

# Containerized Full-Stack Application on AWS

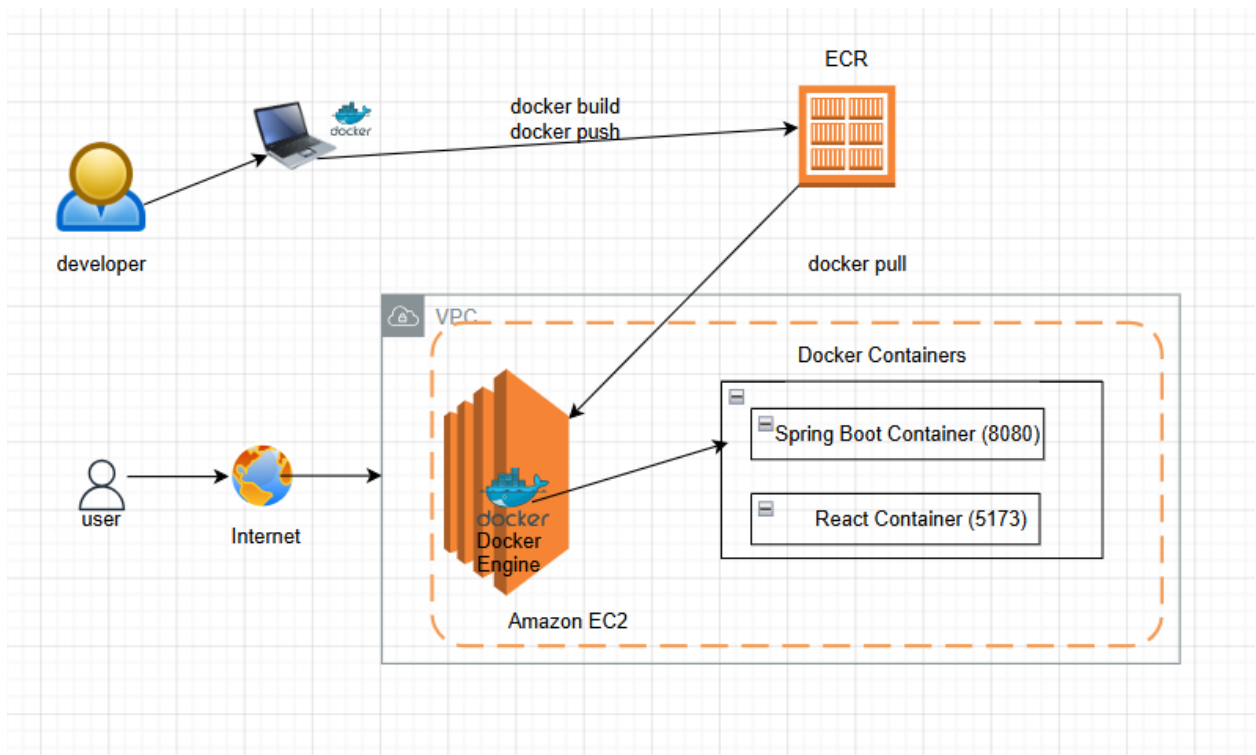
## Project Overview:

Designed and implemented a containerized full-stack web application that recommends AWS architecture solutions based on application requirements. The solution was developed using Spring Boot for the backend and React for the frontend, containerized using Docker, stored in Amazon ECR, and deployed on Amazon EC2. This project demonstrates end-to-end application development, containerization, and cloud deployment on AWS.

## 1. Technology Stack Table:

| Layer            | Technology        |
|------------------|-------------------|
| Frontend         | React + Vite      |
| Backend          | Spring Boot       |
| Containerization | Docker            |
| Orchestration    | Docker Compose    |
| Image Repository | Amazon ECR        |
| Compute          | Amazon EC2        |
| OS               | Amazon Linux 2023 |

## 2. Architectural Diagram



## 3. Spring Boot Backend Set-up:

### 3.1 Installation:

Install VSCode with the following extension packs:

- Extension Pack for Java
- Spring Boot Extension Pack
- Maven for Java
- Docker
- Language Support for Java
- Test Runner for Java

Install JDK 21 from: [Eclipse Adoptium Downloads](#)

### 3.2 Create Spring-Boot application using VS Code:

**3.2.1** Using Ctrl + Shift + P, create a new project

**3.2.2** Search Spring Initializr: Create a Maven Project:

### **3.2.3** Spring Boot 3.x

### **3.2.4** Java

Group Id: com.dipti

Artifact Id: architecture-engine

Packaging: Jar

Java Version: 21

### **3.2.5** Add dependency: Spring Web

### **3.2.6** Choose a folder to save the project

### **3.2.7** Open project – folder - architecture-engine

#### **3.2.7.1** Run the application using the following command in the VSCode terminal `.\mvnw.cmd spring-boot:run`

This file was generated automatically by Spring Initializr. It's called by Maven Wrapper, which is a mini bootstrap program that downloads and runs Maven without installing Maven. It reads POM and downloads all the dependencies needed by the application and converts java file to class files and store it under target/classes.

### **3.2.8** Once it starts, open: <http://localhost:8080> If you can open the default page, it means the installation and the project were successful.

