Building an End-to-End DevOps Pipeline with Cloud-Hosted Web Microservices

Chinmay Haval 1*, Kartik Jagtap 2†, Mayank Kumar 3‡,
Milind Murmu 4§, Nikhil Baghel 5¶, Rahul Singh 6‖, Saksham 7**,
Pranjal Shinde 8††
*21bds014 1, DSAI
†21bds025 2, DSAI
‡21bds037 3, DSAI
§21bds038 4, DSAI
¶21bds044 5, DSAI
‖21bds054 6, DSAI
**21bds058 7, DSAI
††21bds062 8, DSAI

***Abstract*—In this project, we aim to build a comprehensive DevOps pipeline that integrates key tools and platforms, including Docker, Jenkins, Kubernetes, and cloud services. This pipeline is designed to automate and streamline the development, testing, and deployment of a weather forecasting web application built with web microservices. By establishing a fully automated workflow, we seek to enhance the reliability and scalability of the application, ensuring that it is continuously deployable and accessible to users. Additionally, this project serves as a learning experience to understand the practical aspects of creating and maintaining a robust end-to-end pipeline, with a focus on testing, monitoring, and adapting DevOps best practices for improved user availability and service continuity.**

I. Introduction

In this project, we aim to develop an end-to-end DevOps pipeline, leveraging a combination of web frontend and backend microservices to deliver a functional weather forecasting application. This pipeline utilizes Docker, Jenkins, Kubernetes, and AWS for seamless development, deployment, and cloud hosting, demonstrating the full lifecycle of a modern software product. Through this project, we intend to gain hands-on experience with DevOps methodologies and the essential tools for code versioning, containerization, continuous integration, orchestration, and scalable cloud deployment.

To begin, we focused on Microservices Development and Dockerization. This step involved identifying distinct functionalities of the application to divide into self-contained microservices. For this project, at least two microservices were designed, such as User Service and Order Service, each responsible for a specific function within the overall application. By separating these services, we could isolate and manage each component independently, making development and maintenance more efficient. For each microservice, we created a dedicated Dockerfile to facilitate containerization, which allows each service to run consistently across different environments. The Dockerfiles defined the necessary setup instructions, and we built Docker images for each microservice. These images were then pushed to a container registry, such as Docker Hub, providing a central repository for accessible and deployable service images. Once the microservices were containerized, we

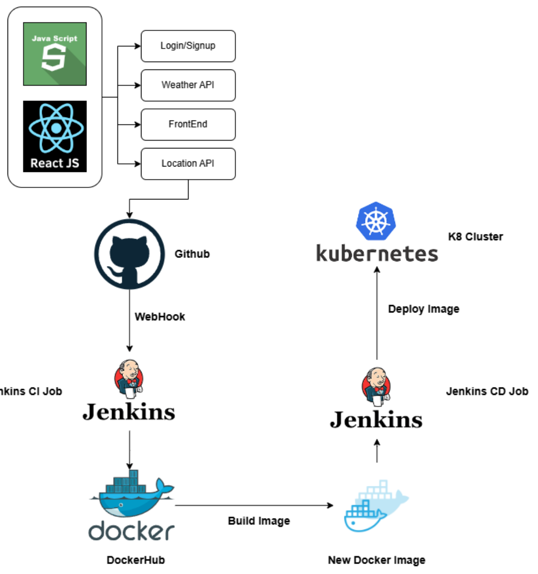


Fig. 1: Devops Framework

moved to Jenkins Setup to establish continuous integration and continuous delivery (CI/CD). Jenkins, an automation server, was installed on a Linux machine (or accessed via a cloud-based Jenkins service) and configured to connect with our GitHub repository. This setup enabled automated builds and testing every time code changes were committed, ensuring that new updates are reliably tested and integrated. Within Jenkins, we created a CI/CD pipeline that triggers build processes on every code commit to the GitHub repository, building the application, running tests, and generating Docker images. In the deployment stage, Jenkins was set up to deploy these Docker images to a cloud environment, allowing automated and consistent deployment with every successful build. Custom scripts were also developed to manage the orchestration of Docker containers, enabling smooth coordination between services.

For orchestration and scaling, Kubernetes was used to deploy and manage the application's containers on AWS. Kubernetes provides advanced features such as load balancing, automatic scaling, and self-healing, which are essential for maintaining a stable and scalable application in production. AWS was chosen as the cloud platform for its reliability and scalability, hosting our Kubernetes clusters and making the application accessible to users worldwide.

Through this project, we learned to coordinate and deploy an end-to-end DevOps pipeline and gained valuable experience in several critical areas. These include maintaining code within GitHub, creating and managing Docker containers, automating build and deployment workflows with Jenkins, and using Kubernetes for orchestration and AWS for scalable cloud hosting. This pipeline not only supports the weather forecasting web application but also serves as a comprehensive framework for managing future microservice-based applications effectively.

