI to fully achieve our desired objectives.

Y. Bello et al [11] uses Tekton framework for building and deploying the applications on OpenShift single node cluster. The author deployed SDP application using Tekton as both CI and CD and the comparison was done with manual installation time, script installation time and CI/CD installation time. From the analysis, the deployment using CI/CD takes no time when compared to manual and script installation. Tekton is opensource, Kubernetes native CI/CD tools which leverages dynamic provisioning of builds in the form of tasks. However, GitOps model is not utilized here for efficient CD. J. Mahboob et al [12] utilizes Tekton as their CI/CD tool in the implementation of DevSecOps model.

To conclude the literature review, there are landscape of tools that are continually evolving based on the needs. As organizations are moving towards microservices and deploying on Kubernetes ecosystems, the Kubernetes native CI/CD solutions like Tekton and ArgoCD looks more optimized way of implementing DevOps. The advantages like portability, open-source nature with no vendor lock in, git ops pull based mechanism for deployment, Kubernetes native, serverless way, easy rollbacks, secure, modular, and can be used on any type of public and private clouds makes us choose Tekton and ArgoCD as our CI/CD solution. In the next section in this paper, we will go through the practical implementation of our approach to deploy bookstore application with Tekton and ArgoCD on Kubernetes.

3 Project background

3-A Transition from Monolithic to Microservices Architecture

With the benefits that the microservices offer, the industry is shifting from monolithic application to microservice architecture. Microservices offer modularity by breaking the application into small independent components making it easy for developers and maintenance team. As the microservices are divided into small independent components, they offer flexibility of using diverse technologies with in one application. For example, consider a bookstore application which shown in Fig. (1), it is polyglot microservice application which consists of independent services that are written in different programming languages. Here on each independent microservice, developer can focus only on that

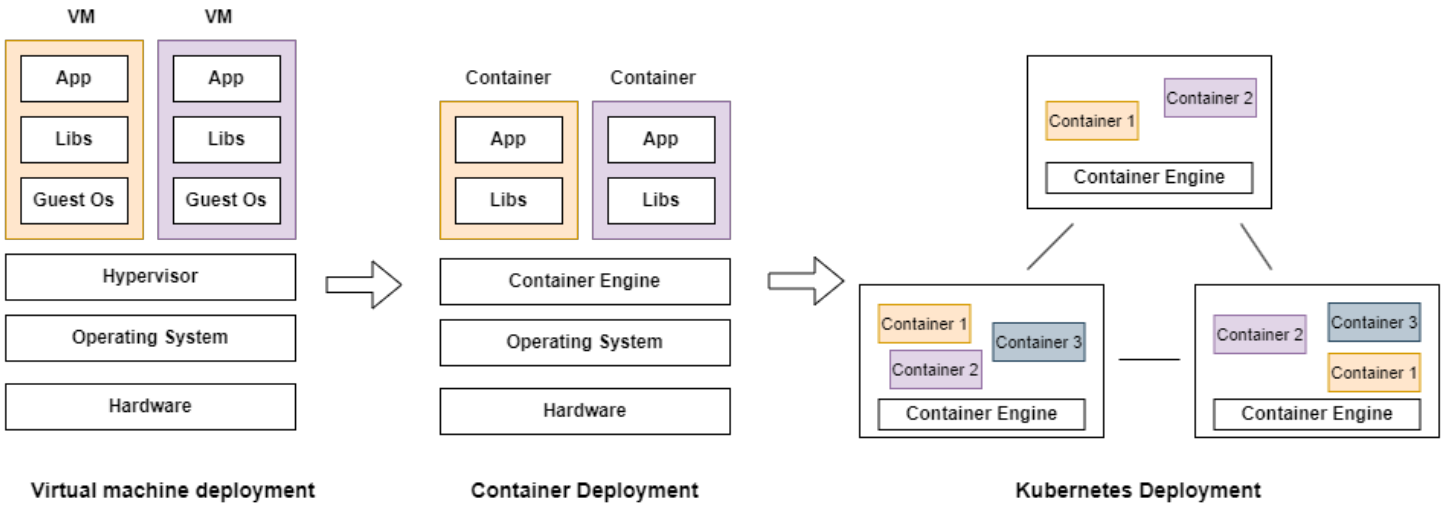

Fig. 2: VM vs Container vs Kubernetes Deployment

particular codebase instead of whole application codebase in the monolithic application. This makes it easier to maintain the code and also allows rapid deployment. These are also easy scalable with granular resource allocation that reduces cost for a company. In this way, easy maintenance, rapid development and deployment can be achieved by focusing on independent microservices rather than a monolithic codebase. This transition from monolithic to microservice enables organizations to stay more agile in this competitive world.