## **3.3 Code**

### **3.3.1** Create

- Controller receives API requests
- Service contains business logic
- Model represents data objects

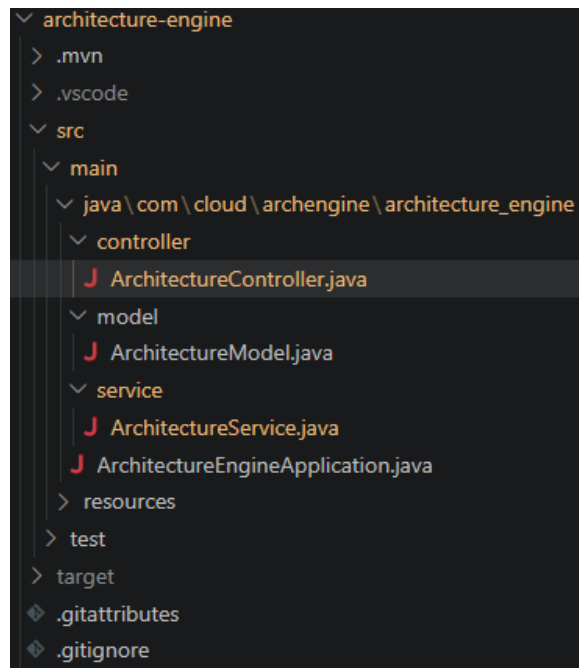


Figure 1. Springboot Project Structure

3.3.2 Code uploaded to my git repository at: <https://github.com/uppaldipti/aws-architecture-recommendation-engine/tree/main/architecture-engine>

```
import com.cloud.archengine.architecture_engine.service.ArchitectureService;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

@CrossOrigin
@RestController
public class ArchitectureController {
    @Autowired
    ArchitectureService archService;

    @Autowired
    private ObjectMapper objectMapper; // Spring Boot automatically injects this

    @GetMapping("/getarhitecture")
    public JsonNode getArchitecture(String appType){
        try {
            //call the method from service class
            JsonNode jsonNode =archService.getArchDetailsJson(appType);
            return jsonNode;
        } catch (Exception e) {
            System.out.println("Exception -> " + e.getMessage());
            e.printStackTrace();
        }
        return null;
    }
}
```

Figure 2: Spring Boot REST controller endpoint.

### 3.4 Local Testing:

**3.4.1** Start and run the server using `.\mvnw.cmd spring-boot:run`

**3.4.2** Call the API with the parameter in Postman.

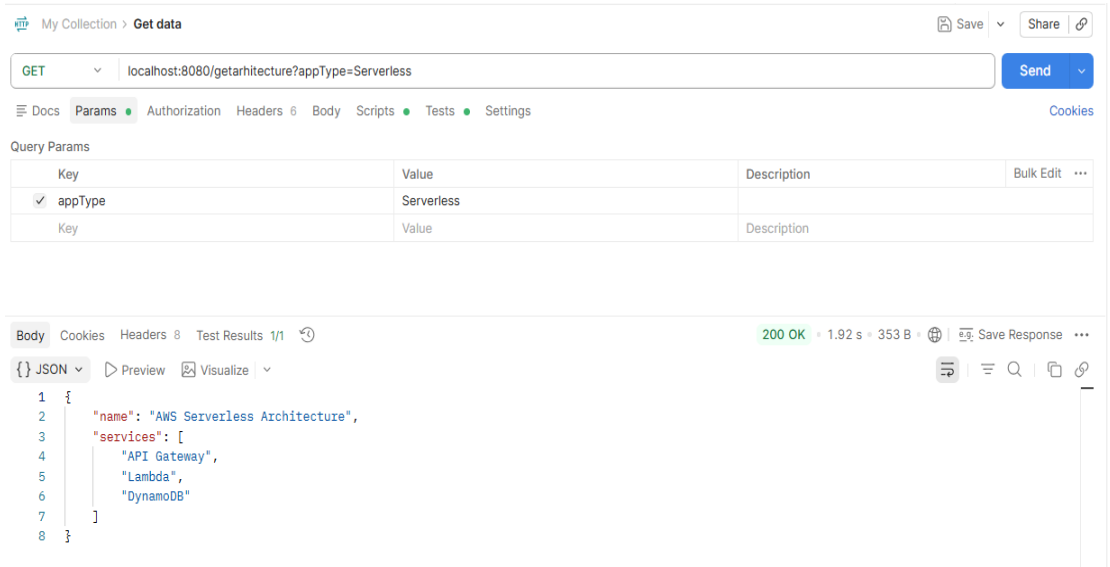


Figure 3: Testing endpoint.

## 4. React Frontend Set-up:

**4.1 Installation:** Install VSCode

**4.2 Create React application using VSCode:**

4.2.1 Open VS Terminal and go to the folder you want to create React app and run

```
npm create vite@latest architecture-ui
```

4.2.2 Select the Framework -> React, Variant -> Javascript

4.2.3 cd to the folder and run

```
npm install
```

```
npm run dev
```

4.2.4 You should see : <http://localhost:5173/> - It means the server is started

## 4.3 Code:

### 4.3.1 Project Structure in React will look like this

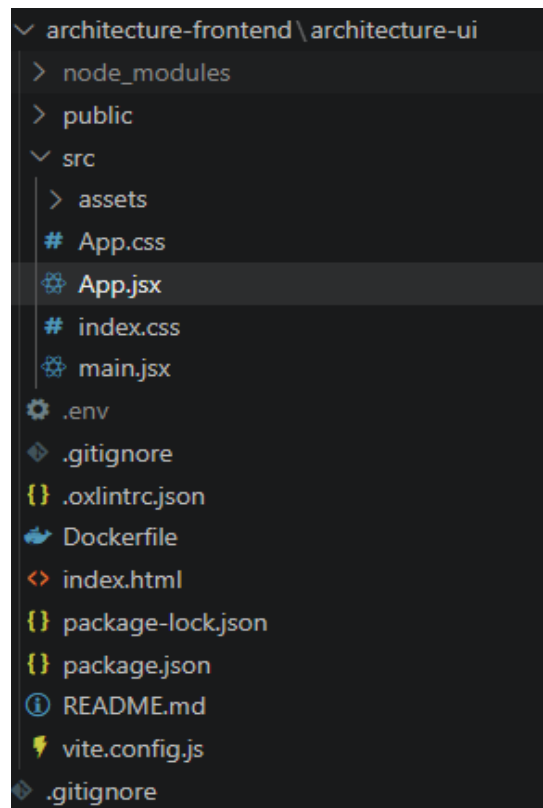


Figure 4: React Project Structure

4.3.2 Edit App.jsx file with the code that can be found in my repo at <https://github.com/uppaldipty/aws-architecture-recommendation-engine/tree/main/architecture-frontend/architecture-ui>

4.3.3 fetch() is used to invoke the Spring Boot REST API endpoint and retrieve the architecture recommendation as a JSON response.

