

# CENG 495 – Cloud Computing Assignment 2 Report

Tufan Özkan  
2580850

May 11, 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Setup &amp; Tools</b>	<b>2</b>
2.1	Tools . . . . .	2
2.2	Microservice Application Choice . . . . .	2
2.3	Initial Setup . . . . .	3
<b>3</b>	<b>Kubernetes Cluster Configuration</b>	<b>4</b>
<b>4</b>	<b>Skaffold Deployment</b>	<b>5</b>
<b>5</b>	<b>Frontend Exposure</b>	<b>8</b>
<b>6</b>	<b>Chatbot Integration</b>	<b>9</b>
6.1	Deploying Ollama in Kubernetes . . . . .	9
6.2	Exposing the Ollama Service . . . . .	10
6.3	Pulling the Model . . . . .	10
6.4	Minikube Restart Due to Memory Error . . . . .	10
6.5	Integration in <code>app.js</code> . . . . .	11
6.6	Frontend Chatbot UI . . . . .	11
6.7	Kubernetes Final Details . . . . .	11
6.8	Final Output in Browser . . . . .	12
<b>7</b>	<b>Conclusion</b>	<b>13</b>

# 1 Introduction

In this project, I deployed the **AcmeAir Node.js** application using **Minikube** and **Scaffold**. I also integrated a chatbot into the system using the **Ollama smollm2** model. During the process, I faced various issues like image pull errors, proxy setup problems, and model loading failures, but managed to solve them one by one. In the end, the app runs successfully on Kubernetes with a working chatbot that responds through a popup on the main page.

## 2 Setup & Tools

### 2.1 Tools

- Minikube v1.35.0
- Scaffold v2.16.0
- Kubectl: v1.27.2 (Client)
- Docker Desktop v4.22.0
- Nodejs: v22.12.0
- npm: v11.0.0
- Ollama: latest (with smollm2 model)

### 2.2 Microservice Application Choice

I chose the **AcmeAir Node.js** project because it seemed more manageable compared to others. In the microservices GitHub page, the projects were listed based on how many services they had, and AcmeAir had fewer components, which made it easier to handle during deployment.

Also, since I had already worked with a **Node.js** frontend in the first assignment, I thought integrating additional features like the chatbot would be easier for me.

Another reason was that the **AcmeAir README** was very clear and well-documented, which helped a lot during setup. The instructions were straightforward, and I was able to get the app running quickly compared to more complex alternatives.

## 2.3 Initial Setup

### a) Minikube Installation:

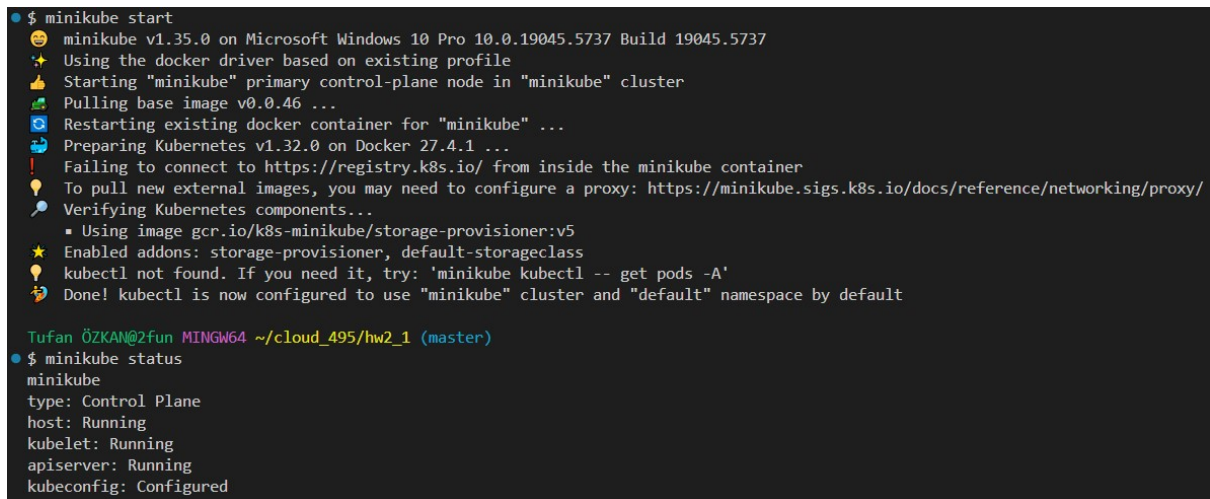
- Rather than modifying BIOS settings or installing Hyper-V, I decided to use Docker as the Minikube driver. Since Docker Desktop was installed instead of the Hyper-V PowerShell Module in my Windows 10, I created minikube with the code below:

```
minikube start --driver=docker
```