3-B Containerization and Kubernetes

OCI (Open Container Initiative) container is a technology that can be used to package your software application with all its dependencies, libraries and configuration files and allows you run the application with these dependencies in a standard isolated environment. To run a virtual machine, you require an operating system image, same way when running a container, you use a container image that has both the application and its necessary dependencies. These containers are portable and are consistent among different environments like development, staging and production. They can reduce errors that are caused by differences in application versions and dependencies across different environments. These containers are lightweight, efficient and share underlying OS kernel which makes them boot faster and utilize very less resources [13] when compared to virtual machines. As shown in Fig. (2), virtual machines run on a hypervisor and includes a complete operating system which makes them more resource intensive compared to containers. Even though VMs provide best isolation than the containers, there is an additional overhead with the resources that they use and slower boot times. Unlike VMs, containers share the host OS kernel and results in lightweight and efficient solution for packing and running application.

Kubernetes aka k8s is an opensource container orchestration tool developed by google. For high availability, we must deploy our application across multiple instances. For this we must manage the copies of these instances, which we call replicas. Then, we set up a load balancer to spread the traffic among the instances and for these multiple instances to communicate with each other, we need to establish service discovery system. Kubernetes simplifies the management of the container applications. When you have your application container images, you can use k8s for scaling the replicas, load balancing, service discovery, rolling updates and much more. As shown in Fig. (2), a group of machines are required to form a Kubernetes cluster. Kubernetes uses Software Defined Networking (SDN) for communication between the containers. Multiple containers (replicas) of same application can run of different machines for high availability and the communication and load balancing is taken care by k8s SDN. Kubernetes has become the widely accepted standard for managing containers, which simplifies the process of launching and managing modern container-based applications.

The combination of microservices, containers and Kubernetes one of the standard approaches in development and managing the applications. Microservices offer modular and independent deployable service while containers providing isolation, consistency. Kubernetes provides the orchestration of these containers by easy scaling, load balancing and service discovery in a distributed manner.

3-C Continuous Integration and Continuous Deployment (CI/CD)

Manually deploying containerized microservice applications to Kubernetes is a time-consuming process and also poses more challenges to developers and the operations teams. When a developer writes code and push it to source code repository, they often need to wait for operations teams to pull the code, build and deploy the application to servers. This often takes time and this delay in feedback slows down the development cycle, as the developers have to wait for the operational tasks to complete. In microservices-based applications, as the number of individual services increases, the manual deployments keep on increasing and may give rise to human error. Finally, it can reduce the agility of the application development process, making it less flexibility to changes and updates.

Continuous Integration (CI) and Continuous Deployment (CD) are fundamental practices in software development life cycle for more agility. CI involves automatically integrating code changes

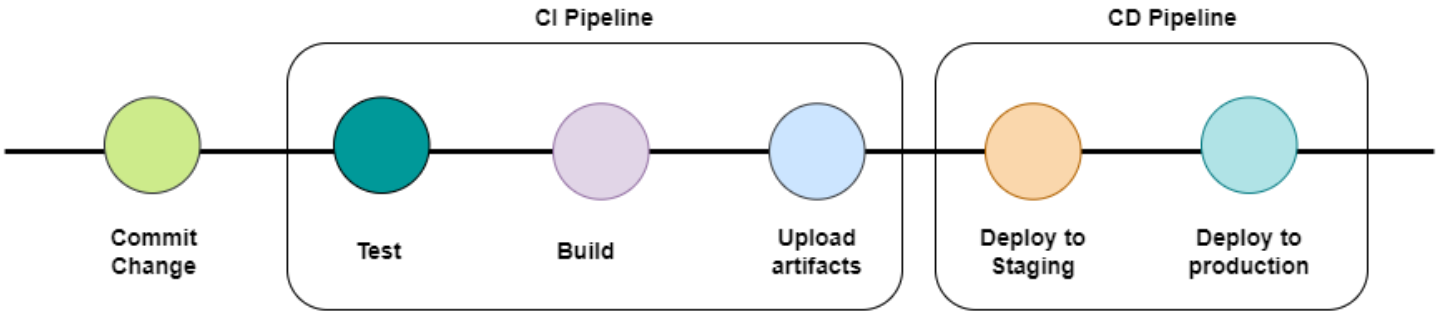

Fig. 3: CI CD pipeline

to deployable artifacts. CD extends this by automating the deployment of artifacts to various environments, including production. For polyglot microservice applications, CI/CD tools play a crucial role in automating the build and deployment processes across different environments. CI/CD not only accelerates the development process but also reduces the scope for human error, making it a vital component of efficient and reliable software delivery in the world of microservices. Fig. (3) shows an outline of a CI CD pipeline. When a developer commits code changes, it undergoes different stages in CI CD pipeline to deploy to desired environments.