II. Methods and Frameworks Used

This section outlines the various methods and frameworks employed in the development, deployment, and continuous integration of the system.

A. Microservices

1) Folder Structure

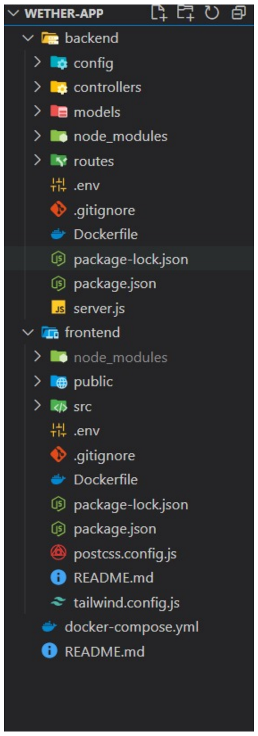
The initial folder structure for our microservices architecture is designed to organize the frontend, backend service (Node.js), and API calls. The structure is outlined as shown in Image 1 below.

2) Front End

Frontend Service: A user friendly interface that displays weather data, retrieved from a third-party weather forecast API.

3) Backend Service

A user authentication and management system supporting login and registration, developed using Node.js. Both components are containerized and managed via Docker, with deployment and orchestration handled by a Jenkins-managed CI/CD pipeline. The backend microservice ensures secure user data handling and serves as the authentication layer. Each microservice is designed with independent functionalities for scalability and easier maintenance.



```
WETHER-APP
  backend
    config
    controllers
    models
    node_modules
    routes
    .env
    .gitignore
    Dockerfile
    package-lock.json
    package.json
    server.js
  frontend
    node_modules
    public
    src
    .env
    .gitignore
    Dockerfile
    package-lock.json
    package.json
    postcss.config.js
    README.md
    tailwind.config.js
  docker-compose.yml
  README.md
```

Fig. 2: Folder structure of the microservices, including frontend, backend service (Node.js), and API calls.

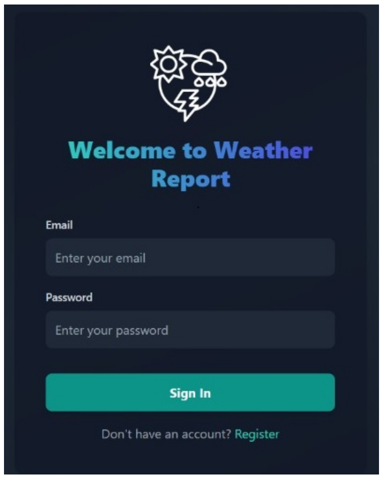


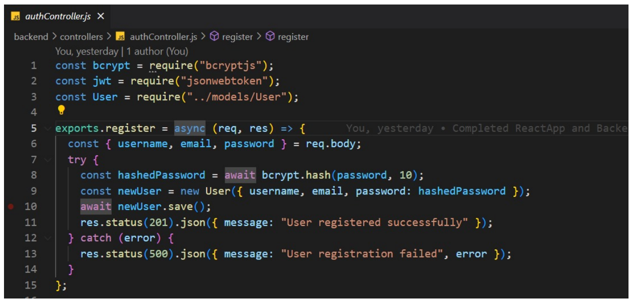
Fig. 3: FrontEnd webapp initial look

4) API Call and Usage

”We utilized the third-party API service, OpenWeatherMap, to predict the weather forecast. An illustration of how the API is integrated into the system is provided below.”

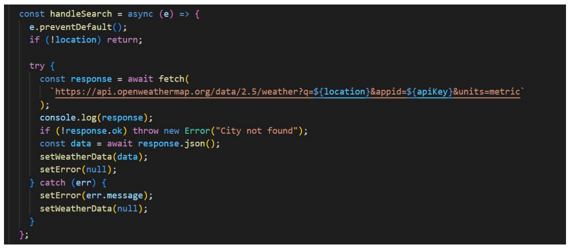
III. Docker for Containerization

Docker is a platform that packages applications and their dependencies into containers, ensuring consistent performance across environments. In our project, Docker was used to containerize both the frontend web application


authController.js
backend > controllers > authController.js > register > register
You, yesterday | 1 author (You)
You, yesterday • Completed ReactApp and Backe

```javascript
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const User = require("../models/User");

exports.register = async (req, res) => {
  const { username, email, password } = req.body;
  try {
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();
    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: "User registration failed", error });
  }
};
```

Fig. 4: Backend service login



```javascript
const handleSearch = async (e) => {
  e.preventDefault();
  if (!location) return;

  try {
    const response = await fetch(
      `https://api.openweathermap.org/data/2.5/weather?q=${location}&appid=${apiKey}&units=metric`
    );
    console.log(response);
    if (!response.ok) throw new Error("City not found");
    const data = await response.json();
    setWeatherData(data);
    setError(null);
  } catch (err) {
    setError(err.message);
    setWeatherData(null);
  }
};
```

Fig. 5: api use frontend

and the backend microservice, along with the API calls, providing isolated and consistent environments for each component.

