IJARIIT INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X

Impact Factor: 6.078

(Volume 7, Issue 4 - V7I4-1586)

Available online at: https://www.ijariit.com

Designing, deploying and managing cloud-native applications effectively using Docker, Jenkins, and Kubernetes.

Jyothi Koushik Paladugu
jyothikoushik1998@gmail.com
Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, Telangana

ABSTRACT

From storing and retrieving information in databases online to delivering various services such as servers, and computing resources over the internet, cloud computing has brought tremendous advancements in the technological world. One of the major sectors of cloud computing is application development. A wide variety of applications are being developed which offer services for almost every working industry globally. Several debates and discussions have arisen on designing the application in the best way possible. Extensive research studies were done using various parameters such as resource usage, storage, efficiency to determine the most optimal method. Cloud-enabled applications were the primary type developed, followed by cloud-based applications. However, the demand for lightweight and flexible applications designed at the least cost possible rendered these approaches obsolete. That is where the cloud-native model came into the picture. This paper explores the model and presents how it is used to design, deploy, and manage cloud-native applications, including discussing the model's core elements and implementing them in making the application.

***Keywords**: Lightweight, Containers, Microservices, Scalability, Load Balancer, DevOps*

1. INTRODUCTION

The art of designing applications has evolved over the years. Catering to the users' needs completely and developing the product at the lowest cost possible is challenging. Developers initially used Service-oriented architecture (SOA) to design products in earlier times. However, with the introduction of "Cloud" and "Cloud computing" in the 1990s, the entire application development phase has taken a shift. Cloud services have become hugely popular in less time because managing large modules and data through interconnected servers has proven advantageous with easy maintenance, almost negligible installation time, and more variety of features. Initially, applications were developed externally and integrated into the cloud before the final run, called "Cloud-enabled" applications which was the traditional way of developing an application. Although this model provides better performance and security due to its underlying architecture, it required high maintenance. The "cloud-native" model was designed to overcome the drawbacks of this model and provide an overall experience.

A Cloud-native application is a set of elastic, independent components developed and deployed entirely in the cloud-based environment. Unlike the other models designed externally and then migrated into the cloud, a cloud-native application is designed inside the cloud from scratch, where all migrations are done automatically, and the application is independent of the hardware configuration. The underlying architecture of these applications is designed using Microservices, where all the modules are broken down into smaller services independent of one another with their respective dependencies and configuration. Finally, these services are packaged using Container services integrated and deployed at regular intervals using Continuous Integration. These modules and the entire workflow is generally implemented using cloud service providers.

What makes the cloud-native approach the most reliable and better than the traditional model? The previous model's functionality was based on the architecture where services were bound together as a module, and these models rely heavily on hardware. As the cloud-native model relies on Microservices, the application developed is flexible, resilient and the request-response time is low. Implementation of crucial tasks such as upgrading modules and systems diagnostics becomes easier with minimal interruptions. As these applications are developed inside the cloud, there is no need for purchasing computing resources as the cloud platform automatically configures them. As a result, these applications are lightweight and consume fewer resources. In addition, these applications are very cost-efficient as the entire construction is done in the public cloud, where the total cost is only for those services used.

The rest of the paper primarily focuses on and explores this idea. The next section discusses the primary components of a cloud-native application in detail and how they help develop a robust application, and their role in justifying the cloud-native principle. The paper then introduces the making of the application, where a Django web application is designed. Subsequently, all the components discussed earlier will come into play in the design workflow, testing, packaging, automation, deployment, and maintenance of the app, which is followed by an emphasis laid on how the application is managed post-release and how the application metrics, performance logs are used to monitor along with scaling the application. Finally, the paper concludes with a summary of the proceedings.

2. COMPONENTS OF A CLOUD NATIVE MODEL

The cloud-native model's primary purpose is to extract all the benefits that cloud computing can offer and provide the best application development experience. Therefore, it is crucial to assess the significant modules that define this schema and know how their functionalities make this model successful. Furthermore, understanding the roles and uses of these components will help in developing a robust application. The three main parts of this architecture include Containers, Microservices, and Continuous Integration. This section will examine all the components and their role in developing efficient products in detail.

2.1 Containers

Initially, when an application was designed, many hardware resources were required to keep the product intact but managing them was difficult. The best solution developed then was virtual machines (V. A virtual machine functions as a system with its resources like Central Preprocessing Unit (CPU), Operating system, and storage. They virtualize the hardware beneath so that multiple VM's could run on a single hardware resource, eliminating redundancy. However, virtual machines couldn't handle the heavier system loads. Containers are a perfect alternative to virtual machines. Containers are pieces of software that bundle up all the source code, dependencies, and surprisingly even an Operating System (OS) into a single unit using a container engine (Shown in Figure 1). The biggest asset of containers is that they perform OS - Level virtualization that makes them easy to use, unlike virtual machines, which need a hypervisor to function. In addition, they can run on the Host OS itself, enabling them to run on various environments irrespective of the physical hardware underneath.

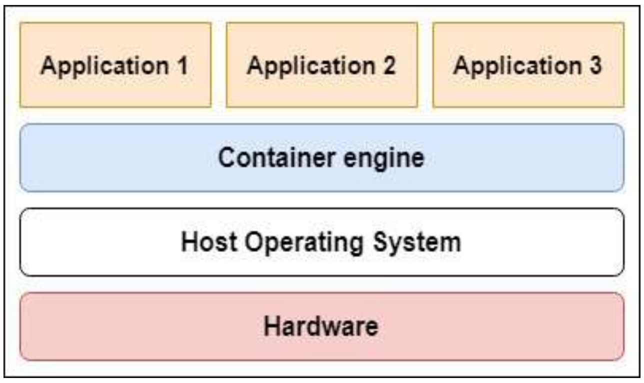


Figure - 1: Container architecture: Virtualization in Containers

Containers offer better load-handling, stable data recovery and backup, robust security features. They save resources and cut down processing costs because they do not require much hardware to configure and run. They also boost the server's performance because of their portability. The container image, which is the most crucial constituent of developing a container, is an executable software with all the code, packages required to build a container. Thousands of Pre-defined images are available in the central repository, or custom image files can be written using shell commands or a base file with all the code required to build the image from scratch.

2.2 Microservices

Microservices is an architectural style where all components are divided into separate, independent modules. Microservices have become very important in the developer community, unlike monolithic architecture, which combines all the components into a single unit. Microservices aim at high efficiency, better fault reduction, and multiple task building. It segregates the entire workflow into numerous loosely coupled services, running its task and processes. Each segment has its share of resource pool such as memory, databases, and servers, as shown in Figure 2.

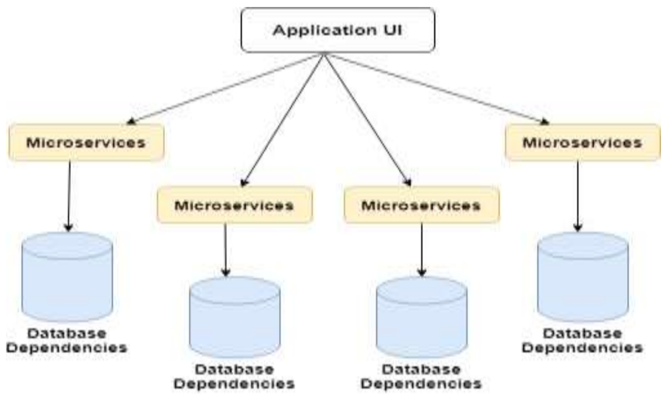


Figure - 2: Containers Distribution of application modules using Microservices

The monolithic architecture has many drawbacks: high cost, frequent components malfunction, and no opportunity to scale. When the application fails to work, it becomes difficult to identify the code segment's error and rectify it because all the services are bound together as a unit. Moreover, it also consumes additional resources to manage this task. On the other hand, Microservices offer the best alternative in terms of performance and maintenance. As the entire working block is sub-divided, errors and bugs can be easily identified with trigger logs and debugging. In the view of a cloud-native application, Microservices are very useful because they promote deployment flexibility and reusability. This feature can scale the services separately without any external interference.