3-D GitOps CD model

Usually, the deployment in a CD pipeline can take two ways. First one is following push-based approach and the second one is using pull based approach. As shown in Fig. (4), in this approach, usually CD pipeline get the Kubernetes deployment manifests from the git repository and uploads to Kubernetes to deploy the applications. This is a push-based mechanism. Triggers can also be added, whenever there is change in the deployment manifests in git repository, the CD pipeline will be triggered and this deploys the application to Kubernetes.

On the other hand, GitOps is pull based approach that is used to deploy the application to Kubernetes clusters. As the name speaks, git repository is the single source of truth here. All the deployable Kubernetes manifests will be stored in the single git repository. As shown in Fig. (5), whenever you make a change to the deployment manifest in the git repository, there is a GitOps operator that will be polls for changes in git repository. The changes that are made will be reflected in the Kubernetes. This GitOps operator will sync all the deployment files in the git repository to the Kubernetes cluster.

4 Recent Methodology

The main goal of this project is to make use of best Continuous Integration and Continuous Deployment practices to efficiently deploy a polyglot microservice application in our case we deploy bookstore application onto Kubernetes in a serverless fashion.

From our Literature Review Y. Bello et al [11] used the Tekton framework for building and deploying the application to OpenShift cluster which is a RedHat distribution of Kubernetes. This work demonstrates us the usage of CI/CD platforms in Kubernetes ecosystem. J. Mahboob et al [12] also used Tekton as their CI/CD tool to demonstrate the practical application of Tekton in CI/CD process. S. Gupta et al [9] and Ramadoni et al [10] used GitOps based delivery model using ArgoCD for deployment. This is a pull-based mechanism that keeps Kubernetes deployments in sync with a git repository and this solves the issues related to deployment security. We have chosen Tekton for CI and Argo CD for deployment. This choice is motivated by the advantages that are outlined in the literature review which includes portability, no vendor locks in, opensource nature, Kubernetes native with serverless capabilities and GitOps based deployment model. The combination of Tekton and ArgoCD can be the robust solution for implementing CI/CD pipelines in modern software development i.e., we can use this for microservice deployment in Kubernetes.

Fig. (6) shows the CI/CD flow diagram that can be implemented with Tekton and ArgoCD. When the developer changes the code, the code will be committed to the git repository. When this happens, the CI pipeline is triggered and the execution starts. You can see this execution takes place with the help of Tekton pipelines. The Tekton CI pipelines consist of individual tasks like cloning the code, building the container image from the application and pushing it to container image repository. Once this is done, the deployment manifests will be updated in the git repository. Till here, the CI pipeline is executed and now its turn for deployment (CD). As we use GitOps based delivery using ArgoCD, our Kubernetes cluster will be in sync with the git repository. As the CI pipeline built a new image and updated the manifests, ArgoCD will detect this change and will synchronize the resources as per the manifest's files in the git repository. All the image files required are pulled from the container image registry docker hub. This is the complete flow of building and deploying the application with Tekton and ArgoCD.

5 System Architecture

Fig. (1), shows the bookstore microservice application that we want to deploy to Kubernetes using Tekton and ArgoCD. This deployment includes three major steps, first one is integrating the git repository with the Tekton server. The second one is executing the Tekton pipelines which includes tasks like building, packaging and pushing the docker images and the third one is deploying with ArgoCD with which application deploys to Kubernetes when the CI pipelines creates a new deployment manifest. Fig. (7) shows the architecture diagram which includes these three steps.

In the first steps, we need to integrate git repositories and Tekton. In this step, Tekton event listeners are used to integrate. This event listener will have an external HTTP endpoint, which we can use to configure in the git repositories as a webhook. Whenever there is a change in the git, this event listener will be notified. When there is change and the event listener is notified,