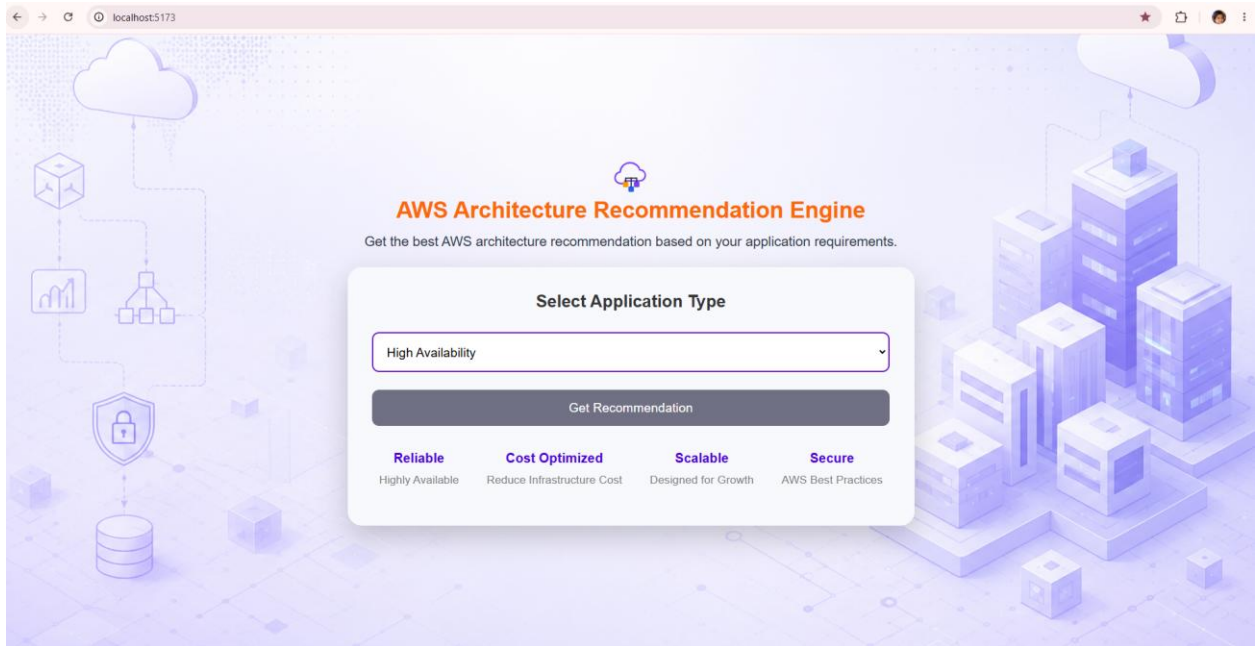
```
const getRecommendation = async () => {  
  const response = await fetch(  
    `${API_URL}/getarhitecture?appType=${appType}`  
  )  
}
```

Figure 5: React endpoint to consume the api request

## 4.4 Local Testing:

4.4.1 Run `npm run dev`

4.4.2 You should see:



*Figure 6: Testing application.*

## 5. End-to-End Architecture Flow:

**5.1 Architecture Flow Description:** The React frontend captures the user-selected application type, invokes the Spring Boot REST API, and dynamically renders the recommended AWS architecture based on the JSON response returned by the service layer.

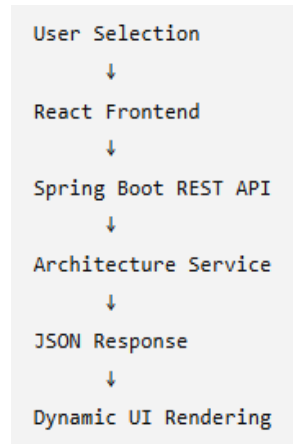


Figure 7: End-To-End Flow Diagram

**5.2 React-to-Spring Boot API Communication:** Figure 8 demonstrates the successful communication between the React frontend and the Spring Boot REST API. The browser Network tab confirms that the selected application type is passed to the backend endpoint and a JSON response is returned successfully.

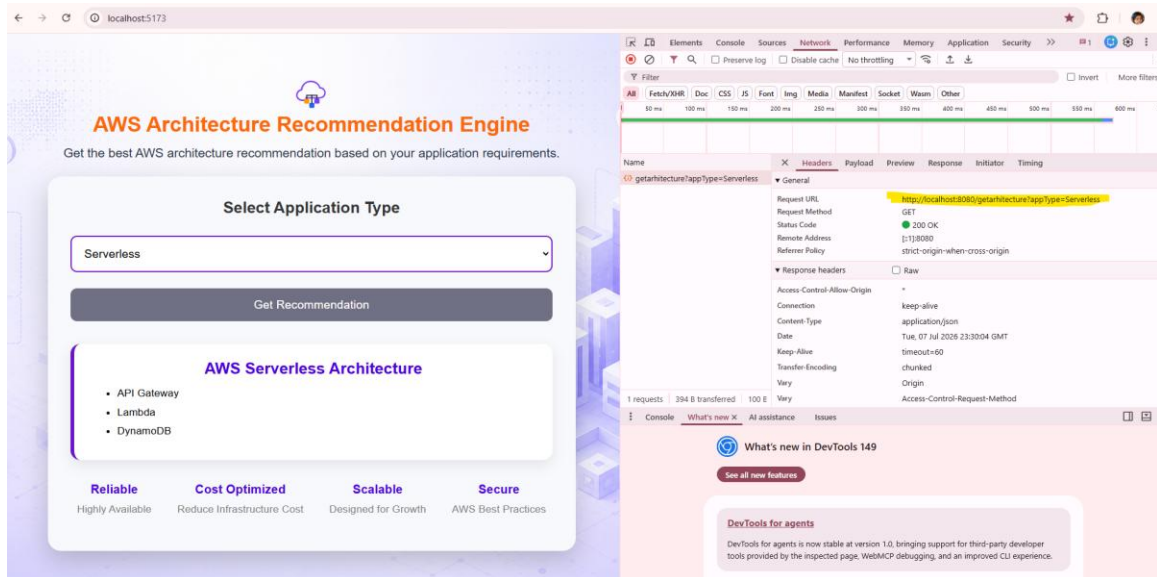


Figure 8: React Frontend Invoking Spring Boot REST API

## 6. Containerization using Docker:

**6.1 Why Docker?** Docker is used to package an application and its dependencies into a portable container that can run consistently across different environments.