We used Docker Compose to efficiently manage the multi-container setup, simplifying orchestration and communication between the containers. It linked the frontend and backend services, enabling smooth integration. Docker images were built for both services, containerizing our web app for scalability, portability, and easy deployment.

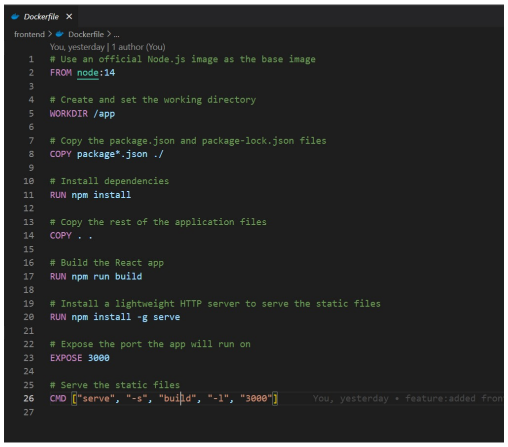
A. Dockerfile for Frontend Microservice

The Dockerfile starts with a Node.js 14 image as the base. It sets a working directory at /app and copies the necessary package.json files to install dependencies using npm install. After installing dependencies, it copies the app files, builds the React app using npm run build, and installs the serve package to serve static files. The Dockerfile exposes port 3000 and runs the app using serve, ensuring it functions properly in a containerized environment.

B. DockerFile for Backend Microservice

The Dockerfile for the backend service starts by selecting the Node.js version 16 image as the base environment. It then creates a working directory at /app to encapsulate all operations. The Dockerfile proceeds to copy the package.json and package-lock.json files into the container, which are crucial for dependency management, followed by running npm install to install those dependencies.

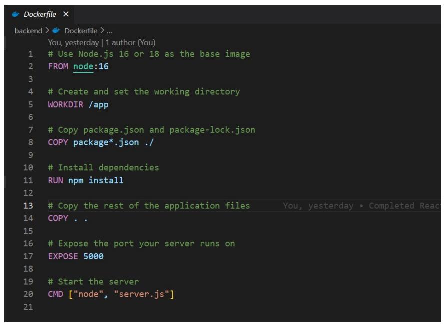
Afterward, it copies all remaining application files to ensure that the complete application code is present in


Dockerfile
frontend > Dockerfile > ...
You, yesterday | 1 author (You)
You, yesterday • feature:added fron

```dockerfile
# Use an official Node.js image as the base image
FROM node:14

# Create and set the working directory
WORKDIR /app

# Copy the package.json and package-lock.json files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files
COPY . .

# Build the React app
RUN npm run build

# Install a lightweight HTTP server to serve the static files
RUN npm install -g serve

# Expose the port the app will run on
EXPOSE 3000

# Serve the static files
CMD ["serve", "-s", "build", "-l", "3000"]
```

Fig. 6: Docker file for frontend

the container. The Dockerfile also exposes port 5000, indicating where the application will listen for incoming connections. Finally, it sets the command to start the server by executing node server.js, ensuring the backend application is ready to run in a containerized setting.


Dockerfile
backend > Dockerfile > ...
You, yesterday | 1 author (You)
You, yesterday • Completed Reac

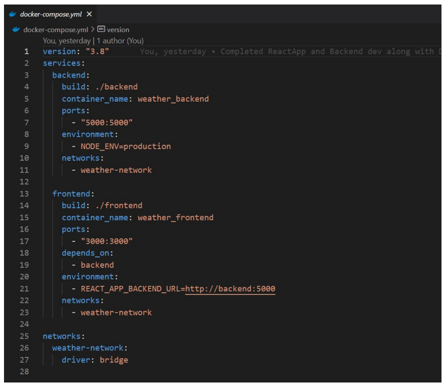
```dockerfile
# Use Node.js 16 or 18 as the base image
FROM node:16

# Create and set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application files
COPY . .

# Expose the port your server runs on
EXPOSE 5000

# Start the server
CMD ["node", "server.js"]
```

Fig. 7: DockerFile for backend

C. Docker Compose

The docker-compose.yml file orchestrates the deployment of both the frontend and backend services for an application. It defines the backend service to build from the ./backend directory, listen on port 5000, and operate in production mode while connected to a weather-network for internal communication. The frontend service is configured to build from the ./frontend directory, run on port 3000, and depend on the backend service, which ensures it starts afterward. An environment variable, react-app-backend-url, is set to facilitate API interaction with the backend. Both services share the same network, promoting seamless communication while maintaining isolation from other host services. The network uses a bridge driver, enhancing connectivity and security within the Docker environment.

