COMP4651 Final Project Report

Topic: Serverless Role-Playing Chatbot Web Application With OpenAI API
Group members: Miu Victor (20867349), Mohammad Sufian (20773940)

Introduction

Background and Motivation

In today's fast-changing technological world, using the cloud for deployment has become a key part of how we develop and deliver applications. More and more businesses and developers are using the cloud to create, launch, and grow applications more efficiently while managing less infrastructure. Docker containers have become a really useful and adaptable option for deploying applications in the cloud, especially with all the different tools and technologies out there.

Docker containers let developers bundle applications with their dependencies into lightweight, portable units that can run reliably in any environment, such as: local machine, private data center, or on a public cloud. This solves the usual "it works on my machine" issue and makes sure that deployment is smooth across various stages of development and production. Docker containers also make scaling up and down the application much easier, which is why they are often chosen for cloud-native applications.

In order to gain more insight and practical experience on these useful techniques, in this project, we aim to use cloud providers like AWS and Google Cloud to deploy our simple role-playing chatbot with Azure OpenAI API for text generation by using docker containers.

Objective

Our objective is to successfully deploy a cloud-native Next.js application by using docker image. Our application should:

1. Correctly build an docker image for deployment
2. Be reliable and fault-tolerant
3. Be optimised for performance and have high availability with quick response times
4. Effectively utilise microservices architecture
5. Be readily available from anywhere with a web browser and internet connection

Implementation

Frontend

We used Next.js and TailwindCSS as a framework to build up the whole web application. Next.js is a powerful React-based web development framework that simplifies building modern, high-performance web applications. It supports features like server-side rendering

(SSR), API routes, and automatic code splitting. TailwindCSS is a utility-first CSS framework that enables developers to design responsive and beautiful interfaces quickly. By combining these two technologies, we can build modern small web projects with ease. On top of TailwindCSS, we implement the shadcn/ui library. It is an UI library that can easily integrate into a TailwindCSS project.

Backend

The backend of the web application is integrated into Next.js app file by using the API route. Using REST API style. We also use Clerk, a developer-focused authentication and user management system designed to simplify integrating user authentication, authorization, and user profiles into web and mobile applications.

Containerization

By referencing the official release of the docker file, we build up our own docker container that is suitable for use case. We have selected the newest version of Nodejs:alpine as our base container.

Setup steps:

1. Define working directory (usually /app)
2. Copy and install necessary dependencies
3. Copy content files and env for correctly building up and configure the web application
4. Define group and user for container safety management, limiting what the application can do even in the case of a breach.
5. Set expose port and hostname to allow accessing the web application

Cloud Deployment

We have tried 3 different deployment methods to deploy the docker image. Which are Google Cloud Run, AWS EC2 instance, and AWS ECS using Fargate.

AWS EC2

After successfully writing the Next.js application, we decided to use cloud-native technology to deploy the application. Given the limitation of the services provided with the AWS learner account, I decided to launch the application on an EC2 instance, using PM2 as a process manager and Nginx as a reverse proxy. In this way, users can access our application by visiting the public IP of our EC2 instance through any browser as long as the EC2 instance is running.

PM2

PM2 is a daemon process manager for Node.js which helps manage and keep our application online 24/7, which also includes tools for monitoring and automatic restarts. Using PM2 on our EC2 instance gives us tools to reliably keep our application running.

Nginx

Nginx is a web server where we launch our applications which also serves as a reverse proxy. Nginx was configured to route external requests to the Next.js application running on the EC2 instance. This setup allows Nginx to serve static assets directly, improving performance and reducing load on the application server. It also manages URL routing.

Deployment Steps

1. Launch an EC2 instance allowing inbound traffic, similar to the usage in lab sessions
2. Git clone the Next.js application onto the EC2 instance and install any dependencies
3. Install PM2 and Nginx
4. Start the Next.js application using PM2
5. Configure Nginx to set up a server block that listens on port 80 (HTTP) and forwards requests to our Next.js application running on port 3000 (or any other custom port)

The Next.js application should now be available through the public IP address of the host EC2 instance (e.g. enter http://18.208.167.182/ in Google Chrome to access the chatbot).

Optional: Purchase a custom domain name (e.g. www.teslachatbot.com) for the public IP address of our EC2 instance for user-friendliness.

Google Cloud Run

Google Cloud Run is a managed compute platform that enables developers to run docker containers directly on Google's scalable infrastructure. Cloud Run is designed for simplicity, making it ideal for developers who want to focus on code rather than managing infrastructure. It abstract away most of the complexities of container orchestration and scaling, where often requiring just a single command to deploy.

Deployment steps

1. Install google cloud SDK and configure it correctly
2. Create a new project in google cloud run
3. Write a Dockerfile and .gcloudignore file
4. Use *gcloud build submit –tag gcr.io /{image-name} –project {projec-tname}* to build a docker image on google cloud run
5. Use *gcloud run deploy –image gcr.io /{image-name} –project {projec-tname} –platform managed* to deploy the container image on serverless platform

---

AWS ECS

ECS stands for Elastic Container Service. It is a fully managed container orchestration service provided by AWS. By enabling the Fargate service option, ECS will run docker containers in production by handling cluster management, load balancing, and auto-scaling in serverless architecture.

To activate ECS service, we also need to configure the Elastic Container Registry. Which is like docker hub that stores the container images on cloud, letting it accessible to ECS.

<u>Deployment steps</u>

1. Create a repository on AWS ECR, install AWS toolkits on local terminal
2. Tag the docker image with the url provided by ECR and push to the repository
3. Create a cluster on ECS
4. Create a task definition, which is a blueprint that shows how to run the docker image. Including compute resources needed, expose ports, require services.
5. Create a service within the cluster, configure the security group and deploy.

Insight

<u>AWS EC2</u>
Using EC2 provided complete control over the infrastructure, allowing for custom configurations and optimization of resources.

Scalability: AWS EC2 allows for easy scaling of resources. If the application experiences increased traffic, additional EC2 instances can be launched or existing instances can be resized to handle the load without significant downtime.This approach was cost-effective for consistent and predictable workloads but required more hands-on management, including provisioning, scaling, and monitoring the underlying instances.

Reliability: By using PM2, the application is monitored and automatically restarted in case of failures, enhancing overall reliability. Nginx further contributes to reliability by managing incoming requests and serving static content efficiently.

Performance: Nginx serves static files, reducing the load on the application server and improving response times for users. This optimization is crucial for delivering a smooth user experience in web applications.

<u>AWS ECS Fargate</u>
Fargate's serverless container orchestration simplified infrastructure management by eliminating the need to manage EC2 instances directly. It provides auto scaling and seamless integration with other AWS services. But it comes with higher costs for long-running or resource-intensive workloads compared to EC2.

<u>Google Cloud Run</u>
Cloud Run is more easy to use and provides a similar feature with AWS ECS architecture. Developers can give full control on infrastructure management to cloud run with minimal configuration. However, for stateful or highly complex workloads, it lacked some of the advanced customization which AWS provides.

Conclusion

In this project, we developed a fully functional role-playing chatbot using OpenAI API and enabled user authentication. We explored and compared the deployment of containerized applications using **AWS ECS Fargate**, **EC2**, and **Google Cloud Run**, each offering distinct approaches to managing and scaling workloads in the cloud. Ultimately, the choice between these services depends on the specific requirements of the application, the team's familiarity with the platform, and the desired balance between control, cost, and ease of management.