**6.2 Dockerfile Overview:** This file is added to the project with the instructions that are required by Docker to build the Docker Image. This Image can be used to run 1 or more containers.

**6.3 Installation:** Use powershell or command prompt

Install Docker Desktop

Verify Docker Installation

Install WSL (Windows Subsystem for Linux)

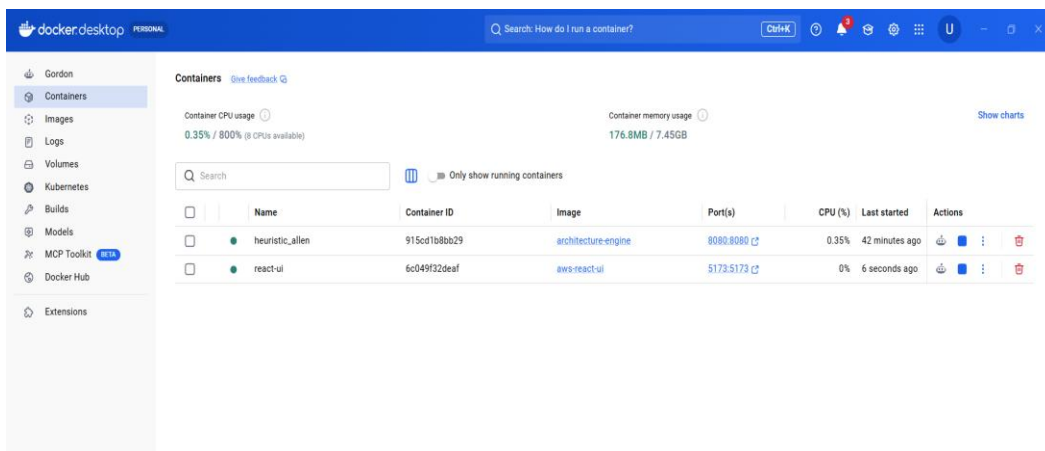
`docker --version`

`docker ps`

## 6.4 Flow / Code / Build / Run:

| Dockerize React                                                                                                                                                                         | Dockerize Spring Boot                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Dockerize React</li></ul> <p>React Code<br/>↓<br/>Dockerfile<br/>↓<br/>Docker Image<br/>↓<br/>Docker Container</p>                              | <ul style="list-style-type: none"><li>• Dockerize Spring Boot</li></ul> <p>Springboot Code<br/>↓<br/>Dockerfile<br/>↓<br/>Docker Image<br/>↓<br/>Docker Container</p> |
| <p>//Code in Dockerfile</p> <pre>FROM node:20-alpine WORKDIR /app COPY package*.json ./ RUN npm install COPY . . EXPOSE 5173 CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0"]</pre> | <p>//Code in Dockerfile</p> <pre>FROM eclipse-temurin:21-jdk WORKDIR /app COPY target/*.jar app.jar EXPOSE 8080 ENTRYPOINT ["java", "-jar", "app.jar"]</pre>          |
| <p>//Build Docker Image<br/>docker build -t aws-react-ui .</p> <p>//Run Docker Image<br/>docker run -d -p 3000:3000 --name react-ui aws-react-ui</p>                                    | <p>//Build Docker Image<br/>docker build -t architecture-engine-backend .</p> <p>//Run Docker Image<br/>docker run -p 8080:8080 architecture-engine-backend</p>       |

6.5 After updating both the applications and running the commands, we can now see that the applications are running in their own containers.



*Figure 9: React and Spring containers*

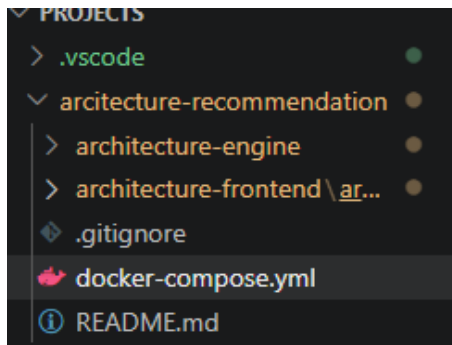
**6.6** In real-world applications, multiple services often need to run together. Managing each container separately can become difficult as the application grows. Docker Compose helps define and manage multiple containers as a single application stack using a single **docker-compose.yml** file, making deployment and maintenance much easier.

**6.7** What is docker-compose doing?

- Creates a private Docker network
- Connects containers together
- Handles startup order
- Starts everything with one command

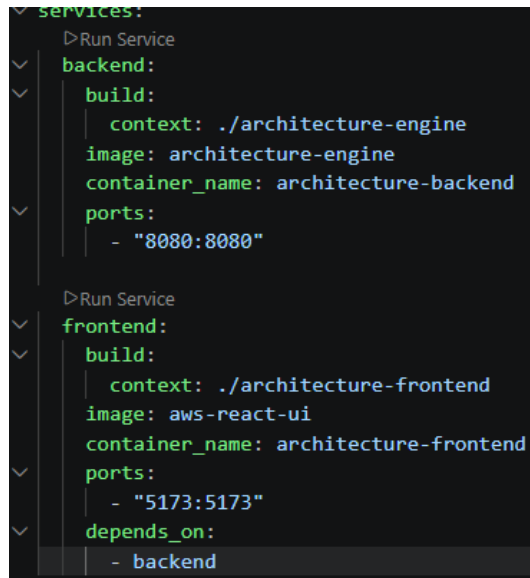
**6.8** docker-compose.yml: This file is created in the root folder of the project.