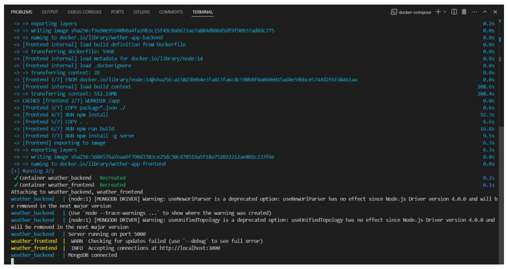
After executing the docker-compose up –build command, all microservices: both the frontend and the backend are


docker-compose.yml ×
docker-compose.yml > version
You, yesterday | 1 author (You)

```
version: "3.8"      You, yesterday • Completed ReactApp and Backend dev along with D
services:
  backend:
    build: ./backend
    container_name: weather_backend
    ports:
      - "5000:5000"
    environment:
      - NODE_ENV=production
    networks:
      - weather-network

  frontend:
    build: ./frontend
    container_name: weather_frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:
      - REACT_APP_BACKEND_URL=http://backend:5000
    networks:
      - weather-network

networks:
  weather-network:
    driver: bridge
```

Fig. 8: Docker Compose

launched simultaneously. This brings the web application online on the localhost, as shown in the figure below


PROBLEMS OUTPUT DEBUG CONSOLE PORTS GITLENS COMMENTS TERMINAL   docker-compose

```
 => => exporting layers                                                                       0.2s
 => => writing image sha256:f9a90e95940b0a4fa39b3c15f49c0ab673ae7a884d886d5df9f90937ad8dc775   0.0s
 => => naming to docker.io/library/wether-app-backend                                         0.0s
 => [frontend internal] load build definition from Dockerfile                                 0.0s
 => => transferring dockerfile: 596B                                                          0.0s
 => [frontend internal] load metadata for docker.io/library/node:14                           0.8s
 => [frontend internal] load .dockerignore                                                    0.0s
 => => transferring context: 2B                                                               0.0s
 => [frontend 1/7] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa  0.0s
 => [frontend internal] load build context                                                  208.6s
 => => transferring context: 512.19MB                                                       208.4s
 => CACHED [frontend 2/7] WORKDIR /app                                                        0.0s
 => [frontend 3/7] COPY package*.json ./                                                      0.6s
 => [frontend 4/7] RUN npm install                                                           92.3s
 => [frontend 5/7] COPY . .                                                                   6.6s
 => [frontend 6/7] RUN npm run build                                                         16.8s
 => [frontend 7/7] RUN npm install -g serve                                                   9.5s
 => [frontend] exporting to image                                                             6.3s
 => => exporting layers                                                                       6.3s
 => => writing image sha256:5606576a7eaa9f790d3383ce25dc50c478519a5f18a752022212ae802c233f6e   0.0s
 => => naming to docker.io/library/wether-app-frontend                                        0.0s
[+] Running 2/2
 ✓Container weather_backend   Recreated                                                       0.2s
 ✓Container weather_frontend  Recreated                                                       0.1s
Attaching to weather_backend, weather_frontend
weather_backend   | (node:1) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.js Driver version 4.0.0 and will b
e removed in the next major version
weather_backend   | (Use `node --trace-warnings ...` to show where the warning was created)
weather_backend   | (node:1) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and
will be removed in the next major version
weather_backend   | Server running on port 5000
weather_frontend  |  WARN  Checking for updates failed (use `--debug` to see full error)
weather_frontend  |  INFO  Accepting connections at http://localhost:3000
weather_backend   | MongoDB connected
```

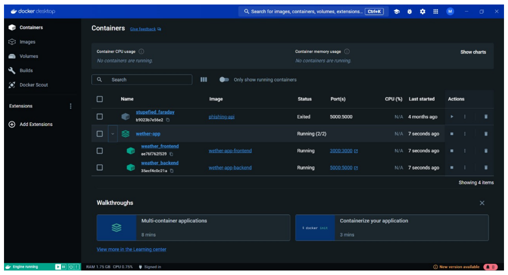
Fig. 9: Image Running after docker compose

D. Building the Docker Image

To build and run the Docker containers, we used the following commands:

- **Building the Docker Image:**
  - The following command builds the Docker image using the Dockerfile in the current directory and tags it with the specified image name:
  - `docker build -t <image-name> .`
- **Running the Docker Container:**
  - This command runs the built Docker image as a container and maps port 3000 of the container to port 3000 of the host machine:
  - `docker run -p 3000:3000 image`
- **Using Docker Compose:**
  - This command builds and starts the multi-container application defined in the `docker-compose.yml` file, managing both the frontend and backend services together:
  - `docker-compose up --build`

Now, the Docker image has been successfully built and is running, as shown in the figure below.



Containers

Container CPU usage: No containers are running.
Container memory usage: No containers are running.

