Deploying and Automating Spring Boot Applications on AWS: A Practical Guide Using Docker and GitHub Actions

Anh Tran

Haaga-Helia University of Applied Sciences
Degree Programme in Business Information Technology
2025

Abstract

Author
Anh Tran

Degree
Bachelor of Business Information Technology

Thesis Title
Deploying and Automating Spring Boot Applications on AWS: A Practical Guide Using Docker and GitHub Actions

Number of pages and appendix pages
55 + 5

This product-based thesis studies deploying and automating a simple pre-built Spring Boot application on Amazon Elastic Container Service (Amazon ECS) with Docker and GitHub Actions. The topic was motivated by the rapid shift of companies to cloud-native microservice architectures and their considerable challenges during the deployment process. The main goal is to create a cohesive step-by-step guideline on deploying the application on Amazon ECS by leveraging containerization and automating the entire process with Continuous Integration and Continuous Delivery/ Continuous Deployment (CI/CD) to eliminate manual work and minimize human errors.

The theoretical framework introduced various key concepts that helped readers better understand the technologies utilized in this project. Those included microservices, containerization with Docker, CI/CD pipelines, and particularly core services of Amazon Web Services (AWS) such as Elastic Compute Cloud (EC2), Amazon Elastic Container Registry (ECR), Application Load Balancer (ALB), and AWS Fargate.

This paper is limited to the Spring Boot framework, Docker, Amazon Web Services (AWS), and GitHub Actions. Other frameworks, containerization tools, cloud service providers, testing practices, and automation tools are out of scope. The target audience includes cloud architects, DevOps engineers, software developers, and companies transitioning to a cloud-based microservices architecture.

The project implementation contained four phases: local setup and Dockerization, AWS configuration, deployment and testing, and CI/CD pipeline automation. The application was successfully containerized with Docker and deployed on Amazon ECS with the Fargate launch type. In addition, the automation workflow using GitHub Actions enabled seamless rebuild and redeployment. The final result reduced manual work, increased the deployment reliability, and provided a replicable solution for similar cloud-native projects. Upon completing the project, the author gained valuable knowledge and hands-on experience with DevOps tools, cloud services, and automation techniques highly demanded in modern software development.

Key words
Amazon Web Services (AWS), containerization, ECS Fargate, Application Load Balancer (ALB), Spring Boot Deployment, Continuous Integration and Deployment (CI/CD)

Table of contents

1 Introduction

1.1 Background and Context

In recent years, the world has seen a massive shift from on-premise architecture towards cloud-based architecture. Cloud computing is a term that anyone has probably heard about at least once, but then what is cloud computing? According to Armbrust et al. (2010, 50), it is defined as the online services provided to users and the underlying infrastructure, including hardware and software, based in data centers that support these services. Instead of purchasing, configuring and maintaining expensive physical servers, enterprises can now rent hardware from various cloud service providers such as AWS, Azure, and Google Cloud Services. Not only does cloud computing improve product performance, but it also facilitates companies' easy scaling up and lowering their operational costs. The rise of cloud computing has significantly changed how software engineers develop and deploy their applications. Traditional monolithic applications have numerous limitations in terms of maintenance, cost, scalability, and security. For that reason, companies are transitioning rapidly to cloud-native applications that promote microservices, containerization, and automation.

1.2 Problem Statement

“Microservices are small, autonomous services that work together” (Newman 2021, 2). In straightforward terms, codebases are divided into numerous small, separate, decoupled, and deployable modules, which can offer several benefits to the system. One of the significant advantages is that developers have more room to experiment with new programming languages, databases, and frameworks without excessive concern regarding the potential disruption of the system. With microservices, risks are minimized since each small piece is independent of the others. However, every benefit comes with a challenge. Microservices can make the process of managing, scaling, and deploying a system more complicated.

1.3 Purpose and Motivation

This is where containerization and automation can assist developers in solving these issues. Containerization technologies, such as Docker and Kubernetes, pack each module with its dependencies into a separate environment, which ensures consistent performance across different stages, including development, testing, and production. Meanwhile, automation through continuous integration and continuous delivery / deployment (CI/CD) pipelines guarantees that each module can be developed, tested, and deployed automatically rather than manually, which can increase the probability of human errors.

This product-based thesis focuses on Spring Boot, a popular microservice-based framework, and how Docker can containerize these microservices to ensure efficient deployment on AWS. Since Spring Boot, Docker and AWS, the leader in cloud service providers, are used widely in several companies, mastering these technologies also improves developers' competitiveness in the rapidly changing technology world. Though there is a variety of documentation regarding microservices, spring Boot, containerization, Docker, deployment, and AWS, there is not yet a cohesive article on how these three technologies are utilized together. Hence, the paper aims to fill the academic gap by providing a comprehensive guideline on how to deploy and automate Spring Boot applications on AWS with Docker and GitHub Actions.

2 Significance of the Project

2.1 Target Audience

This section discusses what audience groups can benefit from this product-based thesis and how the author and the audience can utilize the results. The primary target readers for this paper are cloud architects, software developers, DevOps engineers, or anyone currently building microservices with Spring Boot and searching for a smooth process of deploying applications to AWS with Docker.

In addition, the thesis is helpful for companies in the process of transitioning to a cloud-based microservices architecture by providing step-by-step guidelines on how to deploy Spring Boot applications to AWS using Docker containers. The results of this thesis aim to reduce complications during the deployment stage and enhance the possibility of scaling.

2.2 Organization Benefits

As discussed in the previous section, transitioning from traditional monolithic applications to cloud-native applications offers enterprises countless advantages. Specifically, utilizing Docker and Amazon ECS will lower companies' operational costs, improve management, and facilitate easier scaling. This guideline will provide teams with solid, cohesive documentation to follow and easily automate deploying the Spring Boot application, avoiding unnecessary manual deployment work or potential human errors.

2.3 Author's Learning Outcomes

Lastly, this paper is not only beneficial for readers but also for the author. It offers the author a great chance to learn and gain hands-on experience deploying applications to AWS. This paper will also briefly discuss potential issues that the author might tackle during the process. This should help other readers avoid those mistakes and save time debugging similar problems.

3 Objectives and Scope of the Project

3.1 Objective of the project

The main goal of this project is to present a cohesive step-by-step guideline on deploying and automating a simple Spring Boot application to Amazon ECS (Elastic Container Service) using Docker containers and GitHub Actions. This will provide clear and detailed documentation for anyone who is interested in deployment and efficient automation in CI/CD pipelines. Moreover, this paper also identifies some common deployment issues and how to solve them.

3.2 Scope of the project

The project starts with a brief discussion of microservices architecture and its benefits, followed by a quick setup of a simple Spring Boot microservice, which deepens readers' understanding of microservices structures and functionalities before moving on with containerization. It then presents how to containerize Spring Boot applications into smaller modules with their dependencies using Docker. Lastly, the project handles the automation deployment through CI/CD pipelines with GitHub Actions for building and pushing Docker images to AWS Elastic Container Registry (ECR).

3.3 Out of scope

While this paper involves some topics, including microservices, containerization, and cloud service providers, the following features will not be covered to make sure the paper stays focused, practical, and precise:

- Non-Spring Boot frameworks: Only Spring Boot framework is discussed in this thesis. Other microservice-based frameworks including .NET or Python are out of scope.
- Other cloud service providers: The paper is limited to only Amazon Web Services (AWS) and will not cover other cloud service providers such as Microsoft Azure, Google Cloud, or IBM Cloud
- Testing: Software testing is also out of the scope. This thesis does not cover unit testing, API testing, performance testing or security testing.
- Other CI/CD pipelines: While there are multiple tools to automate the deployment process, this thesis focuses only on GitHub actions. Other CI/CD pipelines such as Jenkins, GitLab CI/CD, Hudson, or Bamboo are out of scope.

3.4 List of Abbreviations

This section introduces a list of key abbreviations utilized throughout this paper. As the project involves deploying and automating the Spring Boot application using several cloud technologies, technical terms are provided in Table 1 to assist readers' comprehension and ensure consistency.