2.3 Continuous Integration (CI)

Continuous integration is a process where project code and modules are pushed into repositories and integrated consistently. CI commits the code periodically to the main branch in the central repository and regularly creates checkpoints and log messages. As a result, all the major tasks such as planning, building, testing are done simultaneously, and small releases are done daily. As a result, it is highly efficient, facilitates easy error detection, promotes faster release, and ensures customer satisfaction. The central repository still holds the up-to-date commits, and the files can be retrieved, ensuring data backup and security. Thus, continuous integration is a crucial component in the cloud-native model. Its scope is attributed to the overall development of a good product.
Continuous integration and Delivery were initially carried out manually (dashboards). As time passed, automated software that can perform CI has come into the market. They have introduced the concept of pipelines and triggers to support the automatic integration of file building. Pipelines implement the automatic installation, building testing, and deployment of code. The most significant advantage is that there is no manual work required. It continuously commits the tasks and reports errors and timestamps until it comes across a stop function and exits. When there is a need to attend a particular section of the code, developers can also use triggers to force-implement that specific workflow. The rate of building application improves and, more importantly, the quality of the final product.

3. DESIGNING AND PACKAGING THE APPLICATION

The complete process of building an efficient cloud-native application consists of two phases. The first phase covers the planning and designing of the application and packaging it using containers. The second stage deals with the deployment and management stages. This section focuses on the first phase of development, and the second phase is examined and discussed in the subsequent sections of the paper.

3.1 Planning the Application design

The first and foremost task to build an application is to plan and design the application with a workflow design covering all the modules and its functionalities. For example, a simple blog application should create new user accounts and support user login. In addition, the website should provide a form structure for article creation and a database to maintain the records and display the posted articles. The use of Microservices is very crucial as it can break down the components and ensure faster development. The simple yet detailed workflow is shown in Figure 3.

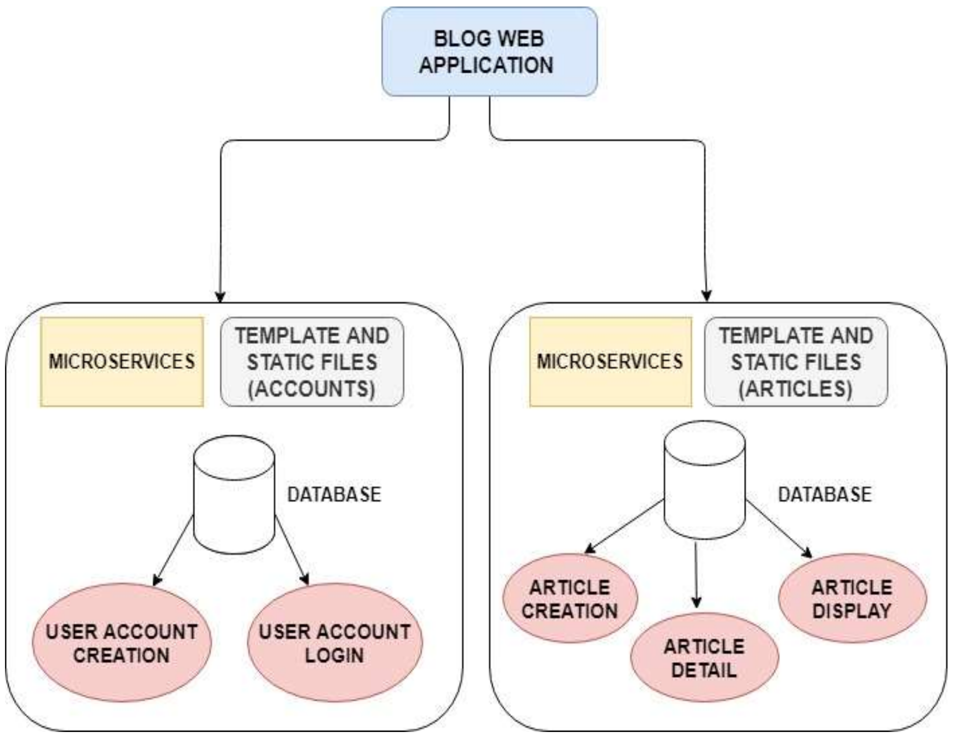
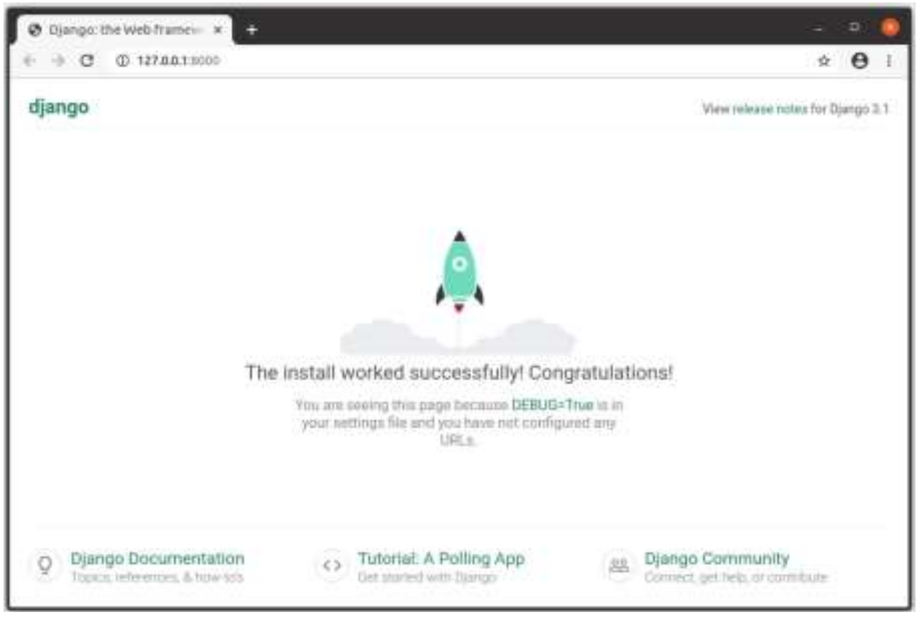


Figure - 3: Blog application designed using Python and Django consisting of Account and Article templates (Architecture Workflow)

3.1.1 Installing modules and testing the server: Python is used to design the application in this scenario. It is a high-level programming language used in a variety of applications and is open-source software. Django: a powerful web framework is required for this purpose which can be installed using pip: a package installer for Python. The following commands (numbered for convenience) are used to install all the required modules for the project.

```
python –-version        // [1]
pip install django // [2]
pip install pillow  // [3]
pip install django-microservices      // [4]
django-admin startproject <app_name>             // [5]
python manage.py migrate// [6]
python manage.py runserver          // [7]
```

Python can be installed using the (.exe) downloadable file from the internet. After installation, the versions of both Python can be verified by [1]. We can install Django with the command [2] that pulls out the latest version available. The web application also requires a package named "Pillow," which supports multiple images and video formats [3]. To also implement Microservices in the design structure, another module called "Django-microservices" is available [4]. Multiple services communicate over the Hypertext Transfer Protocol Secure (HTTPS) server in a Django web application. This package enables the module to activate a separate development server for each service instead of a single server. Command [5] is used to create a project file. This process makes the primary database files and configuration files consisting of all the necessary configurations and server control commands required to run the server. Before the actual development commences, it is essential to synchronize all the changes done when constructing the models to the database schema, which can be done by the migrate command [6]. The server can be accessed using [7], which connects to port 8000 over the localhost using the manage.py file, as shown in Figure 4.