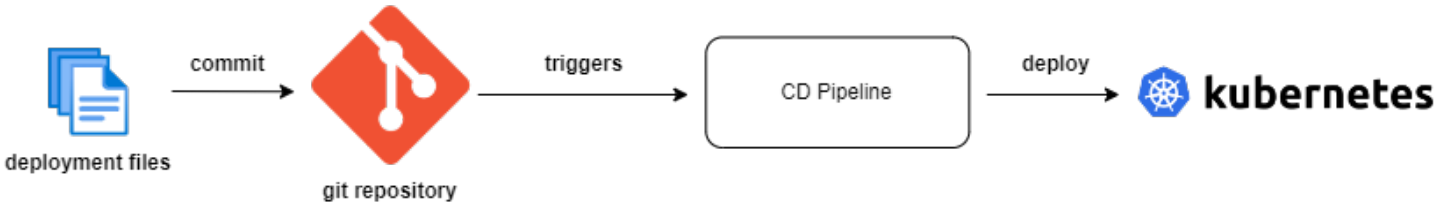

Fig. 4: Push based deployment

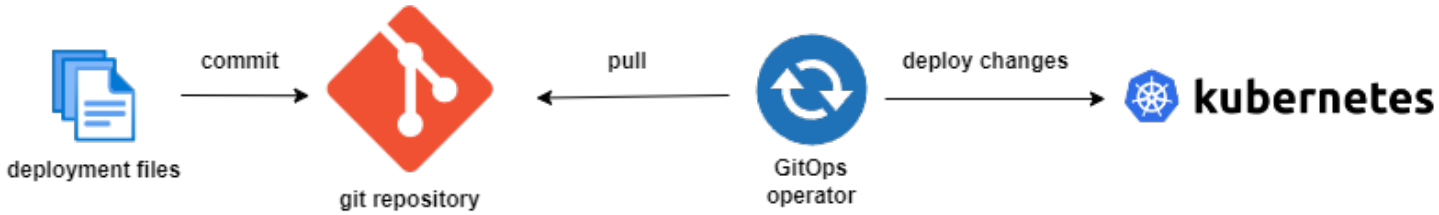

Fig. 5: Pull based deployment

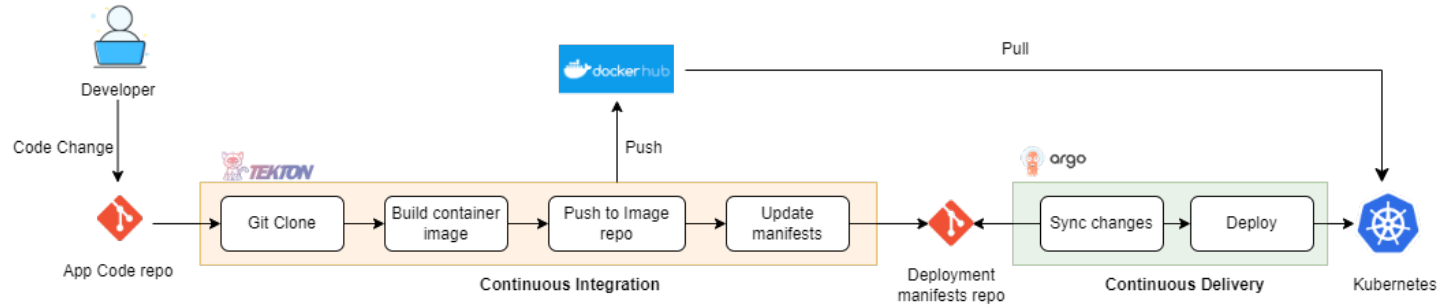

Fig. 6: CI/CD flow diagram with Tekton and ArgoCD

the trigger binding will fetch the data from the webhook like commit id or tag id etc. After trigger binding the trigger template are used to create a pipeline run. We define a pipeline and when we run it, we will get a pipeline run. Trigger template also does the same. Whenever there is change, after event listener gets notified, trigger template creates a pipeline run, where all our build steps will be executed.

The second step is executing the pipelines with Tekton. In these pipelines, we will execute different tasks like cloning the code, building the container image from the application and pushing it to container image repository as shown in Fig. (6). As our goal is to achieve this task execution in container as a serverless approach. Tekton is a serverless model where execution of these tasks happens in dynamically provisioned pods in Kubernetes. These are provisioned only when needed and are removed after completing thus achieving the serverless approach. This reduces the operational complexity, resource utilization and costs. Fig. (8) shows the Tekton pipeline which consists of tasks. When we run this pipeline, each task in this pipeline will be executed in an ephemeral pod. Task can contain a single task or multiple where each step in the task will be running as container in each pod. As these pods are ephemeral, they will be automatically destroyed after execution of the tasks. In this way we can achieve serverless way of implementing CI CD solution. As these pods are ephemeral, if a pod wants to save some state or data, it writes to Persistent Volumes and the other pod can read and modify this data. For instance, a git clone task clones the code to the persistent volume and this code will be used for the next build tasks to build to code.