- After the successful creation,

```
minikube status
```

returned as below:



```
$ minikube start
minikube v1.35.0 on Microsoft Windows 10 Pro 10.0.19045.5737 Build 19045.5737
Using the docker driver based on existing profile
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.46 ...
Restarting existing docker container for "minikube" ...
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
Failing to connect to https://registry.k8s.io/ from inside the minikube container
To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
Verifying Kubernetes components...
  Using image gcr.io/k8s-minikube/storage-provisioner:v5
Enabled addons: storage-provisioner, default-storageclass
kubectrl not found. If you need it, try: 'minikube kubectrl -- get pods -A'
Done! kubectrl is now configured to use "minikube" cluster and "default" namespace by default

Tufan ÖZKAN@2fun MINGW64 ~/cloud_495/hw2_1 (master)
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Figure 1: minikube start & status

### b) Skaffold:

- Installed Skaffold CLI from [skaffold.dev](https://skaffold.dev).
- Verified installation:

```
skaffold version
```

### c) Installing Application

- After cloning the AcmeAir Node.js repository, I followed the instructions in the README to get the project running locally. I installed the required Node.js dependencies using:

```
npm install
```

- To run the application in monolithic mode, I used the following command:

```
node app.js
```

- This helped me verify that the app worked in local environment before deploying it with Skaffold and Minikube.

### 3 Kubernetes Cluster Configuration

I created a **YAML** file named `acmeair-deployment.yaml` that describes how the application is deployed on the cluster. I used a **Deployment** resource because it automatically ensures keeping the pod running — for example, if it crashes, **Kubernetes** restarts it.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: acmeair
spec:
  selector:
    matchLabels:
      app: acmeair
  replicas: 1
  template:
    metadata:
      labels:
        app: acmeair
    spec:
      containers:
        - name: acmeair
          image: acmeair-nodejs
          imagePullPolicy: Never
```

- **apiVersion:** Defines which version of the Kubernetes API is used. Here I used `apps/v1`, which is the stable version for Deployment-type resources.
- **kind:** This field tells Kubernetes what type of object is being defined. I used `Deployment` because it's best for long-running applications that may need updates or restarts. Other possible kinds include: `Pod`, `Job`, and `Service`.
- **metadata.name:** The name of the deployment, which I used as `acmeair` so it's easier to manage with `kubectl get deployments`.
- **selector.matchLabels:** This connects the deployment with the pods it manages. It looks for pods labeled `app: acmeair`.

- **replicas:** I set this to *1* because I only needed one instance of the app for this assignment.
- **template.metadata.labels.app:** This part sets the label that the selector above will match. If these do not match, the deployment will not manage the pod.
- **name:** The name of the container — used mainly for logs and debugging.
- **image:** This is the Docker image that contains the Node.js version of AcmeAir. Since I used Skaffold to build it locally inside Minikube, I didn't push it to Docker Hub.
- **imagePullPolicy:** — This is really important. At first, I kept getting `ErrImagePull` errors because Kubernetes was trying to download the image from Docker Hub. Setting this to `Never` tells Kubernetes to just use the image that's already built locally.
- **containerPort:** — This is the port my app listens to. It must match what is defined in `app.js` inside the Node.js code.

Overall, this file helps Kubernetes know how to run the app and keep it alive. It took some trial and error, especially around the image pull policy and label matching.

## 4 Skaffold Deployment

Skaffold helped me automate the process of building Docker images and deploying them to Kubernetes using my local Minikube setup. Below is my configuration and explanation of what each part does, along with issues I ran into and how I resolved them.

```
apiVersion: skaffold/v4beta6
kind: Config
metadata:
  name: acmeair-nodejs
build:
  artifacts:
    - image: acmeair-nodejs
      context: .
      docker:
        dockerfile: Dockerfile
  local:
    push: false
manifests:
  rawYaml:
    - k8s/*.yaml
```

- **apiVersion:** This defines the Skaffold config version. I originally used an older version, but Skaffold threw an error saying it was deprecated, so I updated it to `v4beta6`, which was compatible with the latest installation.
- **kind:** Standard for Skaffold configuration files. It just declares that this YAML is a Skaffold setup, set to `Config`.