Name	Image	Status	Port(s)	CPU (%)	Last started
stupefied_faraday b9023b7e56e2	phishing-api	Exited	5000:5000	N/A	4 months ago
wether-app		Running (2/2)		N/A	7 seconds ago
weather_frontend ae76f762f539	wether-app-frontend	Running	3000:3000	N/A	7 seconds ago
weather_backend 35ecf4c0c21a	wether-app-backend	Running	5000:5000	N/A	7 seconds ago

Showing 4 items

Fig. 10: Docker Image

This is whole Docker containerization with the microservices and now , we will move forward to the jenkins

IV. Jenkins Automated CI/CD Pipeline

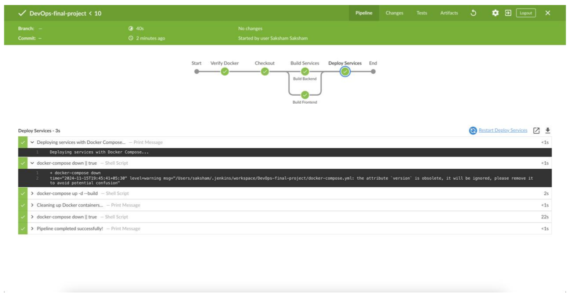
Jenkins played a pivotal role in automating the Continuous Integration and Continuous Deployment (CI/CD) process for the weather forecasting application. By leveraging Jenkins, we ensured that code changes were seamlessly integrated, tested, and deployed to production with minimal manual intervention. Below are the detailed steps and configurations:

A. Jenkins Setup

- Jenkins was installed on a Mac server to serve as the central automation tool. Alternatively, a Local Jenkins service was configured to facilitate easier access.
- The Jenkins instance was integrated with the GitHub repository to monitor code changes. Webhooks were set up to trigger automated pipelines upon every code commit or pull request.

B. CI/CD Pipeline Configuration

The core functionality of Jenkins in this project was the creation of a CI/CD pipeline, designed to automate the build, test, and deployment workflows.



Branch: –   40s   No changes
Commit: –   2 minutes ago   Started by user Saksham Saksham

Deploy Services - 3s

```
Deploying services with Docker Compose... — Print Message              <1s
    Deploying services with Docker Compose...
docker-compose down || true — Shell Script                             <1s
    + docker-compose down
    time="2024-11-15T19:45:41+05:30" level=warning msg="/Users/saksham/.jenkins/workspace/DevOps-final-project/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it
    to avoid potential confusion"
docker-compose up -d --build — Shell Script                            2s
Cleaning up Docker containers... — Print Message                       <1s
docker-compose down || true — Shell Script                             22s
Pipeline completed successfully! — Print Message                       <1s
```

Fig. 11: Jenkins Pipeline

- **Pipeline Stages:**

  1) **Build Stage:**
     - Jenkins pulled the latest code from the GitHub repository.

- Dockerfiles for each microservice (e.g., User Service, Order Service) were utilized to build container images.

2) **Test Stage:**
   - Automated tests were run to validate the functionality and performance of the code.
   - Only successful builds proceeded to the next stage.

3) **Deployment Stage:**
   - Docker images were pushed to Docker Hub for centralized storage.
   - Custom scripts within the Jenkins pipeline deployed the images to the Kubernetes cluster hosted on AWS.

- **Pipeline Script:** The Jenkinsfile for this project is configured to implement a robust CI/CD pipeline. It automates tasks like verifying Docker, cloning the repository, building services, and deploying the application.

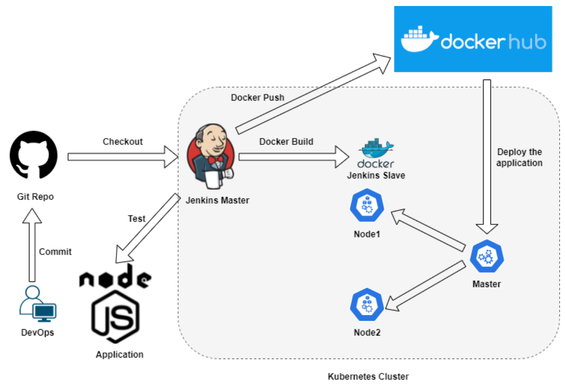


Fig. 12: Role Of Jenkins

C. Key Features Leveraged in Jenkins

- **Automated Triggers:** Jenkins pipelines were triggered automatically upon detecting code changes in the GitHub repository.
- **Integrated Testing:** Tests were incorporated into the pipeline to ensure the reliability of each build.
- **Scalability:** Jenkins pipelines were designed to handle multiple microservices simultaneously.
- **Custom Scripts:** Scripts were added to manage Docker container orchestration and Kubernetes deployments efficiently.

D. Advantages of Using Jenkins

- **Continuous Integration:** Automated builds ensured that the codebase remained stable and up-to-date with the latest changes.
- **Continuous Deployment:** Seamless deployment to Kubernetes enabled rapid delivery of new features and updates.
- **Error Detection:** Failures in the pipeline were immediately flagged, allowing for faster debugging and issue resolution.