The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Figure - 4: Testing Django server after installation and migration using Localhost server port 8000

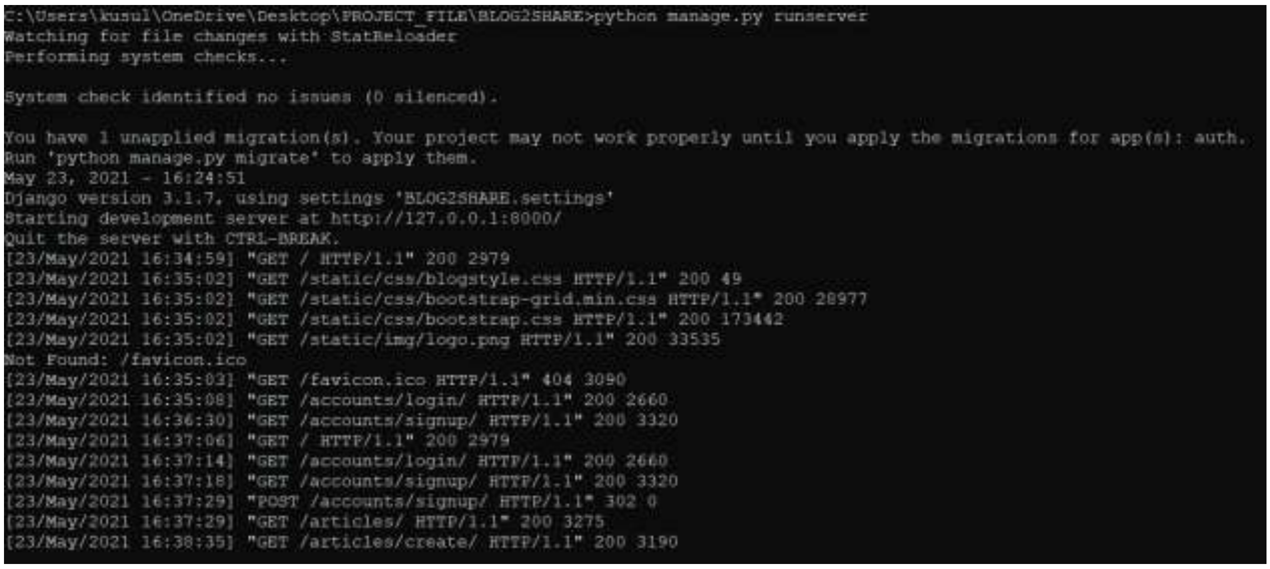
3.1.2 Designing the accounts page (Login and Signup): The first page to be built in the web application will be the homepage which displays all the details regarding the type of website and the service it provides to the users. In our case, this is a blog application that allows writers to write and post articles, and a vast number of readers can read articles of their interest. Templates deal with HTML file generation. The homepage re-directs to a Signup page or a login page where new users can create an account with valid credentials or log in with their existing account, granting them access to post articles. The website database should support the storage of all user credentials and verify them during login to ensure safe access to the website and the account. For this purpose, Django has an inbuilt SQLite Database that securely takes care of the user's data. Finally, an "accounts" model is created, consisting of all the static files (HTML, CSS, and JavaScript files), Uniform Resource Locator (URL) and view files required to function. The login and the signup page are created to validate users, and the code is as shown below.

```
def signup_view (request):                  // (sign-up new user)
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request,user)        // (log the user in)
            return redirect('articless:list')
    else:
        form = UserCreationForm()
    return render(request,'accounts/signup.html',{'form': form})
def login_view(request):                // (login existing user)
    if request.method == 'POST':
        form = AuthenticationForm(data=request.POST)   // (form request)
        if form.is_valid():
            #log in user
            user = form.get_user()
            login(request,user)
            if'next' in request.POST:
               return redirect(request.POST.get('next'))
            return redirect('articles:list')
      else:
         form = AuthenticationForm()
      return render(request,'accounts/login.html',{'form':form})
def logout_view(request):          // (Account logout)
   if request.method == 'POST':
      logout(request)
      return redirect('articless:list')
```

3.1.3 Designing the articles page (Create, Post and Display): The following task is designing the article-related modules. In a blog, the articles are the most crucial elements. The website should provide a structure for the writers to share their views, and the readers should be able to access all the posted articles in the order of time. All the latest articles published by different authors should appear on the homepage. We create a separate template named 'Article,' which consists of two sub templates: 'article_create.html' and 'article_detail.html.' The websites inherit the properties of the base layout with a POST response. The article creation form consists of the title, slug, body, and link for image insertion. As soon as the article is online, it should appear on the home page with the timestamp and the author's username. The corresponding code is as follows.

```
def article_list (request):        // (displaying list of articles)
        articles = Articles.objects.all().order_by('date')
        return render(request,'articles/articles_list.html',{'articless':articless})
def article_detail(request,slug):    // (article information)
        articles = Articles.objects.get(slug=slug)
        return render(request,'articles/articles_detail.html',{'articles':articles})
def article_create(request):       // (article creation form)
        if request.method == 'POST':
                form = forms.CreateArticles(request.POST,request.FILES)
                if form.is_valid():
                        instance = form.save(commit = False)
                        instance.author = request.user
                        instance.save()
                        return redirect('articles:list')
        form = forms.CreateArticles()
        return render(request,'articles/articles_create.html',{'form':form})
```

3.1.4 Testing the application: After developing the logic, it is very crucial to test the functionalities of the application. Testing is a crucial process to achieve a quality product. Django provides the facility to test the website on our local-host server on port 8000 at any stage of development that can detect anomalies and improve the functionalities. Django also provides the facility of producing timestamp log messages, as shown in figure 5, during the interaction with the website, making it easy to identify the code breaks and errors. The facility of preview is also available, enabling us to view the end product beforehand.

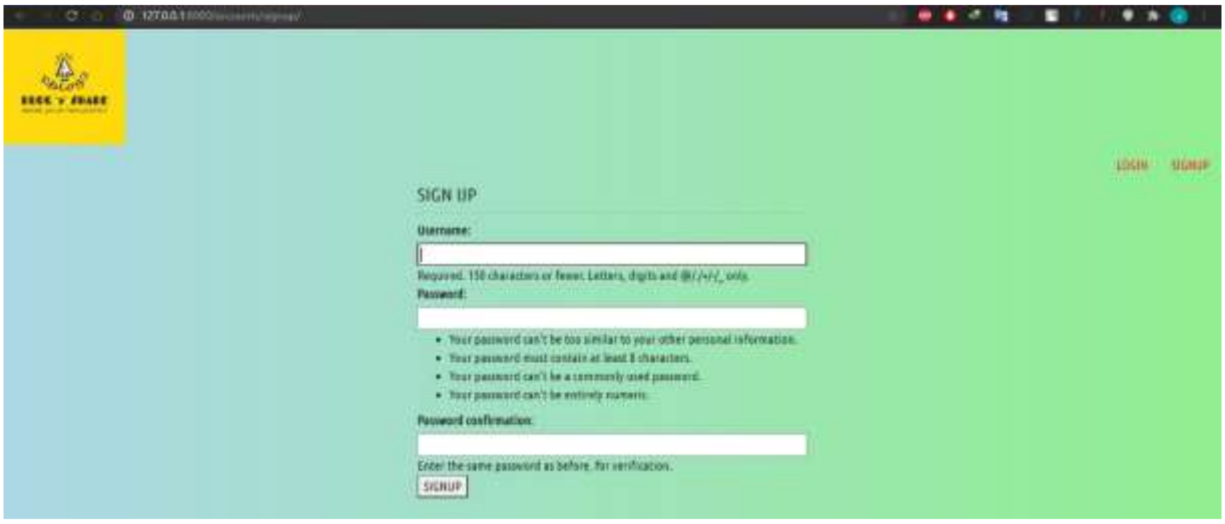


```
C:\Users\kusul\OneDrive\Desktop\PROJECT_FILE\BLOG2SHARE>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 1 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): auth.
Run 'python manage.py migrate' to apply them.
May 23, 2021 - 16:24:51
Django version 3.1.7, using settings 'BLOG2SHARE.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[23/May/2021 16:34:59] "GET / HTTP/1.1" 200 2979
[23/May/2021 16:35:02] "GET /static/css/blogstyle.css HTTP/1.1" 200 49
[23/May/2021 16:35:02] "GET /static/css/bootstrap-grid.min.css HTTP/1.1" 200 28977
[23/May/2021 16:35:02] "GET /static/css/bootstrap.css HTTP/1.1" 200 173442
[23/May/2021 16:35:02] "GET /static/img/logo.png HTTP/1.1" 200 33535
Not Found: /favicon.ico
[23/May/2021 16:35:03] "GET /favicon.ico HTTP/1.1" 404 3090
[23/May/2021 16:35:08] "GET /accounts/login/ HTTP/1.1" 200 2660
[23/May/2021 16:36:30] "GET /accounts/signup/ HTTP/1.1" 200 3320
[23/May/2021 16:37:06] "GET / HTTP/1.1" 200 2979
[23/May/2021 16:37:14] "GET /accounts/login/ HTTP/1.1" 200 2660
[23/May/2021 16:37:18] "GET /accounts/signup/ HTTP/1.1" 200 3320
[23/May/2021 16:37:29] "POST /accounts/signup/ HTTP/1.1" 302 0
[23/May/2021 16:37:29] "GET /articles/ HTTP/1.1" 200 3275
[23/May/2021 16:38:35] "GET /articles/create/ HTTP/1.1" 200 3190
```

Figure - 5: Testing the Django server after building the accounts and the articles templates and configuration

3.1.5 Styling the application design: After coding and testing the functionalities, the next task is to create an attractive website design. Organizing all the components and elements so that they are clear and, at the same time, visually engaging is important because it captures the attention of the user. Using frameworks like Bootstrap and static files like JavaScript can play a crucial role in designing a neat website. Bootstrap offers higher compatibility and better models for implementation. All the User Interface (UI)

Components templates are flexible and reusable, making them very easy to use. As shown in figure 6, the login and the signup pages are organized with relevant elements to interact with, and that follows the same with the articles create form and the display page.