The third step is deploying the applications using Argo CD. From Fig. (6),(7), we can see that after CI pipeline completes its execution, it generates a container image and puts this image in container registry like docker hub and this image reference is updated in the deployment yaml files which are located in git repository. This git repository is in sync with the Kubernetes. This synchronization is taken by Argo CD server. As there is new container image generated and the Tekton CI pipeline updated this image information in the deployment yaml files located in git repository, the ArgoCD will sync these changes in the Kubernetes cluster. In this way we can implement the Tekton pipelines for continuous integration and Argo CD for deployment for our bookstore application. Fig. (7) shows the git repositories for our bookstore microservice application and the pipelines are triggered when a developer makes changes to these code repositories. The deployment yaml files are stored in the git repository with which the Argo CD will synchronize to Kubernetes cluster.

6 Implementation

To implement CI/CD we need to choose an application as a first step. For this project we considered bookstore polyglot microservice application. In second step, we install Kubernetes on servers for our project. As a final step, we need to install all CI/CD and GitOps tools required for our project on Kubernetes and then configure required pipelines.

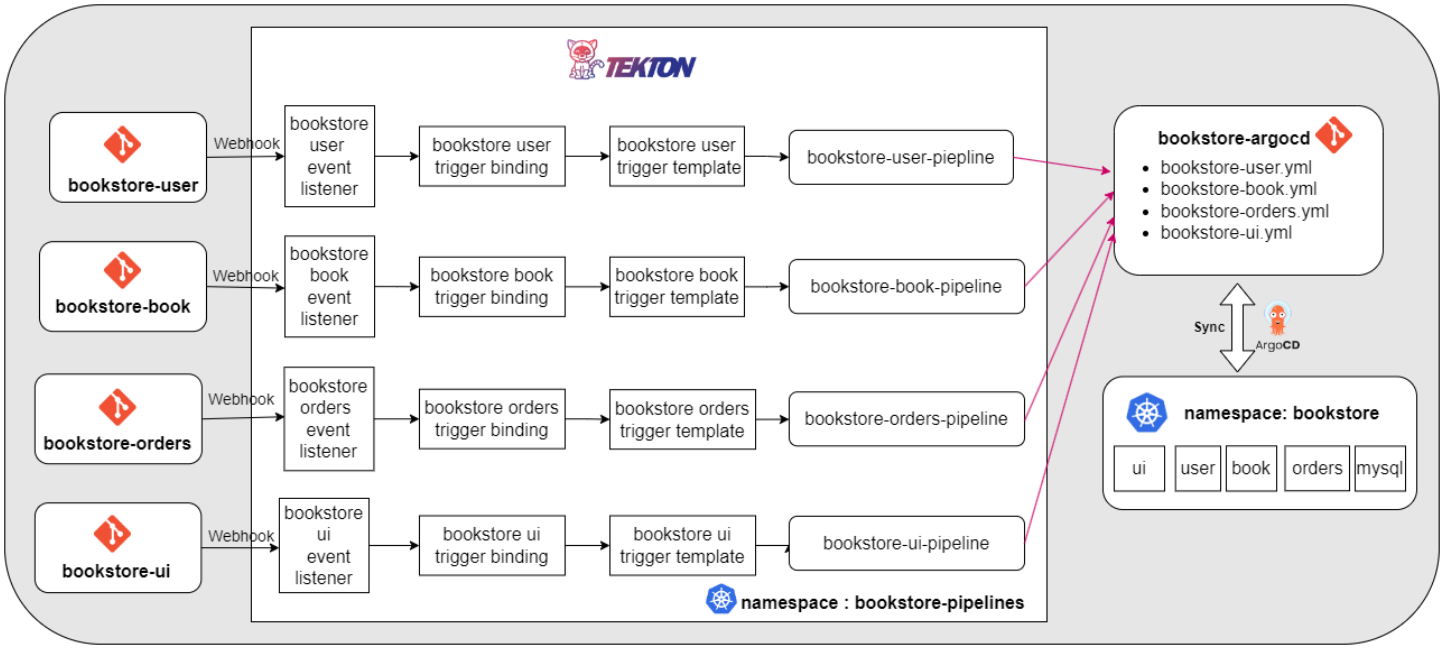

Fig. 7: CI/CD with Tekton and ArgoCD for bookstore application

6-A *Bookstore Application*

A bookstore application is a polyglot microservice application that lets you buy books online. You can check out books, add them to your cart, place an order, and see your order history and status. The admin user can easily add new books, providing details like author, price, and quantity. The app has four main parts: User Service, Orders Service, Bookstore UI, and Books Service, all working together make the application work. All the data is kept in database, MySQL, to keep the data persistent. The Bookstore UI, made with HTML, CSS, and JavaScript, is easy for customers and admin users. Depending on user role, the User Service makes sure you can do what you need. It's made with Java. The Orders Service, made with Python, handles cart details, order history, and delivery status, working well with the other parts. The Books Service, in Java, manages info about the books. This application pays a lot of attention to every detail, making sure everything from looking at books to getting your order is easy with its mix of different services.