Table 1. List of abbreviations

Abbreviation	Full Term
ALB	Application Load Balancer
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Delivery or Deployment
CLI	Command Line Interface
CPU	Central Processing Unit
DNS	Domain Name System
ECS	Elastic Container Service
ECR	Elastic Container Registry
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IAM	Identity and Access Management
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IP	Internet Protocol
JAR	Java ARchive
JDK	Java Development Kit
JSON	JavaScript Object Notation

PaaS	Platform as a Service
RAM	Random Access Memory
SaaS	Software as a Service
SDK	Software Development Kit
SSL/TLS	Secure Sockets Layer / Transport Layer Security
UAT	User Acceptance Testing
URL	Uniform Resource Locator
VPC	Virtual Private Cloud
YAML	YAML Ain't Markup Language

4 Theoretical Framework

4.1 Cloud Computing and Microservices

4.1.1 Introduction to Cloud Computing

Definition

Microsoft Azure (2025) explains that cloud computing is the distribution of computer resources over the Internet, including servers, databases, storage, networking, software, or analytics. Meanwhile, Mell & Grance (2011, 2) have a different definition of cloud computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” Instead of purchasing and maintaining expensive physical infrastructure, companies can rent those services from various cloud service providers and reduce significant operational costs.

Core concepts

According to Murugesan & Bojanova (2016, 5), cloud computing has key characteristics, including self-service availability on demand, widespread network accessibility, usage-based metering and billing, and multiple-user tenancy. On-demand self-service means users can access, adjust, and manage their cloud resources at their own will, increasing flexibility. Secondly, cloud services can also be accessed from any device, such as laptops, phones, and tablets, if connected to the internet. Also, users pay only for computing resources that are currently used, which helps them scale up the applications or software more quickly if needed. Lastly, numerous users or enterprises share the same cloud infrastructure while the data stays private and protected.

Cloud Service Models

Cloud services are mainly categorized into three models including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Table 2 below will provide more detailed information regarding these three models

Table 2. Cloud service models

Cloud Service Model	Description	Examples
Infrastructure as a Service (IaaS)	Provides physical computing resources such as servers, storages, networks or virtual machines. User will handle their own operational systems and applications	AWS EC2, Google Compute Engine, Microsoft Azure Virtual Machines
Platform as a Service (PaaS)	Provides a complete environment where developers can build, run, test, deploy their applications	AWS Elastic Beanstalk, Google App Engine, Microsoft Azure App Service
Software as a Service (SaaS)	Provides a complete functional software that users can get access through internet without installing.	Microsoft 365, Dropbox, Google Drive

4.1.2 Microservices vs. Monolithic Architecture: Benefits and Challenges

Monolithic and microservice architecture are two primary categories in software architecture, with advantages and disadvantages. Monolithic architecture generally indicates a more traditional approach, where the application is “built as a single unit” (Fowler, 2014). In other words, the logic for handling all requests from client servers is set up within one process, which makes it more natural for developers to build an application. For example, a simple landing page can handle all logic for product listings, sorting, shopping cart, user profile, and payment within one single codebase. While monolithic brings some benefits, it is cumbersome in terms of maintaining the codebase, scaling the application, speeding up the development process, and being flexible to technology evolving.

Meanwhile, microservice architecture solves all the issues the traditional monolithic application poses. The microservice approach breaks down a vast application into smaller pieces, with each handling a single independent service that often communicates through APIs. The diagram in Figure 1 provides readers with a better overview of the comparison between monolithic and microservice applications.

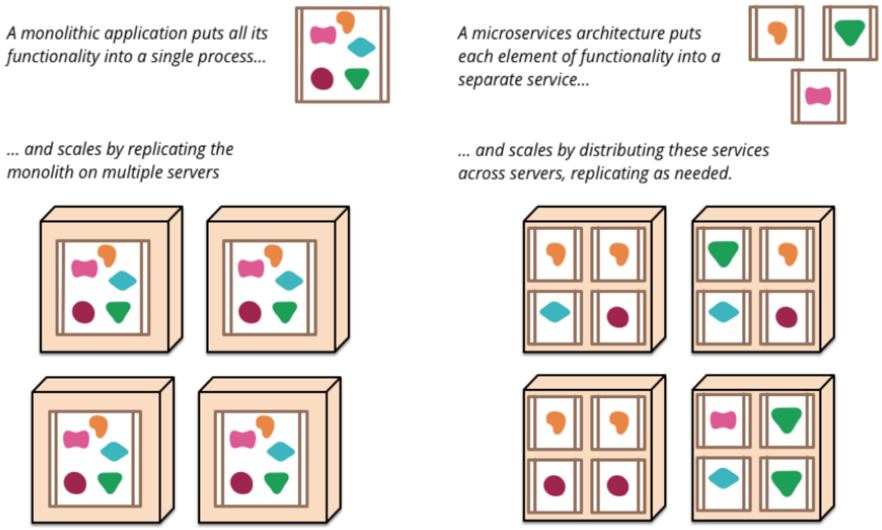


Figure 1. Monoliths and Microservices (Fowler, 2014)

Microservices enable developers to develop faster, maintain the codebase more efficiently, and adapt to changes quickly. However, when it comes to deployment and dependency management, microservices can make the process more challenging. Overall, both monolithic and microservices have their pros and cons, depending on different factors such as application size, scalability potentials, development team expertise, and so on. Implementing microservices requires suitable frameworks and tools. In the next sub-section, we will discuss one of the most popular microservice-based frameworks, which is Spring Boot.

4.2 Spring Boot for Microservices Development

4.2.1 Overview of Spring Boot

Spring Boot, a lightweight Java framework, has been known for “many purpose-built features make it easy to build and run your microservices in production at scale” (Spring Boot 2025). Those features are embedded servers, security measures, metrics, and externalized configuration options. Spring Boot documentation also highlighted that the framework eliminates the need for XML configuration.

Besides, Spring Boot comes with built-in support for dependency management, making it easier for developers to work with libraries and frameworks. It also integrates with Spring Cloud, which provides service discovery, resilience, and distributed tracing for microservices architecture. Such features enable developers to focus on writing business logic rather than working with complex configurations, making Spring Boot the tool of choice for microservices development.

4.2.2 Build Tools for Spring Boot: Maven vs. Gradle

In modern software development, building automation tools is essential in dependency management, code compilation, and efficient application packing. Spring Boot developers frequently use Maven and Gradle to improve project maintainability and accelerate the building process. Each tool has several features suitable to diverse needs during the development process.

Prakash (2022, 87) noted that Gradle offers improved performance and, hence, is the preferred choice for projects requiring flexibility and customization. In contrast, Maven offered a more organized and user-friendly approach with ease of dependency management and configuration. Table 3 provides a comparative analysis of their key features and advantages.

Table 3. Maven vs Gradle - Adapted from Prakash (2022, 87)

Critical features	Maven	Gradle
Configuration	XML-based (pom.xml)	Groovy / Kotlin DSL (build.gradle)
Performance	Slower due to sequential execution	Quicker with incremental builds
Dependency management	Centralized with structured dependencies in pom.xml file	More flexible, supports dynamic dependencies
Customization	Less flexible	Easier to customize applications and support scripting
Ease of use	Easier to learn with user-friendly approach	More concise but with a steeper learning curve

With different strengths, Maven and Gradle are both excellent building tools for Spring Boot applications. Gradle offers greater adaptability and higher productivity through incremental builds, whereas Maven offers an organized and convention-driven approach. The decision between the two is mainly based on the needs of customization, team familiarity, and project requirements. Maven is not popular for new projects because Gradle has become more of a de facto standard.

In addition to building tools, modern Spring Boot applications requires proper containerization for scalability and deployment. Docker is necessary in this situation as it enables developers to bundle software into portable containers together with their own dependencies. The following section examines how Docker improves Spring Boot development by guaranteeing consistency across environments and optimizing deployment.