V. Kubernetes for Orchestration

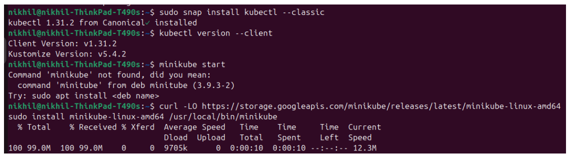
We have integrated Kubernetes into our CI/CD pipeline for orchestrating both front-end and back-end microservices. This integration enables automated scaling, improved deployment processes, and ensures high availability of our services. Kubernetes' ability to manage containers dynamically has led to reduced downtime, allowing seamless updates and continuous integration with minimal manual intervention. Notable advantages include dynamic resource scaling, enhanced reliability with automated failover mechanisms, and optimized resource utilization. By incorporating Kubernetes, we have accelerated our deployments, bolstered application resilience, and improved user experience stability. Below, we outline the step-by-step process for orchestration and exposing microservices using Kubernetes.

A. Setting Up Kubernetes with Minikube

Minikube is used to initialize a local Kubernetes cluster, essential for orchestrating microservices in a local environment. This section outlines the process for updating the system, installing dependencies, and setting up Minikube.

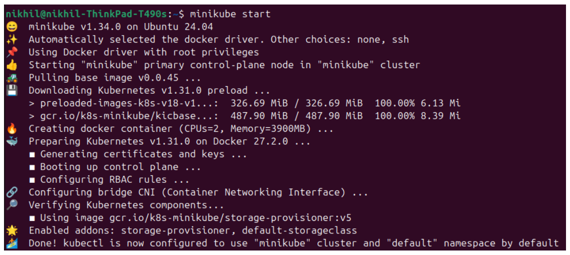
- **Update System Package List:** First, update the system package list to ensure that the latest available packages are installed using `sudo apt-get update`.
- **Install Dependencies:** Install the required dependencies for managing secure HTTP connections and handling Kubernetes-related packages: `sudo apt-get install -y apt-transport-https ca-certificates curl`.
- **Download and Install Minikube:** To install Minikube, run the following commands:
  - `curl -LO googleapi.com(i will ).`
  - `sudo install minikube-linux-amd64 /usr/local/bin/minikube`
- **Start Minikube:** Start the Minikube cluster with `minikube start`.

Images for Setup Steps:



```
nikhil@nikhil-ThinkPad-T490s:~$ sudo snap install kubectl --classic
kubectl 1.31.2 from Canonical✓ installed
nikhil@nikhil-ThinkPad-T490s:~$ kubectl version --client
Client Version: v1.31.2
Kustomize Version: v5.4.2
nikhil@nikhil-ThinkPad-T490s:~$ minikube start
Command 'minikube' not found, did you mean:
  command 'minitube' from deb minitube (3.9.3-2)
Try: sudo apt install <deb name>
nikhil@nikhil-ThinkPad-T490s:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 99.0M  100 99.0M    0     0  9705k      0  0:00:10  0:00:10 --:--:-- 12.3M
```

Fig. 13: Minikube Installation Process



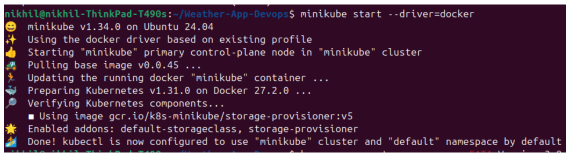
```
nikhil@nikhil-ThinkPad-T490s:~$ minikube start
😄  minikube v1.34.0 on Ubuntu 24.04
✨  Automatically selected the docker driver. Other choices: none, ssh
📌  Using Docker driver with root privileges
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.45 ...
💾  Downloading Kubernetes v1.31.0 preload ...
    > preloaded-images-k8s-v18-v1...:  326.69 MiB / 326.69 MiB  100.00% 6.13 Mi
    > gcr.io/k8s-minikube/kicbase...:  487.90 MiB / 487.90 MiB  100.00% 8.39 Mi
🔥  Creating docker container (CPUs=2, Memory=3900MB) ...
🐳  Preparing Kubernetes v1.31.0 on Docker 27.2.0 ...
    ■ Generating certificates and keys ...
    ■ Booting up control plane ...
    ■ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Fig. 14: Starting Minikube Cluster

B. Installing Kubernetes CLI Tool - kubectl

The Kubernetes Command-Line Interface (CLI), `kubectl`, is a vital tool for interacting with Kubernetes clusters. It allows us to manage services, pods, and other Kubernetes resources directly from the command line.

- **Installing kubectl:** Install `kubectl` using `sudo apt-get install -y kubectl` or `sudo snap install kubectl --classic`.
- **Verifying kubectl Installation:** Verify the installation with `kubectl version --client`.