SIGN UP

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

SIGNUP

Figure - 6: (Rough page) Final view of the application after design handling

3.2 Containerizing the application

After designing the application and ensuring its functionalities are intact by testing, the next major step is to containerize it. The concepts of Containers and Continuous Integration (CI) play a primary role in this process. In simple terms, containerizing the application implies packaging the code. Although many tools provide container facilities, Docker is the best container-based software in the current era. It provides various features that can help developers customize their code according to their requirements and supports secured access. This section of the paper emphasizes creating container images and pushing them to the central repository through an automated process using a Continuous Integration tool "Jenkins" in a pipeline activity.

3.2.1 Configuration of Docker and writing Dockerfile: Before setting up, it is essential to create a Docker account that accesses the Docker hub, a repository that stores millions of pre-written and custom-built images. To access the specific repository, users must configure the authentication. In addition, Docker provides the facility of building public and private images based on security requirements. To use Jenkins to automate this process, we need to include the credentials so that Jenkins can automatically connect to Docker and authenticate for performing the required processes when the CI pipeline process is invoked.

To build a Docker image, the primary document required is the Dockerfile. It is a simple text document that consists of all the steps and commands required to package the code files, link ports and create an image. Docker tracks the specified path of the file and builds the image based on all the requirements described. A Dockerfile has no extension. When the command is executed, the build is done using the commands in the Dockerfile step-by-step. The following code explains the commands required to build a custom image for the blog application, including running the server.

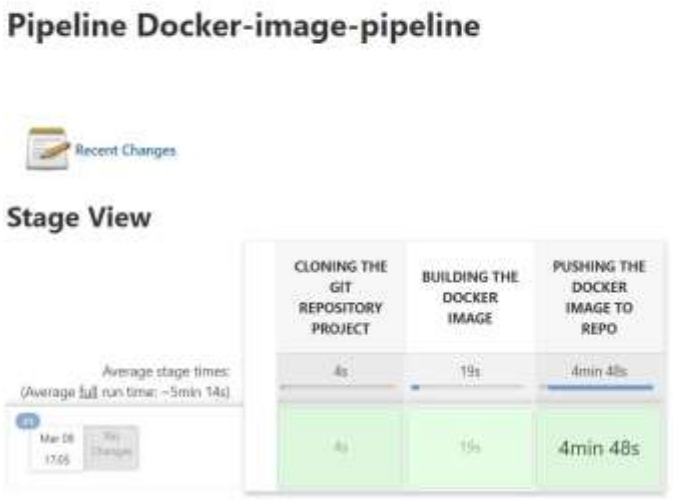
```
FROM python: 3.9           // (Dockerfile)
ENV PYTHONUNBUFFERED 1
RUN mkdir /cnapp1
COPY ./BLOG2SHARE /cnapp1               // (copying project file)
WORKDIR /cnapp1
RUN pip install --upgrade pip
RUN pip install django
RUN pip install pillow
CMD ["python", "manage.py", "migrate"]
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000”] // (execution)
```

3.2.2 Building the Jenkins pipeline: Jenkins offers a wide variety of CI services. The most prominent of all is the pipeline facility, enabling developers to build continuous integration pipelines to automate their tasks. First, a Jenkins Deployment pipeline is built, automatically accessing the code files from GitHub. Then, after pulling the files from the GitHub repository, the pipeline will execute, building the image based on the Dockerfile based on the credentials and pushes the image to the Docker hub.

```
  stages
    stage('CLONING THE GIT REPOSITORY PROJECT')
      steps
        git 'https://github.com/ESCANOR-1998/cnapp'
    stage('BUILDING THE DOCKER IMAGE')
      steps: script
          dockerImage = docker.build registry + "cnxapp:v1"
    stage('PUSHING THE DOCKER IMAGE TO REPO')
      steps: script
```

```
docker.withRegistry('',registryCredential)
   dockerImage.push()
```

The Jenkins dashboard can be installed and accessed on the local server port 8080. There are two ways of building a pipeline: using the dashboard (or) writing a source code file which consists of all the commands of building the pipeline. It is necessary to provide the link to the GitHub repository and the docker account credentials. The pipeline starts running in the 'stage view' section and completes the tasks. The image 'cloudnative1998' public image is built by the pipeline with all the requirements whose access is public, which can be pulled to the local system, and the container is built by running the container image by port mapping (Connecting the ports in the container and the Docker host respectively) from port 8000 to 8000 as shown in Figure 7.



	CLONING THE GIT REPOSITORY PROJECT	BUILDING THE DOCKER IMAGE	PUSHING THE DOCKER IMAGE TO REPO
Average stage times: (Average full run time: ~5min 14s)	4s	19s	4min 48s
Mar 08 17:05	4s	19s	4min 48s

Figure - 7: Jenkins pipeline execution status along with stage description

4. DEPLOYING THE APPLICATION

The cloud-native approach emphasizes automating every stage of development. After designing the web application and containerizing the application by building the container image, the next stage of development focuses on deploying, managing, and scaling the application. The best approach is to use Kubernetes, a container orchestration tool that automates the deployment and the scaling of containerized applications with clusters, nodes, and pods. Another added advantage is that Docker containers are highly compatible with Kubernetes that makes this process flexible. Kubernetes provides a User Interface to run the clusters on the local system itself. The best approach is to use this service to troubleshoot for application anomalies and deploy the fully functional application using a cloud service provider, as discussed at the beginning of the paper.

4.1 Basic functional units in Kubernetes

Before proceeding to the actual deployment process, it is essential to have a clear idea about the basic functional units used in Kubernetes: **pods**, **nodes**, and **clusters**. Pods are the smallest functional units that can be deployed in Kubernetes. In simple terms, the containers run on pods with shared resources and instructions on how to run the applications. A node is typically a virtual machine on which the pods run with the necessary configuration. The nodes are managed by a 'Control plane,' which holds the required network protocols to transfer data. Finally, a Kubernetes cluster is a set of nodes that run the application by integrating the code with the required modules and dependencies. The basic structure of a cluster is shown in Figure 8.

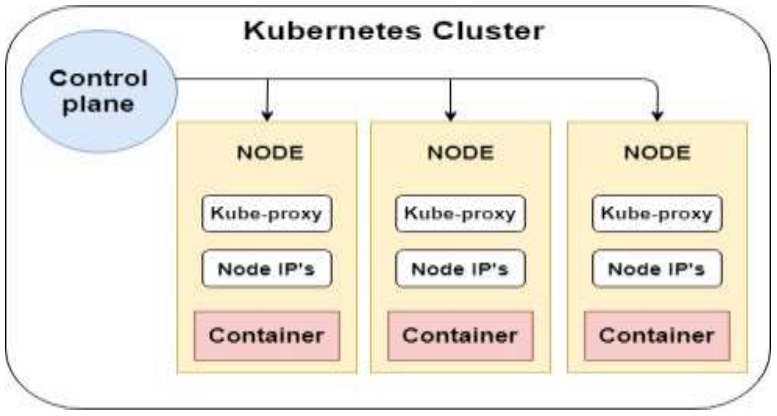


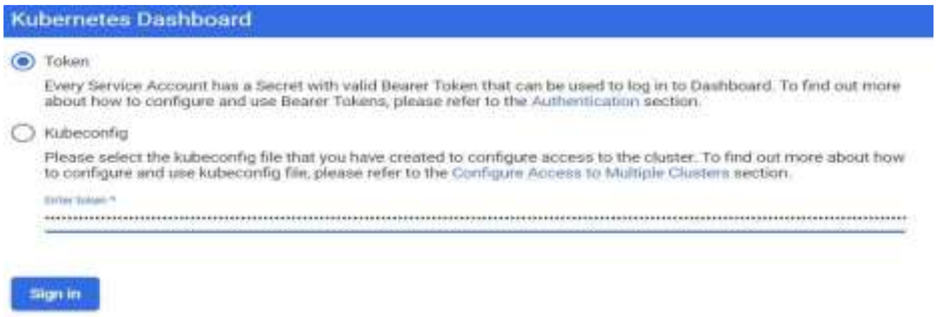
Figure - 8: Structure of a Kubernetes Cluster

A Kubernetes service is a rule-set that describes and defines a specific policy of running pods. In addition, these services are used to configure specific types of properties to the application. There are many services, but the primary focus will be on two services: '**Cluster IP**' and '**Load Balancer**.' Cluster IP is the basic type of service that exposes the application to an internal IP address not accessible from the outside world. The load balancer service divides the entire tasks among multiple resources to efficient workflow and maintains an even workload. Unlike Cluster IP, this service exposes the application to the outside world through an external IP that can be performed in a cloud-service application such as Azure, Google cloud.