- `metadata.name`: This is optional but helps label the Skaffold project. I named it after my app for clarity.
- `build.artifacts`
  - `image`: This section tells Skaffold how to build my Docker image. I had to make sure the image name matched what I used in `acmeair-deployment.yaml`.
  - `context`: It means Skaffold should look at the current directory when building.
  - `dockerfile`: It ensures it uses my custom Dockerfile (modified the current one in the application).
- `local.push`:
  - I ran into `ErrImagePull` issues at first. This field tells Skaffold **not to push the image to a remote registry**, and instead build it locally so Minikube can use it directly.
  - This worked because I ran `eval $(minikube docker-env)` beforehand, so the local Docker daemon was shared.
- `manifests.rawYaml`: I placed all my Kubernetes YAML files inside the `k8s/` folder. This line tells Skaffold to apply every file in there during deployment.

In the beginning, I faced several deployment errors while running `skaffold dev`. One major issue was the `ErrImagePull` error, because Kubernetes tried to pull my Docker image from Docker Hub instead of using the local build. I resolved this by enabling Minikube's Docker daemon using `eval $(minikube docker-env)` and setting `imagePullPolicy: Never` in my Deployment YAML.

Additionally, the original `Dockerfile` had some issues — the working directory was misaligned, and the entrypoint script `run.sh` wasn't being executed correctly due to permission and deprecated command problems. I fixed these by adjusting the `WORKDIR`, installing **NodeJS** explicitly, and updating environment variable declarations to modern standards. These changes were crucial to get the app up and running successfully.

Overall, this configuration helped me rapidly develop and test my project. Skaffold automatically rebuilt the image and updated the cluster every time I changed a file, which was very useful for debugging. Here's the result of running:

```

PS C:\Users\Tufan ÖZKAN\cloud_495\hw2_1\acmeair-nodejs> skaffold dev
Generating tags...
- acmeair-nodejs -> acmeair-nodejs:11fa31a-dirty
Checking cache...
- acmeair-nodejs: Found Locally
Tags used in deployment:
- acmeair-nodejs -> acmeair-nodejs:3497bd45596ed71803a9a614d01958aef5c61a94809cdf07f4dd6977401d7017
Starting deploy...
- deployment.apps/acmeair configured
- service/acmeair-service configured
- deployment.apps/mongo configured
- service/mongo configured
- deployment.apps/ollama configured
- service/ollama-service configured
Waiting for deployments to stabilize...
- deployment/acmeair is ready. [2/3 deployment(s) still pending]
- deployment/mongo is ready. [1/3 deployment(s) still pending]
- deployment/ollama is ready.
Deployments stabilized in 3.111 seconds
Listing files to watch...
- acmeair-nodejs
Press Ctrl+C to exit
Watching for changes...

```

Figure 2: skaffold dev

```

C:\Users\Tufan ÖZKAN\cloud_495\hw2\acmeair-nodejs>kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
acmeair-deployment  1/1     1             1           2m25s

C:\Users\Tufan ÖZKAN\cloud_495\hw2\acmeair-nodejs>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
acmeair-deployment-ddb877cbd-qq556  1/1     Running   0           2m31s

C:\Users\Tufan ÖZKAN\cloud_495\hw2\acmeair-nodejs>kubectl get services
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
acmeair-service     NodePort    10.101.231.153  <none>        9080:30080/TCP   2m34s
kubernetes           ClusterIP   10.96.0.1       <none>        443/TCP          12d

```

Figure 3: Kubernetes Cluster Details - kubectl get deployments/pods/services

## 5 Frontend Exposure

To expose the AcmeAir frontend so I could access it from my browser, I created a Kubernetes **Service** of type **NodePort**. This configuration allowed traffic to reach the application running inside the Minikube cluster using the node's IP and a manually assigned external port.

```
apiVersion: v1
kind: Service
metadata:
  name: acmeair-service
spec:
  type: NodePort
  selector:
    app: acmeair
  ports:
    - port: 9080
      targetPort: 9080
      nodePort: 30080
```

- **apiVersion: v1** is the standard API version for defining services in Kubernetes.
- **kind:** Specifies the type of resource. In this case, it's a **Service**, which allows internal or external access to a set of Pods.
- **metadata.name:** The name of the service, *acmeair-service* — I used this name when referencing the service via `minikube service acmeair-service --url`.
- **spec.type:** This exposes the service on a port on the node (e.g., Minikube), so I could access it via `localhost:30080`. I chose **NodePort** because it's simple and works well for local development.
- **selector.app:** This tells Kubernetes which Pods the service should route traffic to. It selects the pod(s) with the label `acmeair`, which matches what I used in my Deployment.
- **ports.port:** and **targetPort:** Both are set to 9080, which is the internal port that the NodeJS application listens on inside the container.
- **nodePort:** This is the actual port exposed on the Minikube VM (the node). I manually set it to 30080 to avoid dynamic assignment, so I always knew which port to open in my browser.