6-B *Kubernetes Cluster setup*

For this project, we set up Kubernetes on Google Cloud Platform (GCP). We made a Virtual Private Cloud (VPC) network in GCP for creating virtual machines. The Kubernetes cluster has three nodes: one master and two slave nodes. Each node is like a virtual machine with 2 vCPUs, 8GB of RAM, and a 40GB disk. All these nodes run on the Ubuntu 20.04 LTS operating system. We added one more node with 2 vCPUs, 2GB of RAM, and a 40GB SSD. This special machine acts as a Network File System (NFS) server for storing things in one central place. In Kubernetes, an NFS server is useful for keeping data that containers can use. To set up the Kubernetes cluster, we used Rancher Server, a tool that makes it easy. We chose Kubernetes version 1.26 for the cluster. The Kubernetes cluster is configured to dynamically provision (Persistent Volumes or PVs) on the NFS server. This storage can be used by applications or pipelines running in the cluster. When an application or task

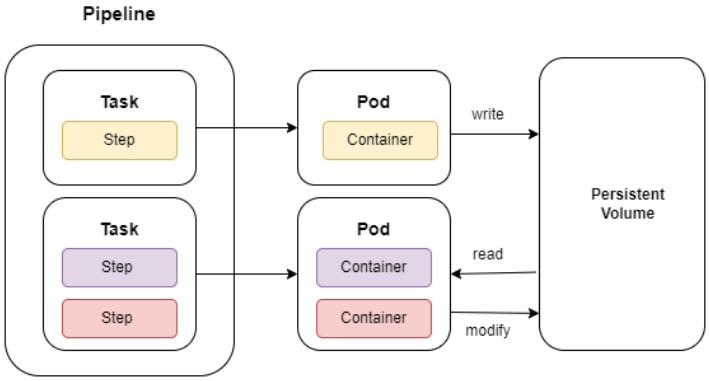

Fig. 8: Tekton pipeline tasks running in ephemeral pods

needs storage, it asks for it by creating something called a PersistentVolumeClaim (PVC). Kubernetes then makes a storage space (PV) just for that application or pipeline. The PV can be connected to the application container or pipeline, letting it use the shared storage. This makes a good way to store data that many applications and pipelines or nodes in the Kubernetes cluster can use, making things scalable and dependable.

6-C *Tools and Pipelines Configuration*

On top of Kubernetes, we need to install CI/CD tools. We considered Tekton for our CI and Argo CD for CD. Tekton with version of 0.53 is installed in tekton-pipelines namespace. For the Tekton pipelines to listen to the events, Tekton triggers with version 0.25.2 is installed in same namespace. For GitOps CD, Argo CD 2.9.0 is installed in argocd namespace. In order to put all the code and deployment manifests, we used Gogs as git hosted repository. In this git repository, we put all the bookstore application codes and Kubernetes deployment manifests required for bookstore application deployment. As shown in Fig. (7), we have different git repository for each microservice. For each microservice, when the developer changes the code, there need