4.3 Containerization with Docker

4.3.1 Introduction to Docker

Before exploring what Docker is and what the roles of Docker in deployment, it is helpful to get some foundational knowledge regarding containers and containerization. According to Douglis & Nieh (2019, 5), containers are defined as one category of system infrastructure that is typically utilized to facilitate microservices. Containers provide a lightweight and efficient way to package applications into smaller isolated environments that work consistently across diverse platforms. Nevertheless, manually managing containers, particularly configuration, dependencies, networking, and scalability, can be tricky and time-consuming. Docker is a powerful containerization platform that automates the development, packaging, and deployment of applications to overcome those obstacles.

Docker (2025a) stated that it provides a framework and tools to assist with container lifecycle management. Containers are key units for distribution and testing, and developers can build apps and their dependencies inside of these containers.

4.3.2 Key Docker Components

Docker consists of a great number of core components that work closely together to create the complete solution for containerization. Gaining a good understanding of these concepts can help developers to utilize Docker more smoothly during the development and deployment process.

Bashari Rad, Bhatti, & Ahmadi (2017, 228) suggested that Docker comprises of four main components including Docker Client and Server, Docker Images, Docker Registries, and Docker Containers. The diagram below demonstrates Docker’s architecture and how these core components work together to create a Docker system. The visualization shown in Figure 2 helps readers better grasp how these elements interact inside the Docker ecosystem.

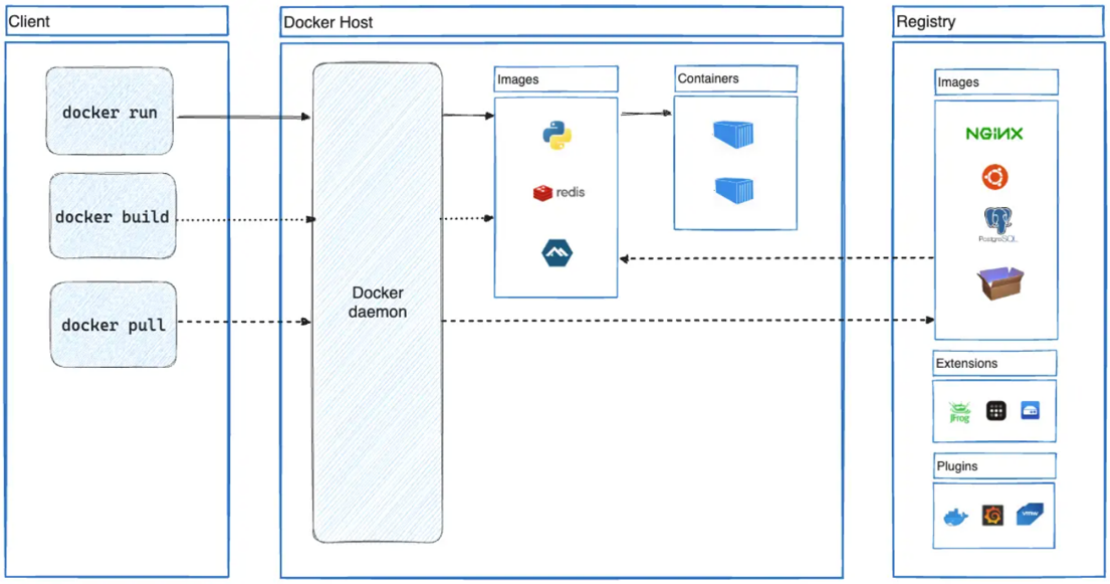


Figure 2. Docker architecture (Docker, 2025)

Docker Client and Server

“Docker uses a client-server architecture” (Docker, 2025a). Users communicate with Docker using the Docker Client interface. It delivers commands to the Docker Sever, also known as Docker Daemon, which executes them, including constructing, launching, and allocating Docker containers. The Docker Client can interact with a remote Docker Daemon via a network or run on the same computer as the Docker Daemon. Given that developers can easily manipulate containers from various settings, this design offer a great amount of flexibility and scalability.

Docker Images

All the files, binaries, libraries, and configurations necessary to execute a container are included in a standardized package called Docker Image. Bashari Rad, Bhatti, & Ahmadi (2017, 229) mentioned that there are two main ways to build an image, which are using a read-only template and creating a docker file.

Docker Registries

According to Docker (2025a), Docker registries are places where all docker images are stored. These registries allow developers to push and pull images for reuse and collaboration. Docker Hub is the most widely used public registry, providing an enormous repository of official and community-contributed images.

Docker Containers

Docker (2025a) defines docker containers as an active and usable version of a Docker image. When a Docker Image is executed, it becomes a container, a very lightweight, isolated environment where the application runs with all the dependencies it requires.

Example Docker Commands

To build and run Docker containers, developers tend to use command-line tools. Figure 3 below demonstrates a few essential commands.

```
# Build an image from a Dockerfile
docker build -t my-application .

# Run a container from an image
docker run -d -p 8080:80 --name my-running-app my-application

# List all running containers
docker ps

# Stop a running container
docker stop my-running-app
```

Figure 3. Essential Docker commands

4.3.3 Benefits of Docker for Microservices

While microservice architecture enables developers to build large-scale applications more easily and quickly by encapsulating software into small isolated functional blocks called microservices, it can become troublesome in the deployment process. Containerization, in general, or Docker specifically, brings considerable benefits to microservice architecture and deployment. Yepuri, Polamarasetty, Donthi and Gondi (2023, 1) highlight that each service of an application can be packed into a separate container image, which decouples it from the system environment. This process makes the deployment less complicated and runs consistently across different platforms without causing any compatibility troubles. Moreover, containers minimize the conflict and dependencies between microservices by offering a significant isolation level.

Containers assist developers in concentrating on managing and updating microservices instead of the system setup or deployment process, as they offer great flexibility, scalability, and consistency (Yepuri et al. 2023, 1). Mhatre (2023, 3) also emphasizes another significant advantage of containers for microservices: resource and cost optimization. Each microservice is encapsulated in an isolated container; if a specific microservice needs to be scaled up to meet unique demands, developers can focus on scaling up just one service instead of the entire application. This leads to better resource usage and cost optimization. Lastly, thanks to the high isolation level, if one of the

microservices is down, it does not pull down the entire software. Hence, developers can considerably reduce the time needed to diagnose and fix the issues, which makes the application more reliable.

4.4 Continuous Integration and Continuous Delivery/Continuous Deployment (CI/CD)

4.4.1 CI/CD Pipeline Overview

Singh, Patel, Raj, Shubham, and Kour (2023, 5218) define CI/CD pipeline as a methodology that "involves the automation of software delivery processes, from development to testing and deployment." The word CI/CD stands for Continuous Integration and Continuous Delivery/Continuous Deployment. In other words, CI/CD enables developers to automate the entire software development process, from updating the codes to the end point of deployment. Therefore, developers are free to update new features and new codes and deliver them more efficiently without worrying about making many errors. CI/CD pipeline for web applications typically includes four steps: Code Development, Continuous Deployment, User Acceptance Testing (UAT), Production Deployment, and Monitoring and Feedback. (Sing et al. 2023, 5218)

Code Development

Without continuous integration, whenever developers make any changes to the source code, and commit to a shared repository, the CI system will automatically build and run to test the application for any unexpected issues. "Continuous Integration refers to the build and unit testing stages of the software release process. Every revision that is committed triggers an automated build and test." (Amazon Web Services 2025a). Mistry (2018, 87) has another definition for Continuous Integration, which means code changes can be merged frequently by developers to the main/master branch, and the test team can then test the newest changes with existing modifications.

Continuous Deployment

In this second step, if new codes pass all tests without any errors, the pipeline automatically deploys the newest version of the application to the staging environment.

User Acceptance Testing

In the third step, testers or users will evaluate the staging environment to ensure the newest version of the application meets the requirements and works properly.