The app does **not** work at `localhost:30080` directly unless using a tunnel (port forwarding). That's why I always checked the full URL using:

```
minikube service acmeair-service --url
```

Overall, the result is:



```
PS C:\Users\Tufan ÖZKAN\cloud_495\hw2_1\acmeair-nodejs> minikube service acmeair-service
-----|-----|-----|-----|
| NAMESPACE | NAME       | TARGET PORT | URL                |
|-----|-----|-----|-----|
| default    | acmeair-service | 9080        | http://192.168.49.2:30080 |
|-----|-----|-----|-----|
🔗 Starting tunnel for service acmeair-service.
-----|-----|-----|-----|
| NAMESPACE | NAME       | TARGET PORT | URL                |
|-----|-----|-----|-----|
| default    | acmeair-service |             | http://127.0.0.1:52352 |
|-----|-----|-----|-----|
🔗 Opening service default/acmeair-service in default browser...
! Because you are using a Docker driver on windows, the terminal needs to be open to run it.
```

Figure 4: Service Details

## 6 Chatbot Integration

To enhance the AcmeAir project, I integrated a chatbot using the **Ollama** runtime and a lightweight model named **smollm2**. This chatbot runs in a separate container and is accessed from the frontend via a REST API proxy. Here's how I implemented it.

### 6.1 Deploying Ollama in Kubernetes

I started by deploying **Ollama** as a container. This YAML named `ollama-deployment.yaml` creates a single-pod Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ollama
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ollama
  template:
    metadata:
      labels:
        app: ollama
    spec:
      containers:
        - name: ollama
          image: ollama/ollama
          ports:
            - containerPort: 11434
          securityContext:
            privileged: true
```

To make sure the Ollama container could actually work inside the pod (especially for running AI models), I had to add `privileged: true` under the container's security settings.

- `privileged: true` basically tells Kubernetes to let the container act like it has full access to the system.
- I noticed that Ollama needs to interact more closely with the machine's CPU or hardware, and it wouldn't start properly without this.
- Before adding this, the pod was stuck in `ContainerCreating` or it failed silently, so adding this fixed the issue and let it pull the model and run smoothly.

This part of the deployment also exposes port 11434, which is used by the **Ollama API** to send and receive chat messages from the frontend.

## 6.2 Exposing the Ollama Service

To allow **AcmeAir** to reach **Ollama**, I exposed the service using `NodePort`:

```
apiVersion: v1
kind: Service
metadata:
  name: ollama-service
spec:
  type: NodePort
  selector:
    app: ollama
  ports:
    - port: 11434
      targetPort: 11434
      nodePort: 31134
```

## 6.3 Pulling the Model

Once the Ollama container was running, I manually pulled the model inside the pod using:

```
kubectl exec -it <ollama-pod-name> -- bash
ollama pull smollm2
```

This downloaded the lightweight `smollm2` model into the running container, enabling it for local inference.

## 6.4 Minikube Restart Due to Memory Error

At first, the Ollama pod failed to start due to memory limitations in the default Minikube configuration. To fix this, I deleted and restarted Minikube with more resources:

```
minikube delete
minikube start --memory=8192 --cpus=4
```

After that, Ollama successfully started and the chatbot became functional.

## 6.5 Integration in app.js

To make the chatbot work with the existing **AcmeAir frontend**, I had to make sure the frontend could talk to the **Ollama** backend. Instead of calling **Ollama** directly, I added a small reverse proxy inside `app.js` using a **Node.js** library called `http-proxy-middleware`.