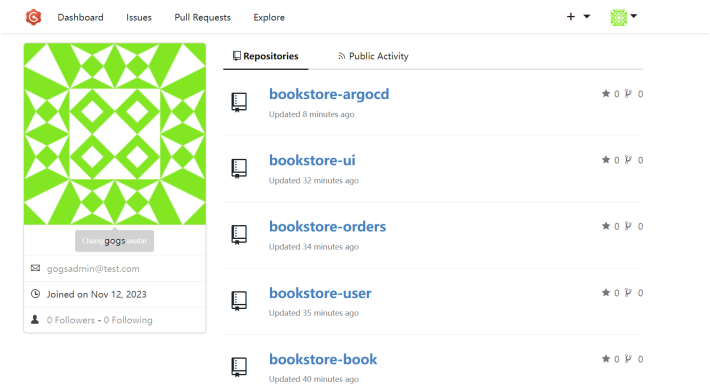

Fig. 9: Gogs git repository with code and deployment manifests

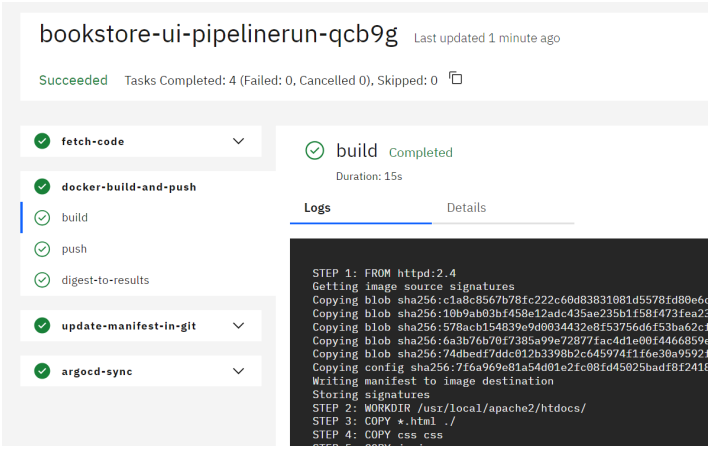

Fig. 10: Tekton pipeline tasks running in ephemeral pods

to be a pipeline that builds and deploys the application. In our pipelines, we follow the sequence of steps taking the code, building it as a container image, pushing it to Docker hub and then deploying the application with argocd. Each step in the pipeline is an individual task as shown in Fig. (8). Whenever there is a change in the code, our pipeline needs to be triggered. For that we create triggers, that listens to the event as shown in Fig. (7). There is an event listener that gives a public URL, and whenever there is change in git repository, these public URL will be invoked with some data payload. That data is read with trigger templates and binds to our pipelines using trigger binding. This also creates a PipelineRun, an instance of our pipeline. This way on every change in the code, pipeline runs will get created.

7 Results

For testing this project for serverless way of deploying applications to Kubernetes, we follow a series of steps. First step is to do any update to the existing code. This changes in the code will trigger a pipeline. Our next step is to ensure that the pipeline is triggered and each task in the pipeline is running in ephemeral pod. All these tasks, cloning of code and building and pushing container image and updating the Kubernetes manifests will be done in the pipeline execution. Next task is to deploy these new manifests using Argo CD. The last step is to verify that Argo CD is synchronizing the changes to the Kubernetes cluster.

As part of first step, Fig. (9) shows the Gogs code repository where all the bookstore application's microservice code and Kubernetes deployment manifests are located. We manually commit some changes to the bookstore user interface microservice as how developers commit their changes in real time. Now, the code commit is done. We need to verify our pipeline is trigged and tasks in the pipeline are running in ephemeral pods. Fig. (10) shows each task in the pipeline is executing step by step in an ephemeral pod.

In this way we have achieved serverless way of deploying applications to Kubernetes. By this time, a new container image is built for bookstore user interface and the Kubernetes manifests are updated with this new version image. As a final step, we need to verify that Argo CD synchronizes the changes and deploys the newly generated container image to Kubernetes. Fig. (11) shows that our cluster is in sync with the new changes and we can access the newly deployed application. Finally, we achieved serverless

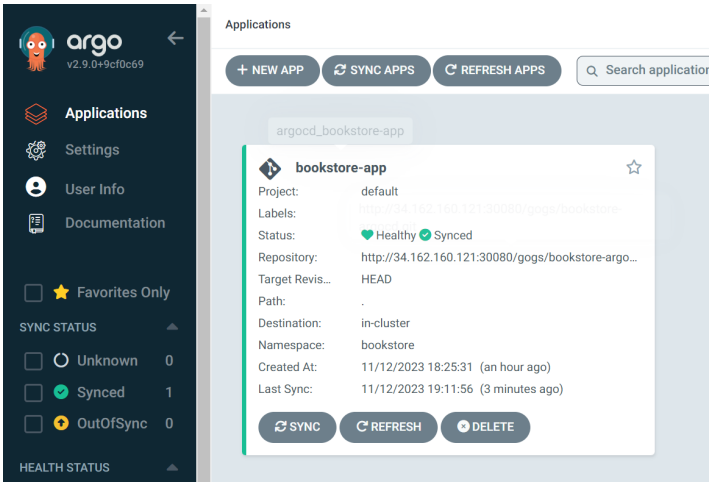

Fig. 11: Argo CD synchronizing changes with git repository

way of building the application and deploying applications in a GitOps way of delivery.