Production Deployment

“Once the new version of the application is approved, it is deployed to the production environment by the pipeline.” (Sing et al. 2023, 5218)

Monitoring and Feedback

During the final step, the pipeline tracks how well the application functions and gives the development team insights to assist them in quickly pinpointing and addressing any problems.

4.4.2 CI/CD Tools for Microservices

Dinu and Ninawe (2025) describe CI/CD tools as software solutions that handle the tasks related to automated code integration, building, testing, packaging, and deploying the application and infrastructure code across different environments. Well-known tools like Jenkins, GitLab CI/CD, and CircleCI provide customizable pipelines that allow for the independent deployment of each microservice.

In this project, GitHub Actions is used as the primary CI/CD tool as it is easy to integrate with GitHub repositories and simple to manage workflows. This tool allows developers to automate numerous tasks such as running tests, building containers, or deploying microservices using YAML configuration files. GitHub Actions offers event-driven automation and the ability to create reusable actions while operating on both GitHub-hosted and self-hosted runners. This flexibility makes it particularly suitable for a microservice architecture (GitHub 2025).

4.4.3 Automating Microservices Deployment

Automation brings numerous advantages to software development. First, Jani (2023, 2984) has mentioned that one of the major benefits of CI/CD is quicker time-to-market, which means automating the processes of building, testing, and deploying cuts down the time needed to roll out new features and updates. Furthermore, CI/CD reduces risk by enabling smaller and more regular releases, which are simpler to handle and have a lower chance of encountering problems.

Microsoft Learn (2025) also emphasizes how CI/CD practices benefit deploying applications. Automation raises productivity by assisting developers in focusing more on developing new features than on manual integration and deployment. Lastly, though proper CI/CD pipelines help developers minimize the number of bugs or errors released, this scenario still occurs. CI/CD will allow for the automated restoration of previous versions of releases.

4.5 Deploying Spring Boot Microservices to AWS

Since microservices-based architectures have become increasingly popular, cloud platforms such as Amazon Web Services (AWS) offer multiple services that refine deploying, scaling, and managing Spring Boot applications.

4.5.1 Elastic Beanstalk and Elastic Compute Cloud (EC2)

AWS Elastic Beanstalk is known as a Platform as a Service (PaaS). According to Amazon Web Services (2025b), Elastic Beanstalk allows developers to deploy and manage their applications on AWS without needing to explore the underlying infrastructure that supports those applications. Due to the abstraction of complex configurations, networks, or scaling, developers can deploy their Spring Boot applications more efficiently and quickly.

When a web server environment is created with AWS Elastic Beanstalk, it launches one or more virtual machines called Instances via Amazon Elastic Compute Cloud (Amazon EC2) (Amazon Web Services 2025c). Saini and Behl (2020) point out that AWS EC2 helps developers eliminate the obligation to purchase hardware by offering a scalable computing capacity in the Amazon Web Services cloud. Once instances are deployed with EC2, it is crucial to handle traffic effectively and ensure the application can adapt to multiple demand levels. AWS has provided users with an efficient solution called Elastic Load Balancing (ELB), which “automatically distributes incoming application traffic across multiple targets and virtual appliances in one or more Availability Zones (AZs)” (Amazon Web Services 2025d).

Alongside load balancing, AWS offers Amazon EC2 Auto Scaling to enhance application performance and cost efficiency. This service automatically assesses the current demand and adapts the number of instances accordingly. It is highly beneficial since it significantly reduces the fault risk, reduces the complication of configuration changes, enhances the efficiency of workload performance, and reduces costs. (Amazon Web Services 2025e).

4.5.2 Amazon Elastic Container Registry (ECR)

As containers have become increasingly popular in modern software development, the need to store, manage, and deploy containers safely and efficiently also increases. AWS Elastic Container Registry (ECR) has proven to be a great solution to this issue as it is “a fully managed container registry offering high-performance hosting” (Amazon Web Services 2025f). Mudasir (2024) explains that thanks to ECR, developers can avoid the hassle of managing container registries, as it smoothly takes care of scaling availability and ensuring the security of their image repositories.

Developers containerize Spring Boot applications with Docker and then push the images to ECR. Once these images are stored in ECR, they can be pulled and deployed to AWS container services, including Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

Container images can be pushed to ECR by either the Amazon Command Line Interface (CLI) or the Amazon Web Service Management Console. The workflow generally consists of several key steps, regardless of the methods used. It starts with building a Docker image, authenticating the local Docker client with ECR, tagging the image, then pushing it to the repository, and lastly pulling from ECS/ECK for deployment. The workflow can be illustrated more clearly in the diagram in Figure 4.

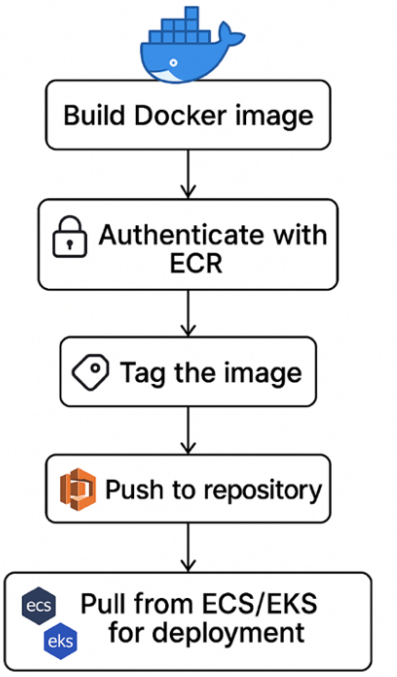


Figure 4. Workflow Diagram: Pushing Docker Images to ECR

Similar to Elastic Load Balancing (ELB), the Application Load Balancer (ALB) is commonly employed in ECS deployments to direct incoming HTTP traffic to the appropriate containers hosting the application. In this project, once the Docker image of the Spring Boot application is pushed to ECR, it is deployed through ECS Fargate. The ALB is then set up to effectively route traffic to the active containers following the guidelines established by the target group.

4.5.3 AWS Fargate: Strengths and Weaknesses

Overview of AWS Fargate

ECS provides users with two launch types: EC2 and Fargate. In contrast to EC2, where developers are required to handle instances, Fagate launch type allows AWS to manage the complex underlying infrastructure. Amazon Web Services (2025l) views AWS Fargate as “a serverless, pay-as-you-go compute engine that lets you focus on building applications without managing servers”. ProsperOps (2024) also points out that since Fargate only requires users to define CPU and memory resources and AWS takes care of the rest of the computing resources, it is easier for developers to concentrate on building and deploying their applications instead of handling complicated servers.

Strengths and Weaknesses

The Fargate launch type offers many benefits for deploying containers on ECS. One great advantage is that Fargate assists users in abstracting the underlying infrastructure. It eliminates the worry of manually providing and managing EC2 instances and enables developers to simplify the deployment process. Moreover, Fargate adjusts the capacity dynamically to match the application demand and manages other work, such as updating operating systems and enhancing compliance (ProsperOps 2024). Another benefit is that Fargate offers built-in security measures like isolation and secure communication between containers.

Nevertheless, Fargate also comes with certain limitations that are worth considering. First, as AWS fully provides and manages compute resources, storage, and networking, it speeds up the deployment process but offers less room for customization. In contrast to EC2, which has over 750 instances to select from, the options to customize CPU and memory with Fargate are far more limited. Second, according to Amazon Web Services (2025l), Fargate launch type charges users for only the time that the resources are consumed to run the task. While it tends to be more convenient and faster to deploy a containerized application on AWS with Fargate, this launch type typically costs more than EC2 for the same workload.

Fargate vs EC2

Table 4 gives readers a broad overview of Fagate’s and EC2’s characteristics. Knowing their pros and cons is crucial to deciding which launch type is best for application purposes.