```
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ minikube start --driver=docker
😄  minikube v1.34.0 on Ubuntu 24.04
✨  Using the docker driver based on existing profile
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.45 ...
🏃  Updating the running docker "minikube" container ...
🐳  Preparing Kubernetes v1.31.0 on Docker 27.2.0 ...
🔎  Verifying Kubernetes components...
    ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Fig. 15: Minikube Docker Starting Process

C. Installing and Configuring Kompose

Kompose simplifies the process of converting Docker Compose configurations into Kubernetes YAML files. This allows for easy deployment of both front-end and back-end services on Kubernetes.

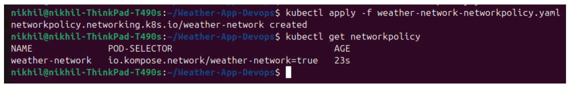
- **Installing Kompose:** To install Kompose, run:
  - `sudo chmod +x /usr/local/bin/kompose`
- **Checking Kompose Version:** Verify the version with `kompose version`.

D. Deploying Microservices with Kubernetes

After converting Docker Compose files into Kubernetes YAML configurations with Kompose, we can deploy the microservices to the Kubernetes cluster.

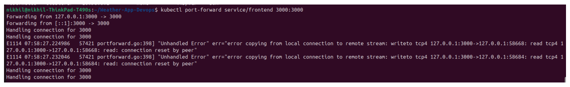
- **Cloning GitHub Repository:** Clone the GitHub repository for the application with `git clone git repo(i will do it)`, and change to the directory with `cd Weather-App-Devops`.
- **Converting Docker Compose to Kubernetes YAML:** Convert the Docker Compose configurations into Kubernetes YAML files with `kompose convert`.
- **Applying Kubernetes Configurations:** Deploy the services using the generated YAML files:
  - `kubectl apply -f frontend-deployment.yaml`
  - `kubectl apply -f backend-deployment.yaml`
  - `kubectl apply -f ingress.yaml`

Images for Deployment Steps:



```
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ kubectl apply -f weather-network-networkpolicy.yaml
networkpolicy.networking.k8s.io/weather-network created
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ kubectl get networkpolicy
NAME              POD-SELECTOR                                AGE
weather-network   io.kompose.network/weather-network=true    23s
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ 
```

Fig. 16: Converting to Kubernetes YAML Files



```
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ kubectl port-forward service/frontend 3000:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
Handling connection for 3000
Handling connection for 3000
E1114 07:58:27.224986   57421 portforward.go:398] "Unhandled Error" err="error copying from local connection to remote stream: writeto tcp4 127.0.0.1:3000->127.0.0.1:58668: read tcp4 1
27.0.0.1:3000->127.0.0.1:58668: read: connection reset by peer"
E1114 07:58:27.232046   57421 portforward.go:398] "Unhandled Error" err="error copying from local connection to remote stream: writeto tcp4 127.0.0.1:3000->127.0.0.1:58684: read tcp4 1
27.0.0.1:3000->127.0.0.1:58684: read: connection reset by peer"
Handling connection for 3000
Handling connection for 3000
```

Fig. 17: Applying the YAML Configurations

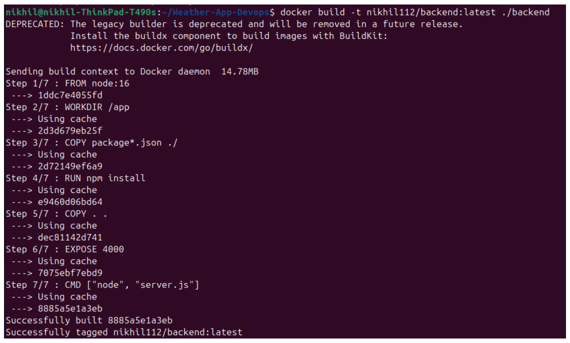
E. Pushing the Docker Backend and Frontend Microservices

Both the front-end and back-end microservices were containerized using Docker. These containerized services were then deployed into the Kubernetes cluster, providing a scalable and isolated runtime environment that supports seamless management, load balancing, and scaling within the CI/CD pipeline.

F. Monitoring and Exposing Services

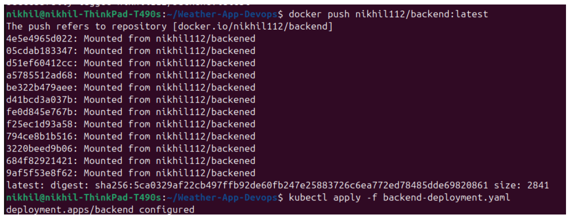
After deployment, we monitored the status of our services and exposed them using Minikube, making them accessible externally.