4.2 Deploying the application as a Cluster IP type using Kubernetes dashboard

Before deploying the application, it is very crucial to double-check whether all the modules of the application are working accordingly or not because after the process of the actual deployment, if any functional error arises, it will have a detrimental effect on the product and will consume extra resources rather than saving it. Kubernetes provides a web interface called the 'dashboard,' which can run on our localhost server and can manage every activity from creating the cluster to deploying and management. It manages every deployment by monitoring the health of every component, including pods which helps in knowing how the application works overall and perform some upgrades before the actual deployment. The commands used to deploy the dashboard are below: the dashboard UI is cloned from GitHub, and a token is generated to access the interface, and as shown in Figure 9, we can access using the generated token.

kubectl apply –f https://raw.githubusercontent.com/kubernetes/dashboard/
nano dashboard-admin.yaml
kubectl apply -f dashboard-admin.yaml
kubectl get secret -n $(admin-user -n -o jsonpath="{.data.token}")//**(secret token generation)**
kubectl proxy // **(connecting to dashboard)**


Kubernetes Dashboard

(●) Token
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

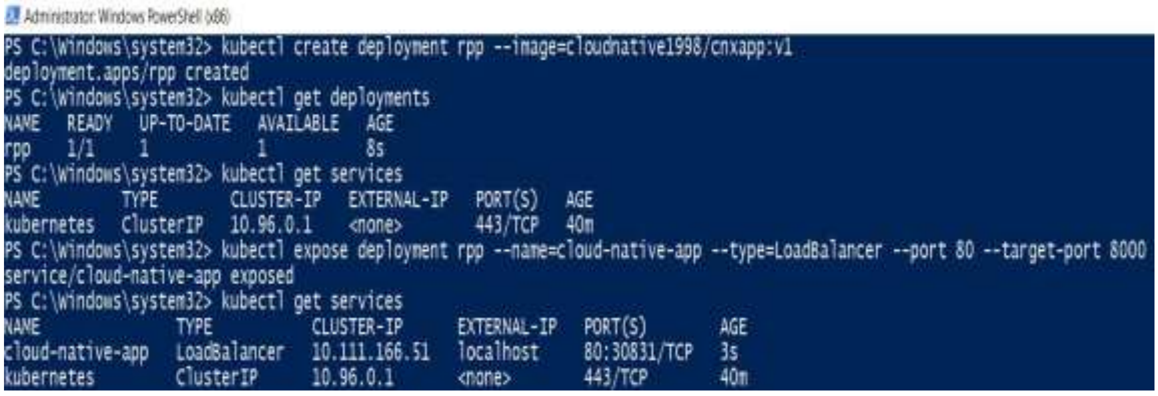
( ) Kubeconfig
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Enter token *
****

Sign in

Figure - 9: Accessing Kubernetes Dashboard using the generated token

The process of deploying the dashboard is done, and the deployment can be done. Using the docker image 'cloudnative1998/cnxapp: v1' built using the Jenkins pipeline, a deployment is created and exposed to the type 'Cluster IP' service (which means the access is only inside the cluster). Even if the deployment is exposed as a 'load balancer,' it will still be a service of type Cluster IP. The commands required to achieve this are described in Figure 10, where we have specified the port for the pods to interact with the server as 80, and the target port is 8080. The health of the pods created and the deployment is monitored, as shown in Figure 11 (the green colour indicates that the pods are healthy and fully functional). The services list is very informative, which displays all the required details regarding a particular service. Figure 12 displays all the contents of our deployed service' cloud-native-app,' including the cluster IP address and the endpoints (the localhost address and the port from where the application is accessed).


Administrator: Windows PowerShell (x86)

```
PS C:\Windows\system32> kubectl create deployment rpp --image=cloudnative1998/cnxapp:v1
deployment.apps/rpp created
PS C:\Windows\system32> kubectl get deployments
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
rpp    1/1     1            1           8s
PS C:\Windows\system32> kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   40m
PS C:\Windows\system32> kubectl expose deployment rpp --name=cloud-native-app --type=LoadBalancer --port 80 --target-port 8000
service/cloud-native-app exposed
PS C:\Windows\system32> kubectl get services
NAME               TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
cloud-native-app   LoadBalancer   10.111.166.51   localhost     80:30831/TCP   3s
kubernetes         ClusterIP      10.96.0.1       <none>        443/TCP        40m
```

Figure - 10: Creating Cluster IP deployment using Kubernetes Dashboard

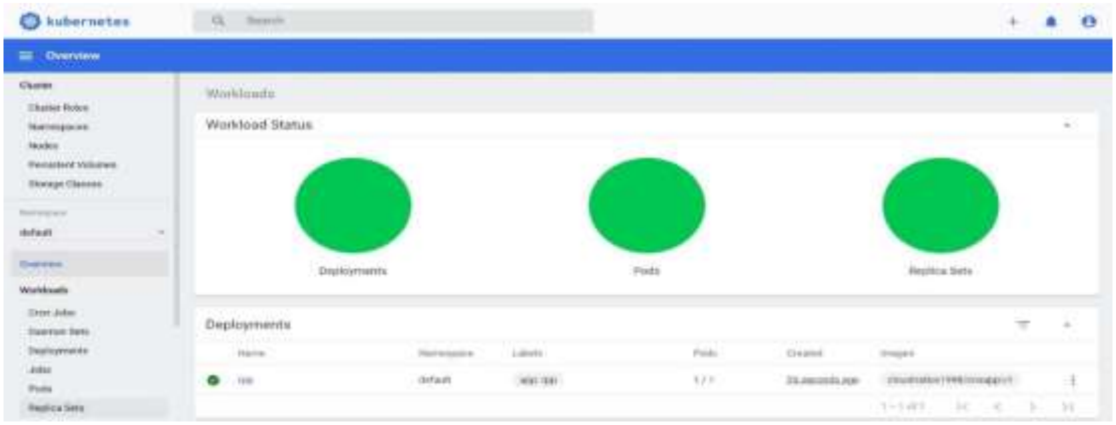
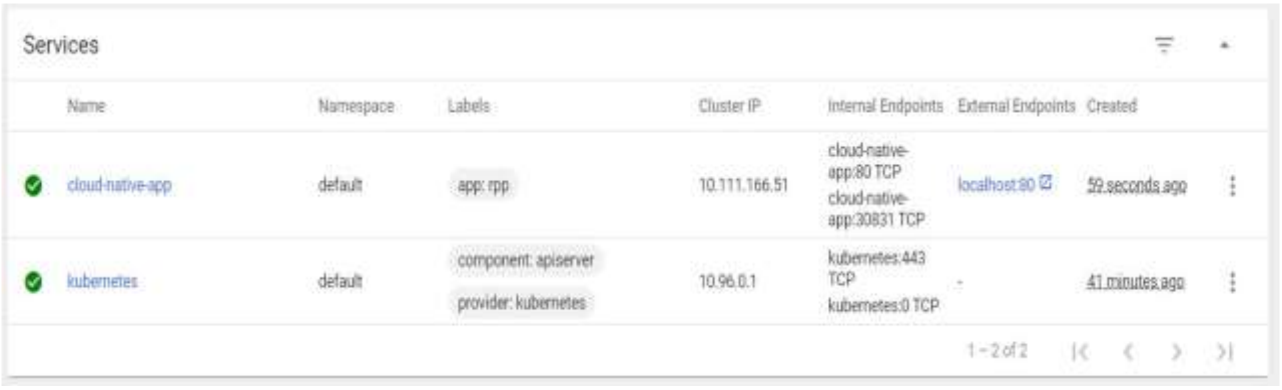


Figure - 11: Kubernetes UI Dashboard page (consisting of all the required details concerning deployment, pods)



Services

	Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created
✓	cloud-native-app	default	app: rpp	10.111.166.51	cloud-native-app:80 TCP cloud-native-app:30831 TCP	localhost:80	59 seconds ago
✓	kubernetes	default	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	41 minutes ago

1 – 2 of 2

Figure - 12: List of successfully deployed service units along with configuration info

The Kubernetes dashboard provides significant advantages that take the final product quality to the next level. The dashboard provides the opportunity to troubleshoot the deployed applications to find out the weak points of the schema that can threaten the workflow. It also provides the platform to monitor and manage the deployment resources, which gives a clear and concise idea of how the application would function when exposed to the outside world and the improvisations possible to improve the quality of the product.