NOTE: Stop the existing containers.



## 6.9 Flow / Code / Build / Run:

//docker-compose.yml



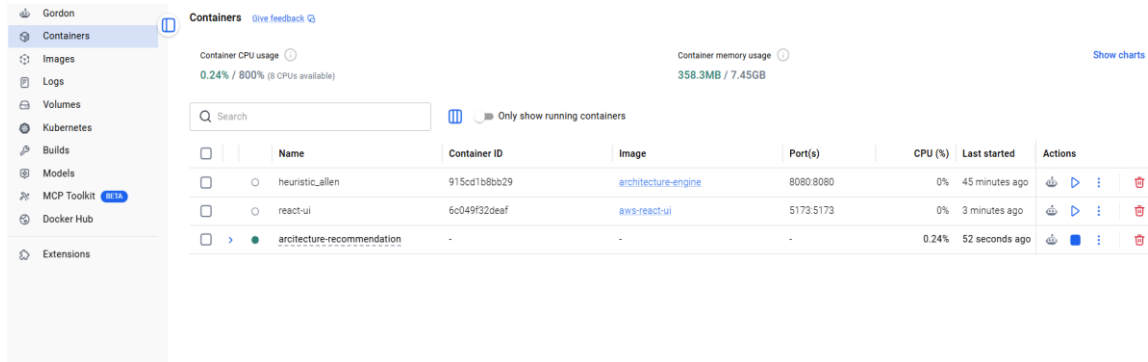
```
services:
  backend:
    build:
      context: ./architecture-engine
    image: architecture-engine
    container_name: architecture-backend
    ports:
      - "8080:8080"
  frontend:
    build:
      context: ./architecture-frontend
    image: aws-react-ui
    container_name: architecture-frontend
    ports:
      - "5173:5173"
    depends_on:
      - backend
```

//Build Docker Image  
docker compose build

//run Docker Image  
docker compose up -d  
(-d runs in background)

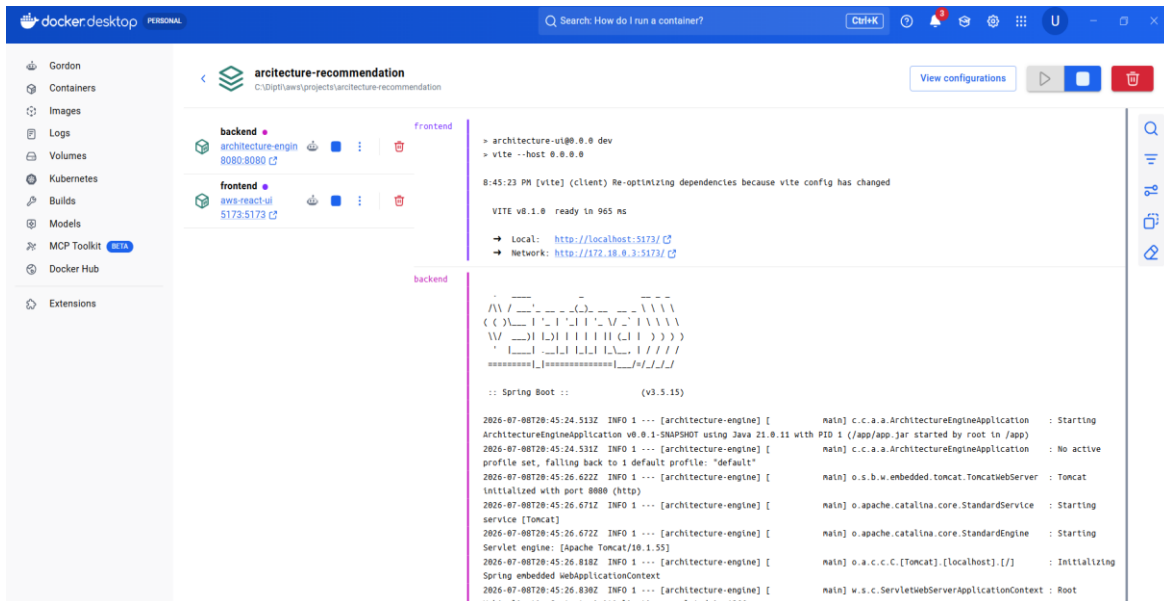
//Verify  
<http://localhost:5173>  
Application webpage will be shown as seen in Figure 8

**6.10** Figure 10 shows both frontend and backend containers running separately. Figure 11 shows Docker Compose managing both containers together under one application stack.



|                                     | Name                        | Container ID | Image               | Port(s)   | CPU (%) | Last started   | Actions |
|-------------------------------------|-----------------------------|--------------|---------------------|-----------|---------|----------------|---------|
| <input type="checkbox"/>            | heuristic_allen             | 915cd1b8bb29 | architecture-engine | 8080:8080 | 0%      | 45 minutes ago |         |
| <input type="checkbox"/>            | react-ui                    | 6c049f32aeaf | aws-react-ui        | 5173:5173 | 0%      | 3 minutes ago  |         |
| <input checked="" type="checkbox"/> | architecture-recommendation | -            | -                   | -         | 0.24%   | 52 seconds ago |         |