- **Checking Deployments and Services Status:** Check the status of deployments, services, and pods with:
  - `kubectl get deployments`
  - `kubectl get services`



```
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ docker build -t nikhil112/backend:latest ./backend
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  14.78MB
Step 1/7 : FROM node:16
 ---> 1ddc7e4055fd
Step 2/7 : WORKDIR /app
 ---> Using cache
 ---> 2d3d679eb25f
Step 3/7 : COPY package*.json ./
 ---> Using cache
 ---> 2d72149ef6a9
Step 4/7 : RUN npm install
 ---> Using cache
 ---> e9460d06bd64
Step 5/7 : COPY . .
 ---> Using cache
 ---> dec81142d741
Step 6/7 : EXPOSE 4000
 ---> Using cache
 ---> 7075ebf7ebd9
Step 7/7 : CMD ["node", "server.js"]
 ---> Using cache
 ---> 8885a5e1a3eb
Successfully built 8885a5e1a3eb
Successfully tagged nikhil112/backend:latest
```

Fig. 18: Building the Docker Backend Service

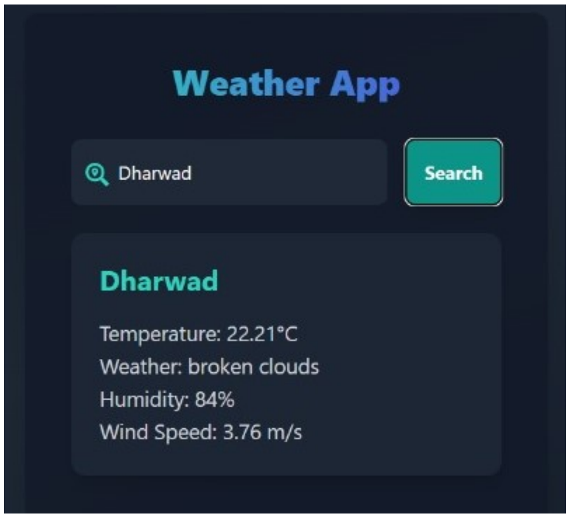


```
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ docker push nikhil112/backend:latest
The push refers to repository [docker.io/nikhil112/backend]
4e5e4965d022: Mounted from nikhil112/backened
05cdab183347: Mounted from nikhil112/backened
d51ef60412cc: Mounted from nikhil112/backened
a5785512ad68: Mounted from nikhil112/backened
be322b479aee: Mounted from nikhil112/backened
d41bcd3a037b: Mounted from nikhil112/backened
fe0d845e767b: Mounted from nikhil112/backened
f25ec1d93a58: Mounted from nikhil112/backened
794ce8b1b516: Mounted from nikhil112/backened
3220beed9b06: Mounted from nikhil112/backened
684f82921421: Mounted from nikhil112/backened
9af5f53e8f62: Mounted from nikhil112/backened
latest: digest: sha256:5ca0329af22cb497ffb92de60fb247e25883726c6ea772ed78485dde69820861 size: 2841
nikhil@nikhil-ThinkPad-T490s:~/Weather-App-Devops$ kubectl apply -f backend-deployment.yaml
deployment.apps/backend configured
```

Fig. 19: Pushing the Backend Service to Kubernetes

  - `kubectl get pods`
- **Exposing Services Using Minikube:** Expose the services and make them externally accessible with:
  - `minikube service frontend-service`
  - `minikube service backend-service`

Images for Monitoring and Exposing Services:



Dharwad

Temperature: 22.21°C
Weather: broken clouds
Humidity: 84%
Wind Speed: 3.76 m/s

Fig. 20: Web microservice successful testing

VI. Teammates Contribution and Learning

In this project, we have successfully learned how to create and manage an end-to-end DevOps pipeline, integrating various modern tools and technologies. Starting with version control using GitHub, we progressed to Docker containerization for the backend and microservices. This was followed by setting up Jenkins for CI/CD automation and leveraging Kubernetes for orchestration and deployment. Additionally, we gained hands-on experience in exposing web frontends, managing backend services, and efficiently handling API calls. This comprehensive approach has enabled us to develop a robust understanding of DevOps practices and their practical implementation, marking a significant step forward in our technical learning journey.

- **Chinmay Haval and Kartik Jagtap:** Managed the entire end-to-end backend microservice development and API integration, enhancing their expertise in microservice architecture and API management.
- **Mayank Kumar and Rahul Singh:** Handled the Docker containerization process and successfully pushed the Docker image, acquiring valuable experience in containerization and image management.
- **Milind Murmu and Pranjal Shinde:** Set up the Jenkins CI/CD pipeline and automated deployment workflows, strengthening their skills in continuous integration, delivery, and automation.
- **Nikhil Baghel and Saksham:** Worked on Kubernetes orchestration and deployment on MiniKube, learning the intricacies of container orchestration and scalable deployments.

VII. Important Links and References

- OpenWeatherMap API Documentation
- Docker Compose Documentation
- Jenkins Installation
- Kubernetes Setup
- Github Link of the project