4.3 Deploying the application as a LoadBalancer type using Google Cloud

The Cluster IP service type is the default abstraction that Kubernetes provides. To deploy the application to the outside world so that all the computing resources are managed perfectly and distributed to facilitate balanced workloads, the best service type available is the 'LoadBalancer' that deploys the application externally and monitors and scales the application at the same time. The LoadBalancer services distribute the workload among multiple servers instead of a priority basis, making it efficient and resilient to system crashes. For example, in Figure 13, the virtual IP address is run through a LoadBalancer where it segregates the load to multiple servers, as shown, which can be adjusted based on scaling the resources. This approach protects the application and makes sure that the heavy traffic doesn’t hinder the application's efficiency or functioning. External cloud services manage the load balancer, and for this purpose, I have used the Google cloud portal (GCP)

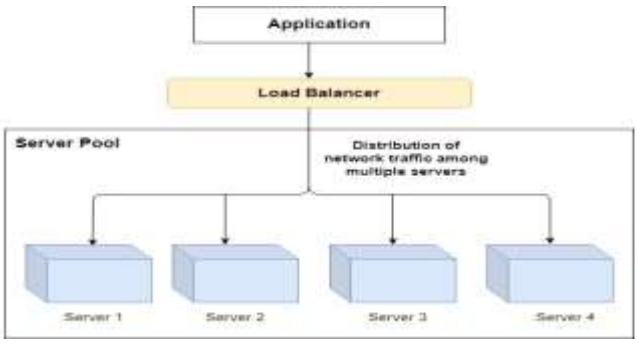
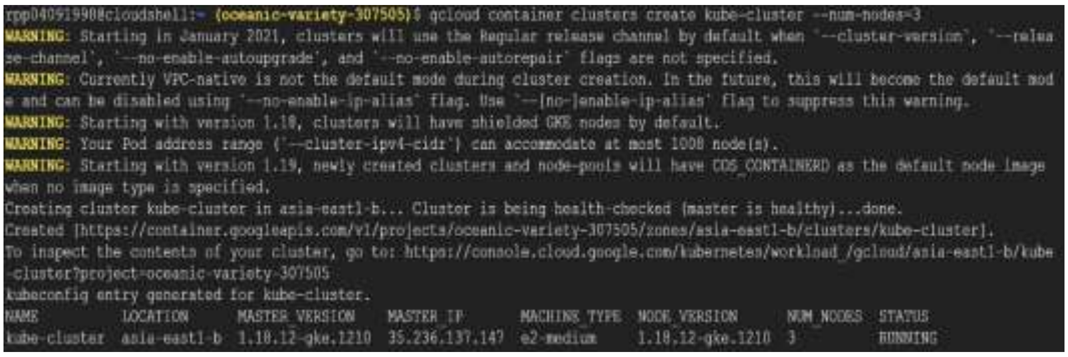


Figure – 13: LoadBalancer workflow and functionality using server pool

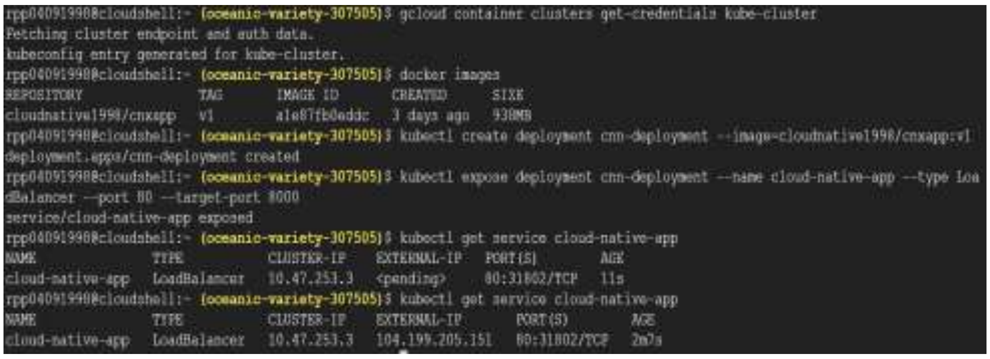
Google Cloud, like Docker, has its repository called 'Container Registry,' which is very similar to the Docker hub repository. Developers can build an image and push it to the registry or pull an image from the Docker hub itself. The deployment process is the same as performing the deployment done in the dashboard, except that the application is exposed to the outside world, and the access is universal and not within the cluster. Google Cloud also has a 'Kubernetes engine' that uses Kubernetes in the cloud premises. The first step is to create a cluster using the **'gcloud containers create**.' In Figure 14, the ‘Kube-Cluster' is created with three node pools and the 'US-west' datacentre region of the cloud as depicted in Figure 14 and is managed entirely by the cloud resources. The number of nodes can be specified when creating the cluster with the attribute '**--node**' (specified as 3). The details of the node pools are displayed when the command **'kubectl get nodes**' is used, which tells whether the nodes are active to function or not. The next step is to pull the 'cloudnative/cnxapp: v1’ image into the cloud environment using the Docker pull command to store the image locally.



```
rpp04091998@cloudshell:~ (oceanic-variety-307505)$ gcloud container clusters create kube-cluster --num-nodes=3
WARNING: Starting in January 2021, clusters will use the Regular release channel by default when `--cluster-version`, `--relea
se-channel`, `--no-enable-autoupgrade`, and `--no-enable-autorepair` flags are not specified.
WARNING: Currently VPC-native is not the default mode during cluster creation. In the future, this will become the default mod
e and can be disabled using `--no-enable-ip-alias` flag. Use `--[no-]enable-ip-alias` flag to suppress this warning.
WARNING: Starting with version 1.18, clusters will have shielded GKE nodes by default.
WARNING: Your Pod address range (`--cluster-ipv4-cidr`) can accommodate at most 1008 node(s).
WARNING: Starting with version 1.19, newly created clusters and node-pools will have COS_CONTAINERD as the default node image
when no image type is specified.
Creating cluster kube-cluster in asia-east1-b... Cluster is being health-checked (master is healthy)...done.
Created [https://container.googleapis.com/v1/projects/oceanic-variety-307505/zones/asia-east1-b/clusters/kube-cluster].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_/gcloud/asia-east1-b/kube
-cluster?project=oceanic-variety-307505
kubeconfig entry generated for kube-cluster.
NAME          LOCATION      MASTER_VERSION    MASTER_IP       MACHINE_TYPE  NODE_VERSION      NUM_NODES  STATUS
kube-cluster  asia-east1-b  1.18.12-gke.1210  35.236.137.147  e2-medium     1.18.12-gke.1210  3          RUNNING
```

Figure – 14: Google Cloud: creating an active Kubernetes cluster ‘Kube-cluster’ with three nodes