Table 4. Fargate vs EC2

Feature	Fargate	EC2
Server management	Serverless (fully provided and managed by AWS)	Manually providing and managing servers
Customization	Limited options with CPU and memory	Over 750 instance types to select and complete host control
Cost	Task-based per second	Pay per instance regardless of usage
Scaling	Auto scale to meet requirements	Require setup for auto scaling
Use case	Suitable for simple and stateless applications such as microservices	Suitable for long running and stateful applications that require full control of servers

With all the advantages discussed, this project opted to deploy the stateless containerized Spring Boot application on ECS with the Fargate launch type to reduce the complications of the deployment process, minimize the manual work with servers, and improve the scalability.

While Elastic Beanstalk abstracts much of the infrastructure and speeds up the deployment process significantly for developers, this project chose to use ECS with the Fargate launch type for its customization, flexibility, and more control over container behaviors and the CI/CD process.

5 Empirical Section

This empirical section will begin by introducing the desired outcome of the project and providing an overview of the pre-built application. Subsequently, the different phases of the project will be described in detail, including the technologies applied, and the corresponding code base.

5.1 Project Overview

This project would deploy a pre-built Spring Boot application using Docker on Amazon Web Services. The main goal is to create a publicly accessible web application hosted on AWS using ECS with Fargate, which simplifies server management by using containers to deploy applications without the need to manage EC2 instances. In addition, the entire application set up to be built, containerized, and deployed automatically through a CI/CD pipeline utilizing GitHub Actions.

Application structure

The pre-built Spring Boot application was a simple web application that offered some basic endpoints for displaying messages and interacting with users through query parameters. The Spring Boot application served plain text responses for demonstration purposes. The structure of the project is demonstrated in Figure 5 below.

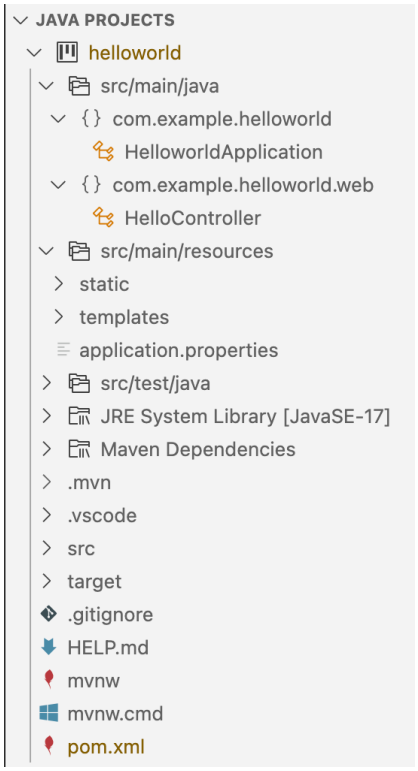


Figure 5. Spring Boot Project structure

Application Functionality

The Spring Boot project contained a HelloController class in the web package. This class defined the application's three endpoints, including:

- Main Page (GET /index) – The page showed the text “This is the main page!”.
- Contact Page (GET /contact) – The page showed the text “This is the contact page.”
- Hello Page (GET /hello) – This page accepted two query parameters: location and name. It then displayed the hello message dynamically.

The source code of HelloController class is demonstrated in Figure 6.

src > main > java > com > example > helloworld > web > J HelloController.java > ...

```java
package com.example.helloworld.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@ResponseBody
public class HelloController {
    @GetMapping("/index")
    public String index() {
        return "This is the main page!";
    }
    @GetMapping("/contact")
    public String contact() {
        return "This is the contact page";
    }
    @GetMapping("/hello")
    public String hello(@RequestParam(name = "location") String location, @RequestParam(name = "name") String name) {
        return "Welcome to the " + location + ", " + name + "!";
    }
}
```