*Figure 10: Single container running*



```

> architecture-ui@0.0.0 dev
> vite --host 0.0.0.0

8:45:23 PM [vite] (client) Re-optimizing dependencies because vite config has changed
VITE v8.1.0 ready in 965 ms
→ Local: http://localhost:5173/
→ Network: http://172.18.0.3:5173/

:: Spring Boot :: (v3.5.15)

2026-07-08T20:45:24.513Z INFO 1 --- [architecture-engine] [main] c.c.a.a.ArchitectureEngineApplication : Starting
ArchitectureEngineApplication v0.0.1-SNAPSHOT using Java 21.0.11 with PID 1 (/app/app.jar started by root in /app)
2026-07-08T20:45:24.531Z INFO 1 --- [architecture-engine] [main] c.c.a.a.ArchitectureEngineApplication : No active
profile set, falling back to 1 default profile: "default"
2026-07-08T20:45:26.622Z INFO 1 --- [architecture-engine] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat
initialized with port 8080 (http)
2026-07-08T20:45:26.671Z INFO 1 --- [architecture-engine] [main] o.apache.catalina.core.StandardService : Starting
service [Tomcat]
2026-07-08T20:45:26.672Z INFO 1 --- [architecture-engine] [main] o.apache.catalina.core.StandardEngine : Starting
Servlet engine: [Apache Tomcat/10.1.55]
2026-07-08T20:45:26.818Z INFO 1 --- [architecture-engine] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing
Spring embedded WebApplicationContext
2026-07-08T20:45:26.830Z INFO 1 --- [architecture-engine] [main] w.s.c.ServletWebServerApplicationContext : Root
WebApplicationContext: initialization completed in 1066 ms

```

*Figure 11: Docker Compose project running both frontend and backend containers as one application stack.*

Now that the application is working locally, the next step is to make it cloud-ready by pushing the Docker images to Amazon ECR, launching an EC2 instance, installing Docker on EC2, pulling the images from ECR, and running the application on AWS.

## 7. Amazon Elastic Container Registry (ECR):

**7.1** Login to your AWS account. ECR – Create Repository.

### Settings:

Repository Name: aws-architecture-recommendation-engine  
Visibility: Private  
Tag Immutability: Disabled (default)  
Encryption: AES-256 (default)  
Click **Create Repository**.

**7.2** Install AWS CLI & Configure the AWS credentials using command prompt or powershell.

**7.3** In the same window - **Login to ECR** from your local machine using:

```
aws ecr get-login-password --region us-east-1 docker login --username AWS  
--password-stdin <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com
```

**OR**

```
$ECR_PASSWORD = aws ecr get-login-password --region us-east-1  
docker login --username AWS --password $ECR_PASSWORD <ACCOUNT_ID>.dkr.ecr.us-  
east-1.amazonaws.com
```

**7.4 Tag Images:** Docker image tags uniquely identify image versions and destinations. Before pushing to ECR, we tag the local image with the ECR registry URL, repository name, and version tag so Docker knows where to store the image. Tags also help manage multiple versions such as development, testing, and production releases. **We are essentially giving Docker the full address of where the image should live in ECR.**

### React:

```
docker tag aws-react-ui:latest <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-  
architecture-recommendation-engine:frontend
```

### Spring Boot:

```
docker tag architecture-engine:latest <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-  
architecture-recommendation-engine:backend
```

## 7.5 Push Images: Pushing the image to the mentioned path

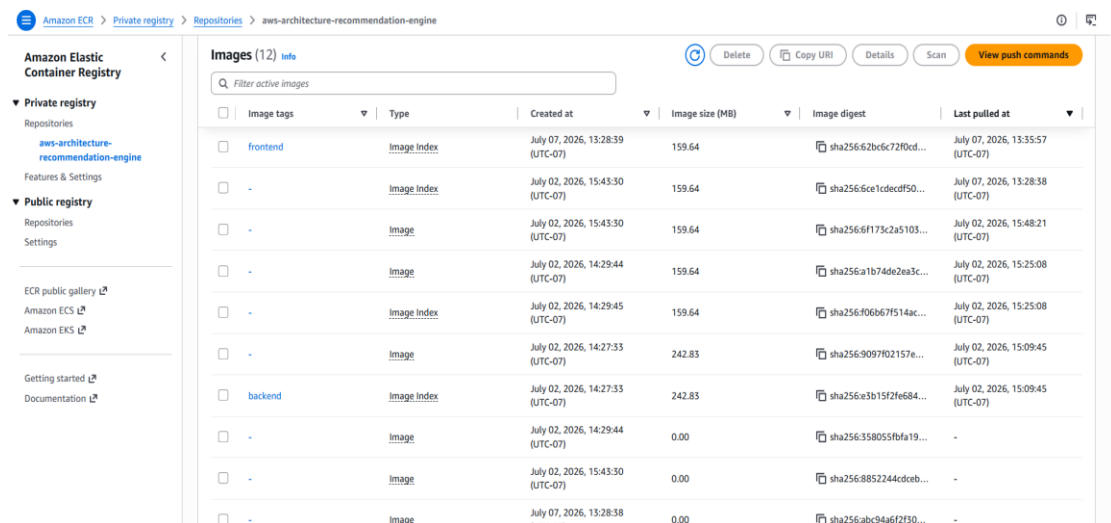
### React

```
docker push <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-recommendation-engine:frontend
```