The next phase of deployment relies on the container image pulled from the Docker repository. It contains all the source code, dependencies, and instructions on running the application (Dockerfile). The next primary step is to generate the credentials for the cluster because when the deployment is done, the kubectl (Kubernetes command-line-tool) should recognize under which cluster the deployment is done, which is clearly shown in Figure 15. The 'kubectl create deployment' command takes the image file and creates the deployment. The final step of the deployment is using the expose command to deploy the application of a LoadBalancer type (specified by --type) with the target port and the cluster port. The deployment is finally done. This process completes the deployment where our application is deployed with the external-IP '**104.199.205.151**.'



```
rpp040919980cloudshell:~ (oceanic-variety-307505)$ gcloud container clusters get-credentials kube-cluster
Fetching cluster endpoint and auth data.
kubeconfig entry generated for kube-cluster.
rpp040919980cloudshell:~ (oceanic-variety-307505)$ docker images
REPOSITORY               TAG       IMAGE ID       CREATED      SIZE
cloudnative1998/cnxapp   v1        a1e87fb0eddc   3 days ago   938MB
rpp040919980cloudshell:~ (oceanic-variety-307505)$ kubectl create deployment cnn-deployment --image=cloudnative1998/cnxapp:v1
deployment.apps/cnn-deployment created
rpp040919980cloudshell:~ (oceanic-variety-307505)$ kubectl expose deployment cnn-deployment --name cloud-native-app --type Loa
dBalancer --port 80 --target-port 8000
service/cloud-native-app exposed
rpp040919980cloudshell:~ (oceanic-variety-307505)$ kubectl get service cloud-native-app
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
cloud-native-app   LoadBalancer   10.47.253.3   <pending>     80:31802/TCP   11s
rpp040919980cloudshell:~ (oceanic-variety-307505)$ kubectl get service cloud-native-app
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP       PORT(S)        AGE
cloud-native-app   LoadBalancer   10.47.253.3   104.199.205.151   80:31802/TCP   2m7s
```

Figure – 15: Google Cloud: pulling image and constructing deployment and exposing it with service type "LoadBalancer"

The external-IP enables the application to function beyond the local network region, i.e., with the external-IP, the product is now accessible from the internet. The LoadBalancer is not free of cost, and the cloud service itself manages it. The cost is calculated based on the usage and the resources computation and other added features such as scaling and the number of servers used. The external IP is '104.199.205.151', which is generated by the Google cloud engine itself. As shown in Figure 16, the deployment is successful, and I can access the external-IP application externally.

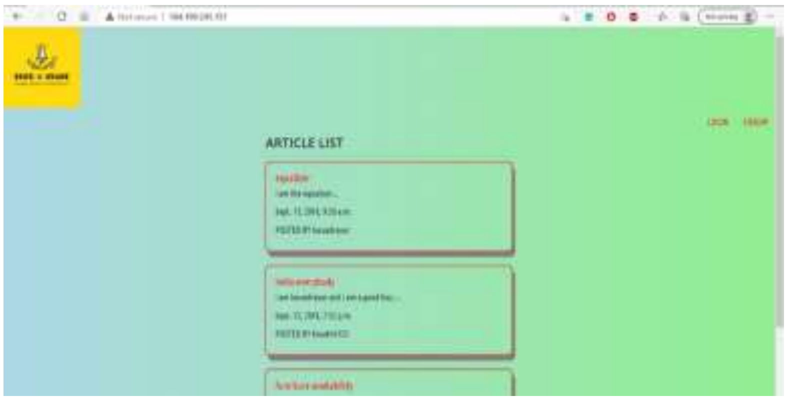
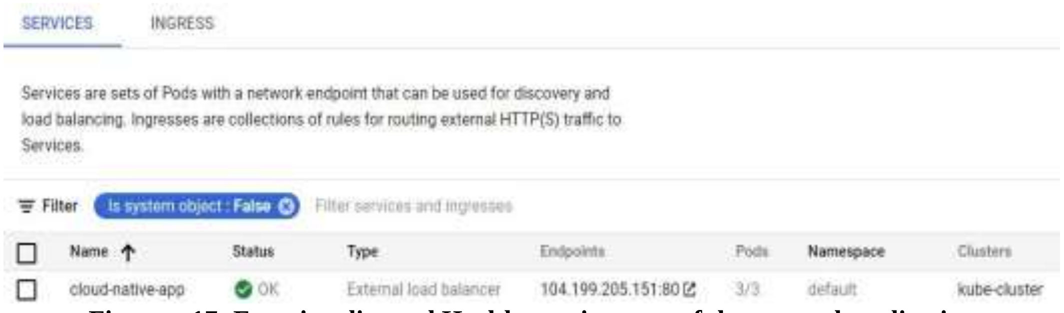


Figure – 16: Accessing the fully exposed application using external-IP ('104.199.205.151')

4.4 Scaling and monitoring the application

After the deployment is completed and the application is exposed, it is essential to run diagnostics to monitor the functioning of the application. For example, although the application is deployed and the external-IP is working, if the node pools go inactive (or) the control plane inside the clusters stops managing the network ports due to external interruptions, it might crash the application. Google cloud provides the best diagnostic engines to monitor the health of the application. In Figure 17, the workloads and the application's services (deployment and the exposed application) are functioning healthily. These status attributes turn red as soon as the application experiences an error.



SERVICES INGRESS

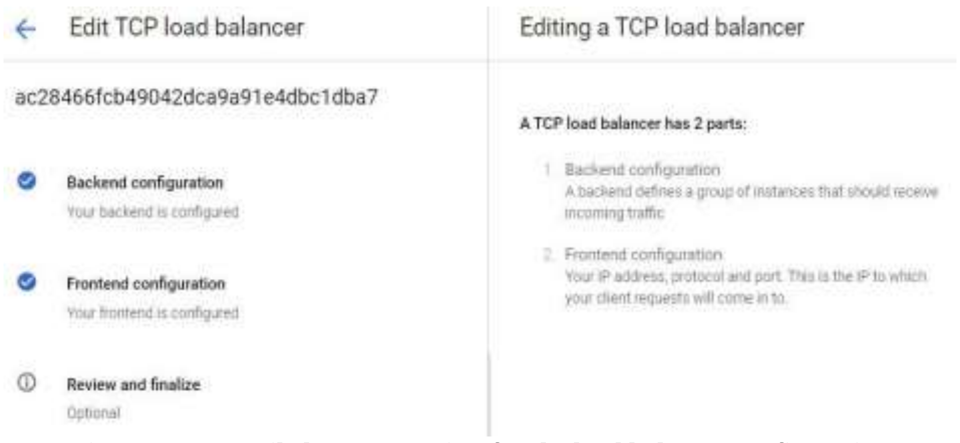
Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Filter | Is system object : False | Filter services and ingresses

☐	Name ↑	Status	Type	Endpoints	Pods	Namespace	Clusters
☐	cloud-native-app	✓ OK	External load balancer	104.199.205.151:80	3/3	default	kube-cluster

Figure – 17: Functionality and Health metric status of the exposed application

The external Load Balancer service is used to deploy the application, but it still needs to be configured, including scaling and resource usage. This process is of the highest priority as scaling has a significant role in improving the quality of the application. The Load Balancer needs to be configured in two sections: the frontend and the backend. In Figure 18, the Load Balancer configuration is explained. The frontend configuration takes care of the Internal IPs and the ports, whereas the backend configuration manages the resources.



Edit TCP load balancer

ac28466fcb49042dca9a91e4dbc1dba7

Backend configuration
Your backend is configured

Frontend configuration
Your frontend is configured

Review and finalize
Optional

Editing a TCP load balancer

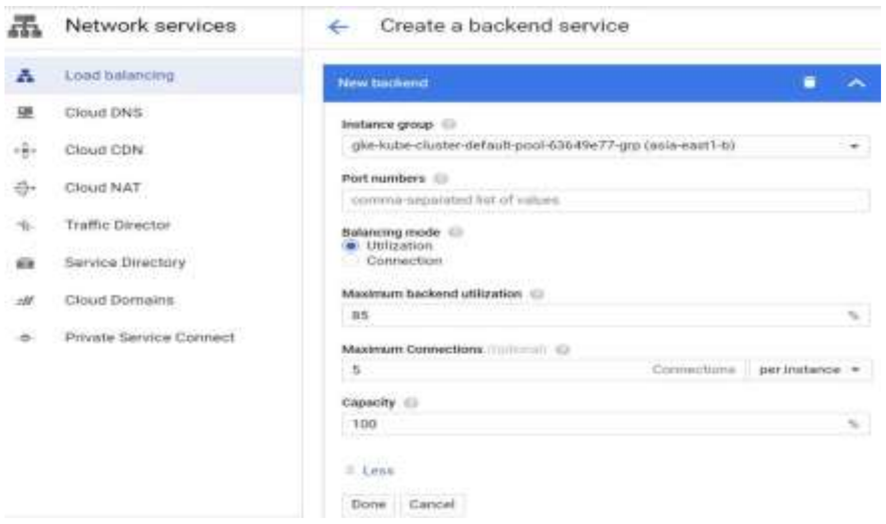
A TCP load balancer has 2 parts:

1. Backend configuration
A backend defines a group of instances that should receive incoming traffic

2. Frontend configuration
Your IP address, protocol and port. This is the IP to which your client requests will come in to.