Figure 6. HelloController class

This application was built with Maven, a popular tool for building and managing Java-based projects. Therefore, the structure also included an important configuration file, pom.xml, which held the project metadata (groupId, artifactId, version, name), all dependencies (spring-boot dev tools, spring boot starter web,…), and plugins for bundling the application into a runnable .jar file.

```xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

Figure 7. Snippet from pom.xml file

Figure 7 displays parts of the pom.xml file with test dependencies and the Spring Boot Maven plugin. After running 'mvn clean package', Maven enabled us to pack the application in the .jar file, which then became the key artifact for Docker image.

5.2 Project Phasing

To achieve the final product, the project went through four main phases:

- Phase 1: Local setup and Dockerization
- Phase 2: AWS Configuration
- Phase 3: Deployment and Testing
- Phase 4: CI/CD pipeline automation

5.3 Project Implementation

5.3.1 Phase 1: Local setup and Dockerization

The primary objective in phase 1 was to have the Spring Boot application prepared for deployment in the container. The process included setting up the development environment, code editor Visual Studio Code (VSCode), installing Docker on the local machine, and having Java and Maven readily available. Once set up, the application was run locally and tested to confirm that all functionalities worked as expected. It was packaged into a JAR file in the target folder using Maven and configured to run within the Docker image. At the end of this phase, the Spring Boot application was ensured to be built and run reliably in an isolated environment, ready for the next step, deployment to AWS.

Development environment setup

Setting up the following tools properly before deploying the Spring Boot application on AWS is essential.

- Java Development Kit (JDK 17): JDK17 was needed to compile and run Java-based projects. It was downloaded from the Oracle website or by using a package manager such as Homebrew (macOS) with the command: brew install openjdk@17
- Maven: Maven built the project and managed all dependencies. It can also be installed through Homebrew (macOS) with the command: brew install maven
- Docker Desktop: Docker was downloaded from the Docker website. It is available for three operating systems including Windows, macOS, and Linux. This project used Docker to build local images and test containers.

- VSCode: VSCode was used as the main code editor

Local Testing of Spring Boot application

After finishing setting up the development environment, the Spring Boot application was run locally to test its functionality using the Maven command ‘mvn spring-boot:run’. The application was executed successfully on embedded Tomcat server with the port 8080, which was shown in the console as Figure 8.

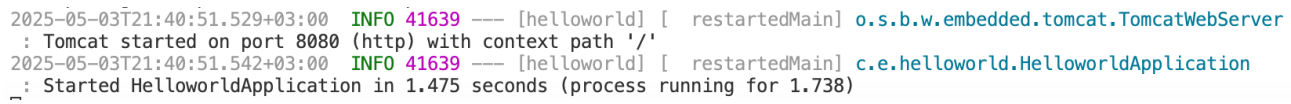


Figure 8. Console output confirming successful local start of Spring Boot application

At this point, the endpoints /index, /contact, and /hello were all accessible via any web browser. For example, the result from successfully assessing the localhost:8080/index endpoint was the simple text message “This is the main page!” as Figure 9.

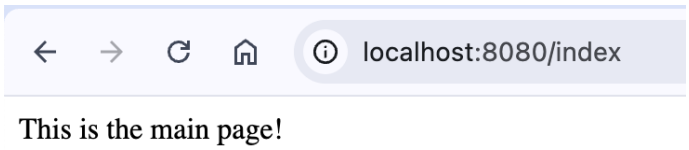


Figure 9. Browser output of accessing localhost:8080/index

Application Configuration File

The application.properties file should be configured with the following settings to ensure that the Spring Boot application worked appropriately both on the local machine and in the containerized environment. Navigate to the application.properties file in the src/main/resources directory and include the following properties shown in Figure 10.

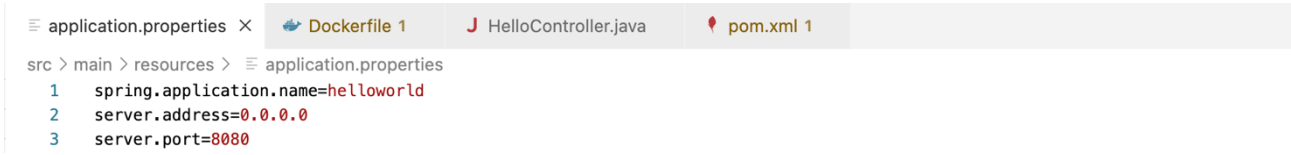


Figure 10. application.properties file

Assigning the server.address to 0.0.0.0 made sure that this Spring Boot application listened to connections from all available networks. This setting was crucial, especially during the execution process inside Docker containers.

Building the application JAR

According to Oracle (2025), JAR stands for Java ARchie, and this zip-format file is used to combine multiple files into one single. Building the application JAR was required not only to compile the source code but also to package all dependencies needed to run this Spring Boot application. In the root directory of the project, the Maven command ‘mvn clean package’ was executed to clear the previous builds in the folder target (clean), then package source code with all dependencies into the JAR zip file inside the folder target.

After running the command, a new JAR file named helloworld-0.0.1-SNAPSHOT.jar was created inside the /target directory, as presented in Figure 11.

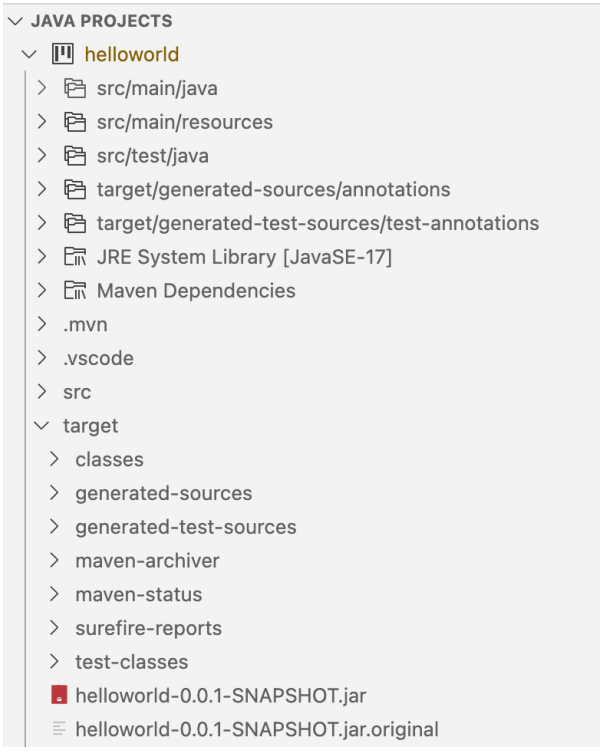


Figure 11. Structure of target folder

In the next step, this JAR file would be used to create the Docker image.

Creating the Dockerfile

The Dockerfile defines how our Spring Boot application should behave inside the Docker container. It is highly important that this file is created in the root directory of the project, at the same level as the pom.xml file, as Docker uses the context of the current directory during the building process. If the Dockerfile is placed in the wrong directory, the image build will fail as the necessary files are not found.

Navigate to the root directory of the Spring Boot application “helloworld”, select the plus sign, and choose “File” to create a new file. Input the name as “Dockerfile” in the bar and press “Enter”. Once this step was done, the project structure should look similar to the one shown in Figure 12.

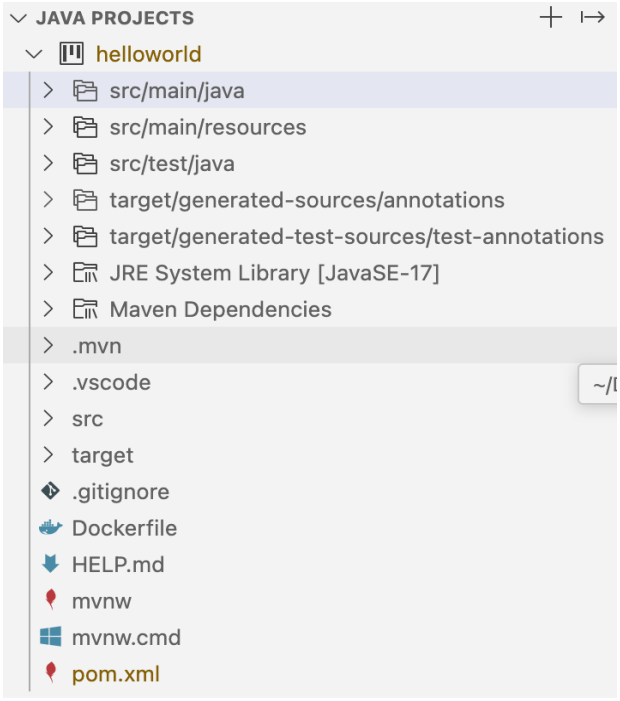


Figure 12. Dockerfile location in project directory

The Docker file used in this project was presented in Figure 13 below.

Figure 13. Dockerfile code

The Docker instruction FROM set the OpenJDK image as the base for the container. The WORKDIR instruction defined the working directory inside the container, which was set to /app. The COPY instruction copied the JAR file from the local target directory into the container. EXPOSE documented the port that the application listened on at runtime. Finally, ENTRYPOINT defined the command that will be executed when the container started, which, in this case, ran the Spring Boot application using the java -jar command.

Building and running the Docker image locally

Once the Dockerfile was ready, the next step was to build the Docker image from the created Dockerfile. This process was done by using the command ‘docker build -t helloworld .’ in the root directory. In this step, the command instructed Docker to build the image with the current directory and tag it (-t) as helloworld. Docker (2025b) explains that “when you run a build, the builder pulls the base image, if needed, and then runs the instructions specified in the Dockerfile.”. This means that all the declared instructions in the Dockerfile were executed during the building process. It pulled the base image, set the working directory, copied the JAR file from /target folder to the image, exposed the port 8080, and set up the startup command for the container.

This project was built and run on a macOS machine, which typically runs on the ARM64 architecture, while AWS Fargate uses Linux/AMD64 architecture. This results in an incompatible issue during the deployment process. Therefore, in this case, the Docker build command should be slightly different as ‘docker build --platform linux/amd64 -t helloworld .’ The --platform linux/amd64 flag forced Docker to build an image on the linux/amd64 architecture, which avoided runtime error when deployed on AWS Fargate. Figure 14 illustrated that once Docker was complete building, the console displayed messages to confirm the successful build.