### Springboot

```
docker push <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-recommendation-engine:backend
```

## 7.6 Verify the Images are stored in ECR Repository as shown.



| Image tags                        | Type        | Created at                       | Image size (MB) | Image digest           | Last pulled at                   |
|-----------------------------------|-------------|----------------------------------|-----------------|------------------------|----------------------------------|
| <input type="checkbox"/> frontend | Image Index | July 07, 2026, 13:28:39 (UTC-07) | 159.64          | sha256:62bdc672f0cd... | July 07, 2026, 13:35:57 (UTC-07) |
| <input type="checkbox"/> -        | Image Index | July 02, 2026, 15:43:30 (UTC-07) | 159.64          | sha256:6ce1cdecdf50... | July 07, 2026, 13:28:38 (UTC-07) |
| <input type="checkbox"/> -        | Image       | July 02, 2026, 15:43:30 (UTC-07) | 159.64          | sha256:6f173c2a5103... | July 02, 2026, 15:48:21 (UTC-07) |
| <input type="checkbox"/> -        | Image       | July 02, 2026, 14:29:44 (UTC-07) | 159.64          | sha256:a1b74de2ea3c... | July 02, 2026, 15:25:08 (UTC-07) |
| <input type="checkbox"/> -        | Image Index | July 02, 2026, 14:29:45 (UTC-07) | 159.64          | sha256:f06b67f514ac... | July 02, 2026, 15:25:08 (UTC-07) |
| <input type="checkbox"/> -        | Image       | July 02, 2026, 14:27:33 (UTC-07) | 242.83          | sha256:909702157e...   | July 02, 2026, 15:09:45 (UTC-07) |
| <input type="checkbox"/> backend  | Image Index | July 02, 2026, 14:27:33 (UTC-07) | 242.83          | sha256:3b15f2fe684...  | July 02, 2026, 15:09:45 (UTC-07) |
| <input type="checkbox"/> -        | Image       | July 02, 2026, 14:29:44 (UTC-07) | 0.00            | sha256:358055fbfa19... | -                                |
| <input type="checkbox"/> -        | Image       | July 02, 2026, 15:43:30 (UTC-07) | 0.00            | sha256:8852244dceb...  | -                                |
| <input type="checkbox"/> -        | Image       | July 07, 2026, 13:28:38 (UTC-07) | 0.00            | sha256:abc94a6f2f30... | -                                |

Figure 12: ECR Repository

## 8. Preparing EC2 for Deployment

**8.1 IAM role:** Will be created so the EC2 instance will have the authority to pull the image from the ECR Repository

### Permissions policies (2) [Info](#)

You can attach up to 10 managed policies.

| Search                   |                                                                                                                                      | Filter by Type |                   |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|----------------|-------------------|
|                          |                                                                                                                                      | All types      |                   |
| <input type="checkbox"/> | Policy name <a href="#">↗</a>                                                                                                        | Type           | Attached entities |
| <input type="checkbox"/> |  <a href="#">AmazonEC2ContainerRegistryReadOnly</a> | AWS managed    | 1                 |
| <input type="checkbox"/> |  <a href="#">CloudWatchAgentServerPolicy</a>        | AWS managed    | 1                 |

Figure 13: IAM Roles

### 8.2 Security group:

#### Inbound rules (4)

| Search                   |      |                        |            |            |          |            |           |  |  |
|--------------------------|------|------------------------|------------|------------|----------|------------|-----------|--|--|
| <input type="checkbox"/> | Name | Security group rule ID | IP version | Type       | Protocol | Port range | Source    |  |  |
| <input type="checkbox"/> | -    | sgr-025e47a524c2d96fc  | IPv4       | SSH        | TCP      | 22         | 0.0.0.0/0 |  |  |
| <input type="checkbox"/> | -    | sgr-03b7cb03eeb26f283  | IPv4       | HTTP       | TCP      | 80         | 0.0.0.0/0 |  |  |
| <input type="checkbox"/> | -    | sgr-0e8c66f5e3fb27b61  | IPv4       | Custom TCP | TCP      | 8080       | 0.0.0.0/0 |  |  |
| <input type="checkbox"/> | -    | sgr-0321caec4d0a057af  | IPv4       | Custom TCP | TCP      | 5173       | 0.0.0.0/0 |  |  |

#### Outbound rules (1)

| Search                   |      |                        |            |             |          |            |             |  |
|--------------------------|------|------------------------|------------|-------------|----------|------------|-------------|--|
| <input type="checkbox"/> | Name | Security group rule ID | IP version | Type        | Protocol | Port range | Destination |  |
| <input type="checkbox"/> | -    | sgr-006c7e338acef1811  | IPv4       | All traffic | All      | All        | 0.0.0.0/0   |  |

Figure 14: Security Rules

**8.3 Launch EC2 instance:** Launch an EC2 Instance with the IAM Role, Security group and accept all the default settings. Connect to EC2 by “connect using public IP”. Login and install Docker.

- `sudo dnf update -y`
- `sudo dnf install docker -y`
- `sudo systemctl start docker`
- `sudo systemctl enable docker`
- `sudo usermod -aG docker ec2-user`

Instances (1/1) [Info](#) Last updated 42 minutes ago [Connect](#) [Instance state](#) [Actions](#) [Launch instances](#)

Saved filter sets [Choose filter set](#)

| <input checked="" type="checkbox"/> | Name                           | Instance ID         | Instance state | Instance type | Status check      | Availability Zone | Public IPv4 DNS          | Public IPv4 ... |
|-------------------------------------|--------------------------------|---------------------|----------------|---------------|-------------------|-------------------|--------------------------|-----------------|
| <input checked="" type="checkbox"/> | aws-architecture-engine-docker | i-0dc659545484bd9ba | Running        | t3.micro      | 3/3 checks passed | us-east-1a        | ec2-44-199-252-104.co... | 44.199.252.104  |

---

**i-0dc659545484bd9ba (aws-architecture-engine-docker)**

[Details](#) [Status and alarms](#) [Monitoring](#) [Security](#) [Networking](#) [Storage](#) [Tags](#)

▼ Instance summary [Info](#)

|                                                                               |                                                                                           |                                                                                                                                         |
|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>Instance ID</b><br><a href="#">i-0dc659545484bd9ba</a>                     | <b>Public IPv4 address</b><br><a href="#">44.199.252.104</a> <a href="#">open address</a> | <b>Private IPv4 addresses</b><br><a href="#">172.31.10.160</a>                                                                          |
| <b>IPv6 address</b><br>-                                                      | <b>Instance state</b><br><a href="#">Running</a>                                          | <b>Public DNS</b><br><a href="#">ec2-44-199-252-104.compute-1.amazonaws.com</a> <a href="#">open address</a>                            |
| <b>Hostname type</b><br>IP name: ip-172-31-10-160.ec2.internal                | <b>Private IP DNS name (IPv4 only)</b><br><a href="#">ip-172-31-10-160.ec2.internal</a>   | <b>Elastic IP addresses</b><br>-                                                                                                        |
| <b>Answer private resource DNS name</b><br>IPv4 (A)                           | <b>Instance type</b><br>t3.micro                                                          | <b>AWS Compute Optimizer finding</b><br><a href="#">Opt-in to AWS Compute Optimizer for recommendations.</a> <a href="#">Learn more</a> |
| <b>Auto-assigned IP address</b><br><a href="#">44.199.252.104</a> [Public IP] | <b>VPC ID</b><br><a href="#">vpc-01fd94a28dd9f358f</a> (default VPC)                      |                                                                                                                                         |

Figure 15: Launched EC2.