```
const { createProxyMiddleware } = require('http-proxy-middleware');

app.use('/api/chat', createProxyMiddleware({
  target: 'http://ollama-service:11434',
  changeOrigin: true,
  pathRewrite: {
    '^/api/chat': '/api/chat'
  }
}));
```

This means whenever the frontend sends a request to `/api/chat`, it gets forwarded to the **Ollama** service running in **Kubernetes**. This also helped me avoid issues, and I didn't have to change anything on the frontend side.

## 6.6 Frontend Chatbot UI

I updated the `index.html` file to add a simple chatbot popup using basic HTML, CSS, and JavaScript. I used a small airplane icon as the button to open the chat window. When opened, it shows a chatbox with profile pictures for both the user and the bot, and messages appear as chat bubbles.

The JavaScript part sends user messages to `/api/chat` and handles the streamed response from **Ollama**. It then updates the chatbox in real time, giving the feel of a real conversation — but built right into the old **AcmeAir** interface.

## 6.7 Kubernetes Final Details

To ensure that all resources were deployed correctly, I ran the following commands:

```
kubectl get pods
kubectl get services
kubectl get deployments
kubectl get endpoints
```

These confirmed that the application pod, Ollama model pod, and their services were running as expected.

```

PS C:\Users\Tufan ÖZKAN\cloud_495\hw2_1\acmeair-nodejs> minikube service acmeair-service

```

NAMESPACE	NAME	TARGET PORT	URL
default	acmeair-service	9080	http://192.168.49.2:30080

```

✈ Starting tunnel for service acmeair-service.

```

NAMESPACE	NAME	TARGET PORT	URL
default	acmeair-service		http://127.0.0.1:52352

```

🔔 Opening service default/acmeair-service in default browser...
! Because you are using a Docker driver on windows, the terminal needs to be open to run it.

```

Figure 5: Service Details

## 6.8 Final Output in Browser

This screenshot shows the complete working application at local browser, including the embedded chatbot powered by `smollm2` running through Ollama.

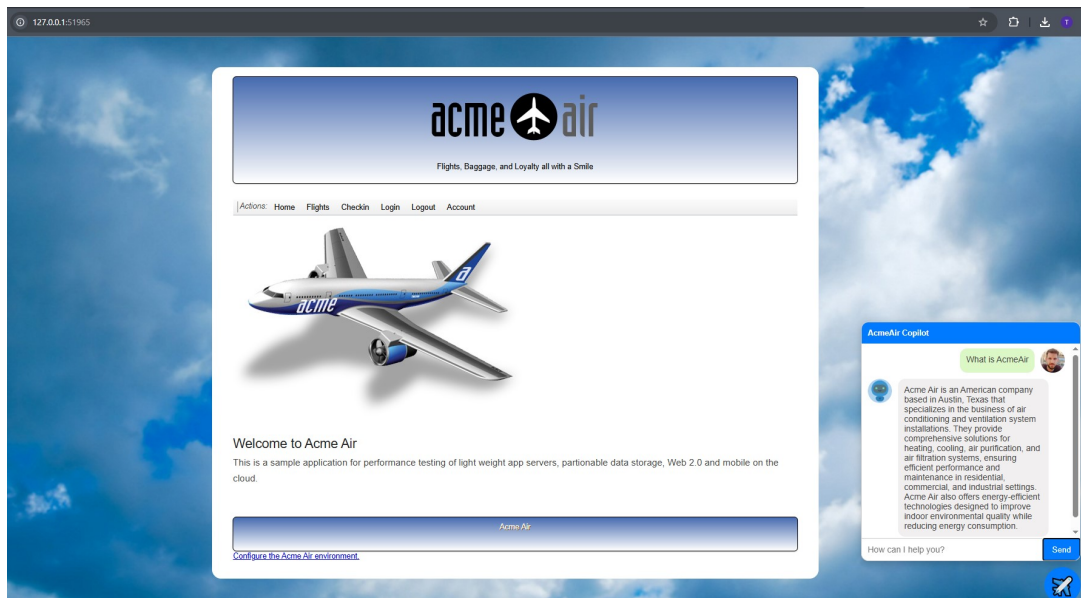


Figure 6: AcmeAir app running in browser with integrated chatbot popup

## 7 Conclusion

Overall, this project helped me understand how to deploy a full-stack cloud-native application using Kubernetes. I worked through several challenges like configuring deployments, solving image pull issues, exposing the frontend, and integrating a chatbot using Ollama. Even though I had to troubleshoot many problems — especially with resource limits and networking — I learned a lot about how things work behind the scenes. In the end, I was able to run the AcmeAir app locally with a working AI-powered chatbot.