```
[+] Building 1.9s (9/9) FINISHED                                                                    docker:desktop-linux
 => [internal] load .dockerignore                                                                                   0.0s
 => => transferring context: 2B                                                                                     0.0s
 => [internal] load build definition from Dockerfile                                                                0.0s
 => => transferring dockerfile: 395B                                                                                0.0s
 => [internal] load metadata for docker.io/library/openjdk:17-jdk-slim                                              1.8s
 => [auth] library/openjdk:pull token for registry-1.docker.io                                                      0.0s
 => [1/3] FROM docker.io/library/openjdk:17-jdk-slim@sha256:aaa3b3cb27e3e520b8f116863d0580c438ed55ecfa0bc126        0.0s
 => [internal] load build context                                                                                   0.0s
 => => transferring context: 160B                                                                                   0.0s
 => CACHED [2/3] WORKDIR /app                                                                                       0.0s
 => CACHED [3/3] COPY target/helloworld-0.0.1-SNAPSHOT.jar app.jar                                                  0.0s
 => exporting to image                                                                                              0.0s
 => => exporting layers                                                                                             0.0s
 => => writing image sha256:d040fc88c1f2e5225e030a68cdfb8ee99da4e7f1467bb9c152068cdb6289d11b                        0.0s
 => => naming to docker.io/library/helloworld                                                                       0.0s
```

Figure 14. Console output confirming successful Docker image build

The final step in this phase was to test if the Spring Boot application behaved as expected in the Dockerized environment by running the command ‘docker run -p 8080:8080 helloworld’. The -p 8080:8080 mapped the container port 8080 to the port 8080 on the local machine and Docker would run the container which contained the image tagged as helloworld. This command enabled the application available at http://localhost:8080 on the web browser.

The console logs generated by the Docker run command were closely similar to the output from running `mvn spring-boot:run,` indicating that the application started up properly in the containerized setup. All endpoints /index, /contact, /hello were also tested successfully on the web browser.

5.3.2 Phase 2: AWS Configuration

As the application was Dockerized and run properly in the containerized environment, it was well-prepared for the deployment. In phase 2, the main goal was to set up the crucial AWS services that hosted the Spring Boot project. This stage included creating the ECR Repository, authenticating the Docker client with ECR, creating ECS Cluster, defining ECS Task Definition, setting up the networking, and setting up Application Load Balancer (ALB).

Configurations and deployments on AWS can be performed by either Amazon Management Console or AWS Command Line Interface (CLI). This project chose to utilize Amazon Management Console, a graphical user interface, as it was more visual and accessible, particularly to individuals who are not familiar with command-line operations.

Creating an ECR Repository

To begin the creation of an ECR repository, an AWS account was required. The ECR repository served as storage for the previously created Docker image named 'helloworld'. First, access the Amazon Elastic Container Registry (ECR) via the Amazon Management Console using a web browser at https://console.aws.amazon.com/ecr. In the console, navigate to the left hamburger menu icon, then select ‘Private Registry’ followed by ‘Repositories.’ This section managed all ECR private repositories and offered the option to create a new repository. Click the orange button labeled ‘Create Repository’ in the console's upper right corner to create a new private repository.

In the creation form, the repository name is filled with ‘helloworld’ to stay consistent with the Docker image tag. All other settings were left as default. Once the repository was successfully created, it appeared in the repository list within the ECR console, as shown in Figure 15.

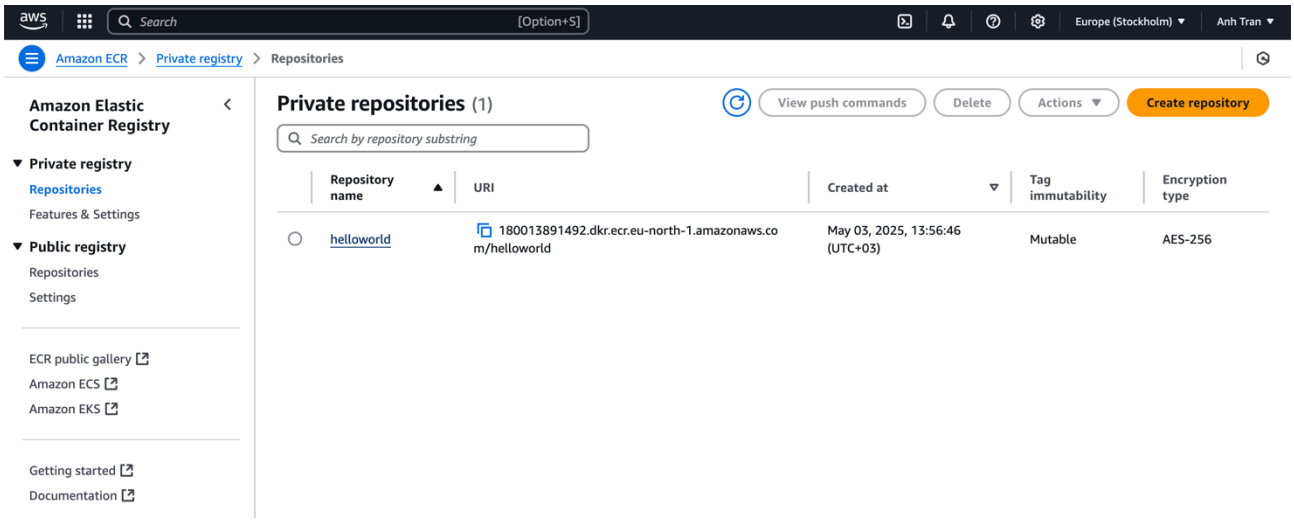


Figure 15. ECR Console with the created helloworld repository

Authenticating Docker with ECR

Authenticating Docker client was crucial as it helped Docker to communicate securely and effectively with the Amazon ECR service. In the Amazon Management console, select the created repository ‘helloworld’, then select the orange button “View Push Commands”. As depicted in Figure 16, it displayed all essential commands to authenticate and push an image to the repository.

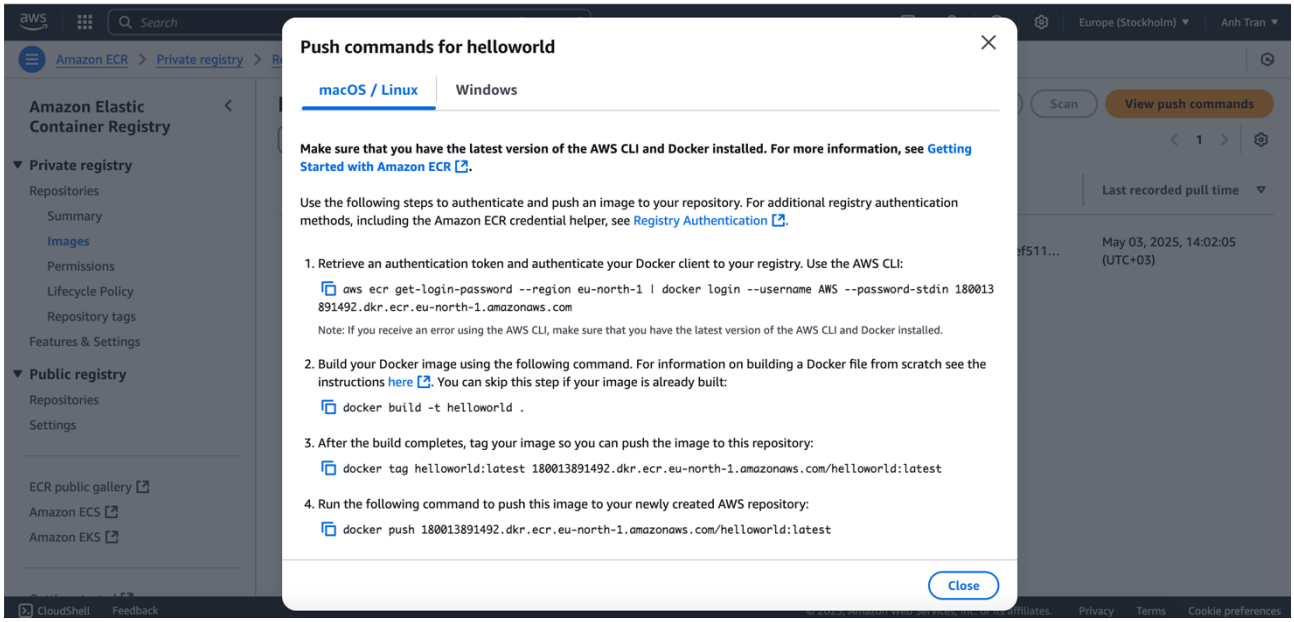


Figure 16. Push commands for ‘helloworld’ repository

In this step, the first command ‘aws ecr get-login-password --region eu-north-1 | docker login --username AWS --password-stdin 180013891492.dkr.ecr.eu-north-1.amazonaws.com’ was copied and executed using AWS CLI. This command authenticated Docker to have proper access to Amazon ECR. Upon successful authentication, a confirmation message ‘Login Succeeded’ was returned.

Tagging and Pushing the Docker Image to ECR

As the Docker image had already been successfully built in Phase 1, the second command in the list of push commands for helloworld was skipped. In this step, the Docker image was tagged and pushed to the repository ‘helloworld’ through the AWS CLI. The third command ‘docker tag helloworld:latest 180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld:latest’ aligned the local image name with the ECR repository name. After completing tagging, the last step was to push the image tagged as ‘helloworld’ to the repository using the fourth command ‘docker push 180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld:latest’. The result of the successful execution is displayed in the console output, as presented in Figure 17.