8.4 Once we get the Public IP for the EC2 instance, update the address in:  
react project-> .env file->

NOTE: We have to build, tag & push this image again to ECR again

```

.env
architecture-recommendation > architecture-frontend > architecture-ui > .env
1 VITE_API_URL=http://44.199.252.104:8080

```

Figure 16: Updating IP for backend Service.

## 9. Pulling Images from ECR to EC2

9.1 Connect to EC2.

9.2 Login to ECR from EC2 using

```
aws ecr get-login-password --region us-east-1 docker login --username AWS  
--password-stdin <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com
```

You should see: Login Succeeded

9.3 Pull and Run the **SPRINGBOOT** Image using

```
docker pull <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-  
recommendation-engine:backend
```

```
docker run -d -p 8080:8080 --name architecture-engine-backend  
062541600148.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-  
recommendation-engine:backend
```

9.4 Pull and Run the **REACT** Image using

```
docker pull <ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-  
recommendation-engine:frontend
```

```
docker run -d -p 5173:5173 --name achitecture-engine-frontend  
062541600148.dkr.ecr.us-east-1.amazonaws.com/aws-architecture-  
recommendation-engine:frontend
```

9. 5 Check container using:

```
docker ps -a  
  
docker images  
docker logs architecture-engine-backend
```

## 10. Test the application in Browser

Frontend - <http://<EC2-PUBLIC-IP>:5173>

Backend- <http://<EC2-PUBLIC-IP>:/getarhitecture?appType=Serverless>

## 11.End-to-End Application running on Cloud

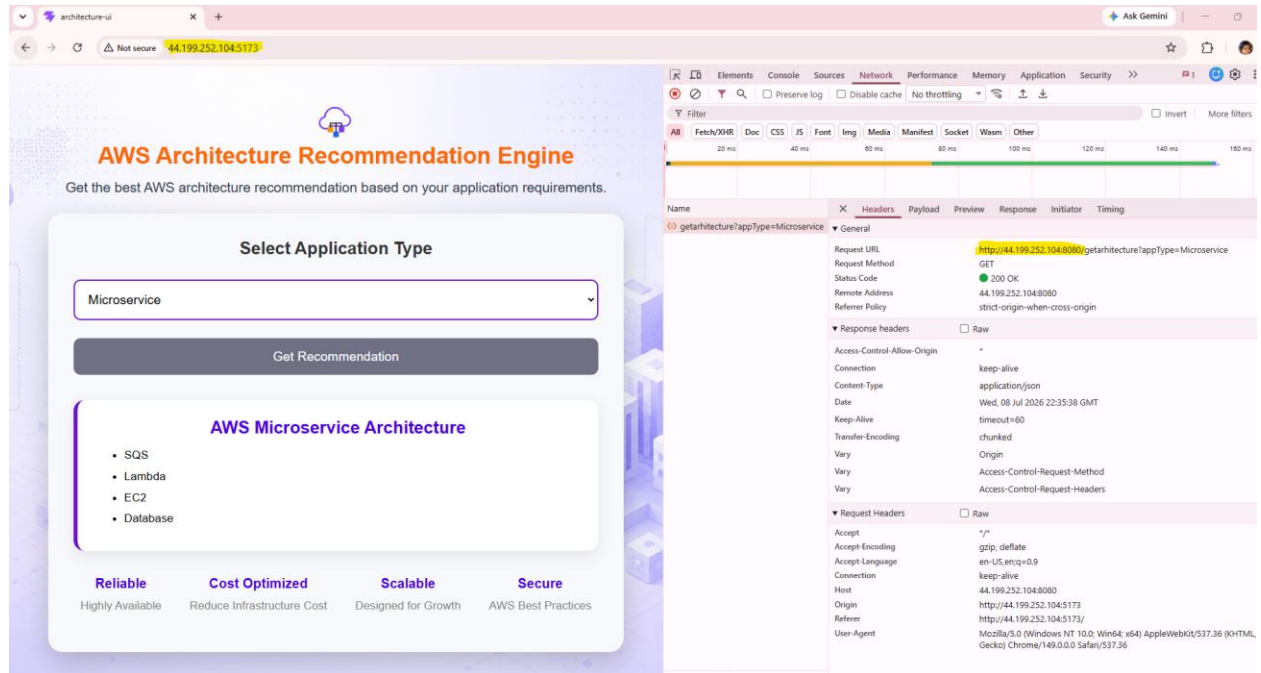


Figure 17: Containerize Cloud native Application

## 12. Challenges & Resolutions:

| Challenge                       | Resolution                      |
|---------------------------------|---------------------------------|
| Docker Desktop WSL issue        | Enabled WSL Integration         |
| React container not reachable   | Updated Vite configuration      |
| EC2 security group issue        | Opened port 5173                |
| Public IP changed after restart | Updated frontend API URL        |
| Docker login to ECR failed      | Configured IAM user and AWS CLI |

### **13. Conclusion:**

#### **Docker → ECR → EC2 → Container Running**

This project successfully demonstrated the complete lifecycle of a cloud-native application, from development and local testing to containerization, image management, and deployment on AWS. The application was built using React and Spring Boot, containerized with Docker, stored in Amazon ECR, and deployed on Amazon EC2, providing hands-on experience with modern cloud deployment practices.