Figure – 18: Detailed process review for the load balancer configuration

The frontend configuration is done automatically by the cloud server using the cluster details stored in the Kubernetes engine and the external IP. The protocol and the IP address, the external-IP, are taken for the configuration and the node pools. The backend configuration is very crucial. As shown in Figure 19, the backend configuration scales the application, the instance group (the cluster), and the ports are taken as the parameters on which the scaling process would implement. The maximum CPU utilization is set to 85%, which is the threshold value, and the number of server connections is also set to 5 per instance (server). This configuration is adaptive and can be changed anytime, depending on the workload sustained by the application. The biggest advantage of this setting is that auto-scaling is active, i.e., the application manages its resources automatically under the limit of these threshold values from the usage of CPU to the servers. These features make the Load Balancer very powerful because the distribution of incoming requests is done, so the costs incurred and the resources consumed for every response are minimum.



Create a backend service

New backend

Instance group
gke-kube-cluster-default-pool-63649e77-grp (asia-east1-b)

Port numbers
comma-separated list of values

Balancing mode
[✓ Utilization]
Connection

Maximum backend utilization
85 %

Maximum Connections (Optional)
5 Connections per instance

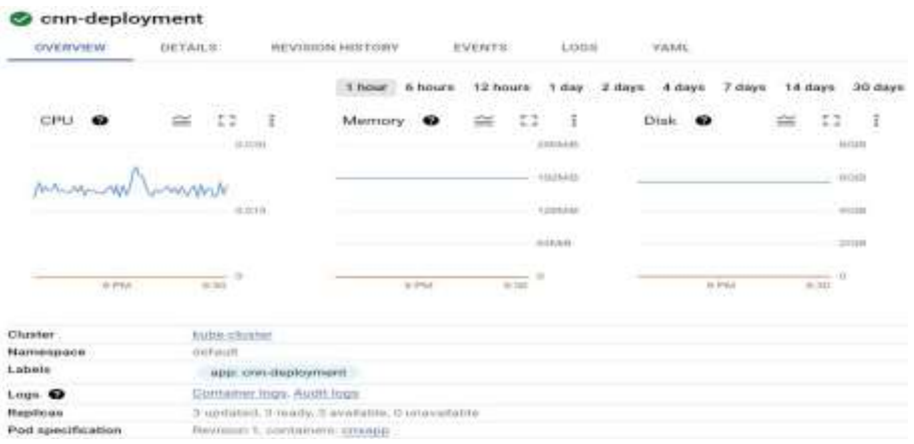
Capacity
100 %

Less

Done Cancel

Figure – 19: LoadBalancer backend configuration module

Dashboards and log systems are used in full scale to monitor every component of the application. For example, in figure 20, the resource statistics of the deployment and the exposed application are monitored in graphs where CPU, memory, and storage are displayed with multiple timestamp attributes. Every resource is constantly monitored with relevant recorded events and Audit logs. The cluster functions status, including network routing, authorizations is recorded every second in logs. These diagnostics provide a remarkable opportunity to scrutinize the working of every instance and augment their working to provide the highest level of quality to the customer.



Cluster kube-cluster
Namespace default
Labels app: cnn-deployment
Logs Container logs, Audit logs
Replicas 3 updated, 3 ready, 3 available, 0 unavailable
Pod specification Revision 1, containers: cnnapp

Figure – 20: Performance and resources monitoring statistics using Google cloud dashboard

5. CONCLUSION

In this paper, I have argued that with drastic improvements in technology and the increasing need to achieve the best performance and functional flexibility with the least cost, the cloud-native approach is considered the big trend in the modern era of cloud computing. Microservices, DevOps, Containers, and Continuous Integration are the significant components of cloud-native applications. The idea of breaking down an application into smaller and more compatible modules has given excellent results. Microservices and Containers have made the task of managing vast amounts of source code and deployments so robust and straightforward. Continuous integration integrates and deploys small pieces of code at regular intervals to central repositories, enhancing the entire development process and increasing the chances of recognizing complex errors quickly, contributing to the effort. I have also discussed how I constructed the application and divided the process into two parts: designing the application and then deploying the application. The mainframe application is a blog website using Django and Python. The title page is created first, followed by the account and the article models. Unlike monolithic architecture, Django divides the entire application in the form of templates. Each section of the application is constructed individually and their own allocated database and static files (CSS, JavaScript), which abides to the principle of microservices. I have also explained how to test the application through the local server. In conclusion, the cloud-native approach ensures better performance, security, and management. Docker is the best tool for containerizing the application, developing customized container images with robust security protocols. Jenkins provides the best secured CI pipeline to automate the entire process, and the workflow is followed by creating a deployment and exposing it using Kubernetes and Google cloud platform. Kubernetes makes use of worker nodes and clusters to bind the application and deploy it externally. In this paper, I have stressed the importance of services such as Cluster IP and Load-Balancer, which play a pivotal role in conserving resources and improving the performances of the applications. We can use Kubernetes UI to troubleshoot and manage the resources and operate from the local server itself. A detailed analysis of how the load-balancer works drive heavy traffic throughout the internet and the servers. One of the most vital parts of a cloud-native application is its ability to handle the resources optimally in different situations; that's where scalability comes in. We can scale the resources horizontally (increasing the number of server instances) and vertically (increasing the resources), which helps set up the threshold values for CPU levels, server storage, number of connections, and performance auto-scaling.

6. REFERENCES

[1] Kubernetes. 2021. Docker desktop UI, Containers, components, clusters, nodes, pods, deployments, networking diagnostics. Retrieved from https://kubernetes.io/docs/tutorials/kubernetes-basics/.

[2] Kubernetes. 2020. Deploying services to clusters and exposing them to the outside world, external load balancer and introduction to scaling computing resources. Retrieved from https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/.

[3] Docker. Documentation regarding image pull and deploying on Kubernetes and dashboard features along with 'kubectl.' Retrieved from https://docs.docker.com/desktop/kubernetes/.

[4] Django. 2021. Services and modules included Django-microservices from Pypi, project description and available files for access and downloads. Retrieved from https://pypi.org/project/django-microservices/.

[5] Dbader. 2019. GitHub repository: Django development with Docker compose and Machine (including blog articles and operating system Instructions). Retrieved May 28, 2021 from https://github.com/realpython/dockerizing-django.

[6] Django. Installation guide, modules, writing the first basic Django app, database migrations and server testing and debug. Retrieved from https://www.djangoproject.com/start/.

[7] Google Cloud. 2021. Deployments and pod templates (specifications) in GKE clusters and description of their use in Google Kubernetes engine. Retrieved from https://cloud.google.com/kubernetes-engine/docs/concepts/deployment.

[8] Django. Writing and managing static files like HTML, CSS, and JavaScript in Django applications with migrations, configurations and testing with deployment. Retrieved from https://docs.djangoproject.com/en/3.2/howto/static-files/.

[9] W3schools. Bootstrap 4 concepts and tutorials and grids and references, including basics and templates. Retrieved from https://www.w3schools.com/bootstrap4/bootstrap_ref_all_classes.asp.

[10] Pramod. Working with Cluster IP service type in Kubernetes with basics and workflow. (March 24). Retrieved May 28, 2021 (https://medium.com/the-programmer/working-with-clusterip-service-type-in-kubernetes-45f2c01a89c8).

[11] Jenkins. Prerequisites: defining a pipeline through Jenkins Classic UI, managing Jenkins, scaling, administration, syntax. Retrieved from https://www.jenkins.io/doc/book/pipeline/getting-started/.

[12] Ernest Mueller. 2019. DevOps: Introduction, History, Characteristics, Use in the modern world, DevOps references and reading list by the agile admin. (January 2019). Retrieved May 28, 2021 from https://theagileadmin.com/what-is-DevOps/.

[13] Phil Wittmer. 2020. Monolithic architecture vs Microservices architecture: In-depth study and analysis. (March 2020). Retrieved May 28, 2021 from https://www.tiempodev.com/blog/monolithic-vs-microservices-architecture/.

[14] Nginx. Load-balancing: Definition, Use in the modern cloud, types, advantages, load balancing in cluster environments. Retrieved from https://www.nginx.com/resources/glossary/load-balancing/.