Table (I), shows the comparison of different CI/CD frameworks available. In evaluating various Continuous Integration/Continuous Deployment (CI/CD) solutions, several key factors come into picture. Jenkins, a opensource widely used tool for CI/CD, is not serverless but is Kubernetes-native and supports only push deployment workflows. GitLab, although it is not serverless and lacks in native Kubernetes integration, it is portable like Jenkins, and open-source nature, operating under a freemium pricing model. GitHub Actions, a serverless solution, is not portable because of the vendor lock in with GitHub and refactoring may be needed for portability. Most of the Cloud providers such as GCP (Cloud Build), AWS (CodeBuild and CodeDeploy), and Azure DevOps offer push-based workflows and GCP uses containers to build the applications. AWS hides the implementation details, but it offers scalability with pay-as-you-go pricing. Notably, Tekton with ArgoCD emerges as a best choice among other popular CI/CD tools, for being serverless, Kubernetes-native, pull-based, highly portable, open-source, and, significantly, free. This combination of features positions Tekton with ArgoCD as an appealing solution for organizations who are looking for a flexible, cost-effective, and Kubernetes-centric CI/CD workflow.

TABLE I: Comparison with CI/CD frameworks

CI/CD Solution	Serverless	Kubernetes Native	Push/Pull Based	Portable	Open Source	Pricing Model
Jenkins	No	Yes	Push	Yes	Yes	Free
GitLab	No	No	Push	Yes	Yes	Freemium
GitHub Actions	Yes	No	Push	No	No	Freemium
GCP (Cloud Build)	Yes	No	Push	No	No	Pay-as-You-Go
AWS CodeBuild and CodeDeploy	No	No	Push	No	No	Pay-as-You-Go
Azure DevOps	No	No	Push	No	No	Pay-as-You-Go
Tekton with ArgoCD	Yes	Yes	Pull	Yes	Yes	Free

8 Conclusion

This project uses Tekton, a serverless CI framework, and Argo CD, a CD tool, for building and deploying applications. Compared to other CI/CD tools, our solution fully leverages cloud and serverless benefits, making the platform more agile, cost-effective, scalable, and portable across different clouds. Additionally, it follows the GitOps model, enabling security, version control for both infrastructure and application code. This speeds up development, deployment, and enhances the reliability of application systems. However, it may not be suitable for smaller projects due to higher resource usage, making it more suitable for larger organizations with numerous applications. It's important to note that Tekton lacks built-in support for workflows requiring approvals before deployment sign-offs.

Acknowledgement

We, as a team, express our collective gratitude to Dr. Augustine Samba, Professor at CS Department, and Hemanth Lella, Senior Developer at Optum, for their invaluable guidance and unwavering support throughout the project.

References

[1] Ziyu Li. Comparison between common virtualization solutions: Vmware workstation, hyper-v and docker. In *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, pages 701–707. IEEE, 2021.

[2] Artur Cepuc, Robert Botez, Ovidiu Craciun, Iustin-Alexandru Ivanciu, and Virgil Dobrota. Implementation of a continuous integration and deployment pipeline for containerized applications in amazon web services using jenkins, ansible and kubernetes. In *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6. IEEE, 2020.

[3] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. Comparison of different ci/cd tools integrated with cloud platform. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 7–12. IEEE, 2019.

[4] Carmelo Aparo, Cinzia Bernardeschi, Giuseppe Lettieri, Fabio Lucattini, and Salvatore Montanarella. An analysis system to test security of software on continuous integration-continuous delivery pipeline. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–67. IEEE, 2023.

[5] Håkon Teppan, Lars Halvdan Flå, and Martin Gilje Jaatun. A survey on infrastructure-as-code solutions for cloud development. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 60–65. IEEE, 2022.

[6] *https://docs.gitlab.com/ee/ci/docker/using*$_{d}ocker_{i}mages.html$.

[7] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245. IEEE, 2022.

[8] Magomedbashir Kushtov. Serverless ci/cd pipeline based on google cloud platform. 2022.

[9] Saumya Gupta, Madhulika Bhatia, Meenakshi Memoria, and Preeti Manani. Prevalence of gitops, devops in fast ci/cd cycles. In *2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*, volume 1, pages 589–596. IEEE, 2022.

[10] Ema Utami, Hanif Al Fatta, et al. Analysis on the use of declarative and pull-based deployment models on gitops using argo cd. In *2021 4th International Conference on Information and Communications Technology (ICOIACT)*, pages 186–191. IEEE, 2021.

[11] Yahuza Bello, Emanuel Figetakis, Ahmed Refaey, and Petros Spachos. Continuous integration and continuous delivery framework for sds. In *2022 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 406–410. IEEE, 2022.

[12] Jamal Mahboob and Joel Coffman. A kubernetes ci/cd pipeline with asylo as a trusted execution environment abstraction framework. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0529–0535. IEEE, 2021.

[13] Rabindra K Barik, Rakesh K Lenka, K Rahul Rao, and Devam Ghose. Performance analysis of virtual machines and containers in cloud computing. In *2016 international conference on computing, communication and automation (iccca)*, pages 1204–1210. IEEE, 2016.