```
Using default tag: latest
The push refers to repository [180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld]
36044bbb9758: Pushed
```

Figure 17. Console output confirming successful image push to the ECR repository

Creating an ECS Cluster

Amazon Web Services (2025g) defines an ECS cluster as “a logical grouping of tasks or services”. A cluster can function as an environment for ECS tasks to run inside. Amazon Web Services (2025b) describes Amazon Elastic Compute Cloud (Amazon EC2) as “virtual machines, known as Instances”. In other words, Amazon EC2 behaves exactly like a computer in the cloud, and AWS allows users to configure its specifications, such as CPU and memory. Since using EC2 instances to host containers requires lots of managing and handling underlying infrastructure, this project used Fargate launch type to eliminate the need to manually provide and maintain the server.

To create a new cluster, first navigate to the ECS console on the web browser at https://eu-north-1.console.aws.amazon.com/ecs, and select “Clusters” in the menu on the left-hand side. This section managed all created clusters and provided the option to create a new cluster. Then, click on the orange button “Create cluster” on the right-hand side of the browser. In the cluster creating form, the cluster name field was filled with “helloworld”, and “AWS Fargate (serverless)” was selected as the infrastructure to remove the need for EC2 instances. All other settings were left as default. As can be observed in Figure 18, after successful creation, the “helloworld” cluster showed up in the cluster dashboard.

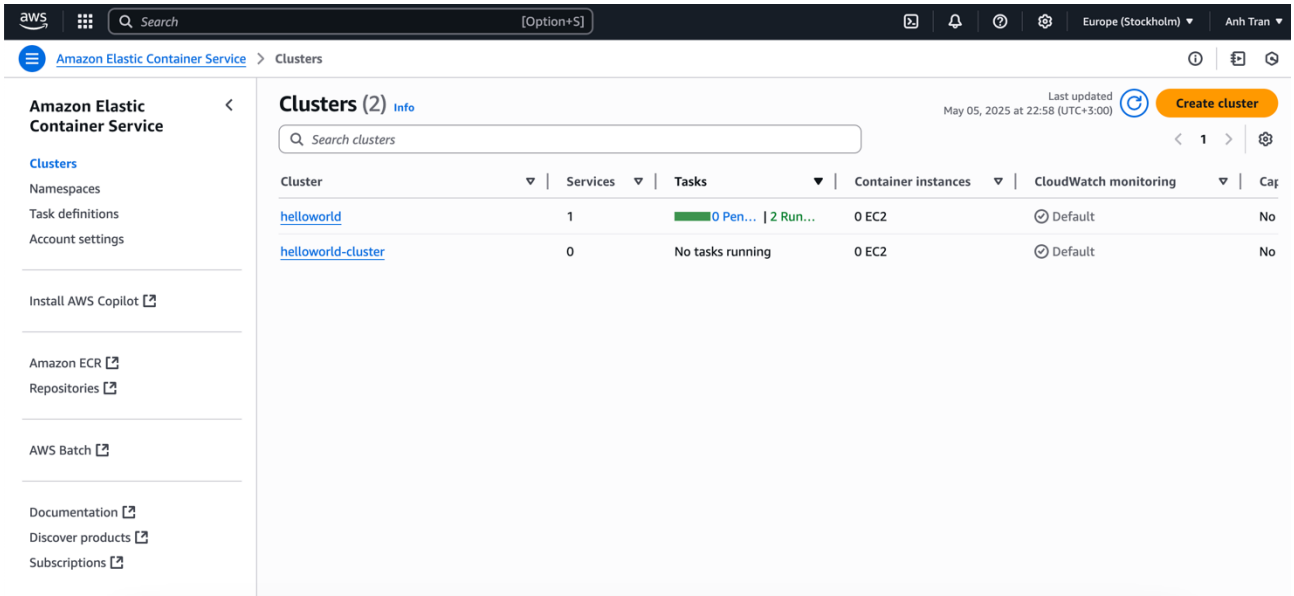


Figure 18. ECS Console with the created “helloworld” cluster

Defining the ECS Task Definition

Once the cluster was set up, the next step was to create a task definition, which is “a blueprint for your application” (Amazon Web Services 2025h). In this project, the task definition defined how the Docker container should run on ECS. Amazon Web Services (2025h) states that a task definition requires users to specify parameters such as launch type, Docker image, CPU, memory, and logging configuration.

This step continued to work in the ECS console to create a new ECS task definition. Navigate to the left-hand side menu and select “Task definitions”. Then, on the right-hand side of the browser, click the orange button “Create new task definition”. It showed a dialog to choose between “Create new task definition” and “Create new task definition with JSON”. Select the first option to continue with a graphical view.

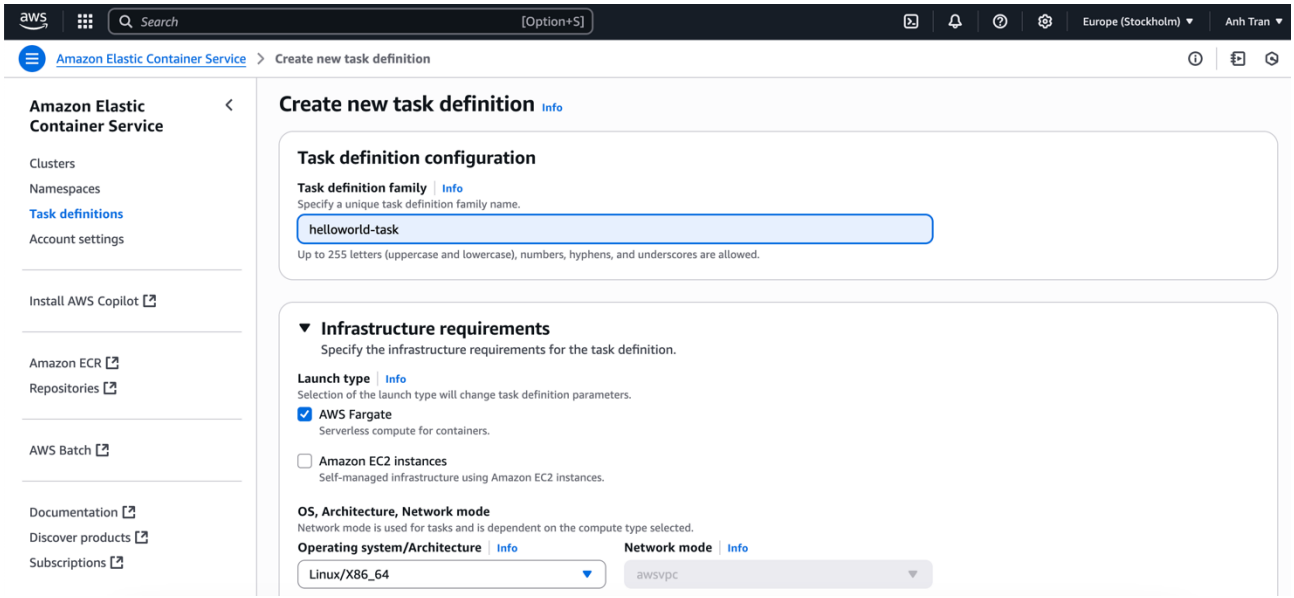


Figure 19. Create new task definition form

In the form to create a new task definition shown in Figure 19, the following important fields were required to be filled with:

- Task definition family : “helloworld-task”
- Launch type: select “AWS Fargate”
- Task role: ecsTaskExecutionRole
- Container name: “helloworld”
- Image URI: 180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld (repository-uri/image:tag)
- Container port: “8080”

- Logging: select “Use log collection”
- Other fields are kept as default

Task definition family allowed users to group many versions of the exact task definition for easier control. Choosing “AWS Fargate” as the launch type eliminated the requirement for manually providing and maintaining servers. In the creating form, setting the task role as “ecsTaskExecutionRole” was highly important as this role permitted the task definition to interact safely with other AWS services, including Amazon ECR. In other words, this setup allowed ECS to pull images from ECR for deployment. Providing the correct container name and image URI defined which Docker container should be deployed through port 8080. Lastly, enabling “Use log collection” was necessary to monitor the container’s runtime behavior, which helped developers save time in troubleshooting or adjusting performance.

After ensuring all key fields were set up properly, scroll down to the bottom of the creating form and click the orange button “Create” to initiate a new task definition. Once the action was done successfully, the “helloworld-task” task definition was displayed in the Task definitions dashboard as Figure 20 demonstrated below.

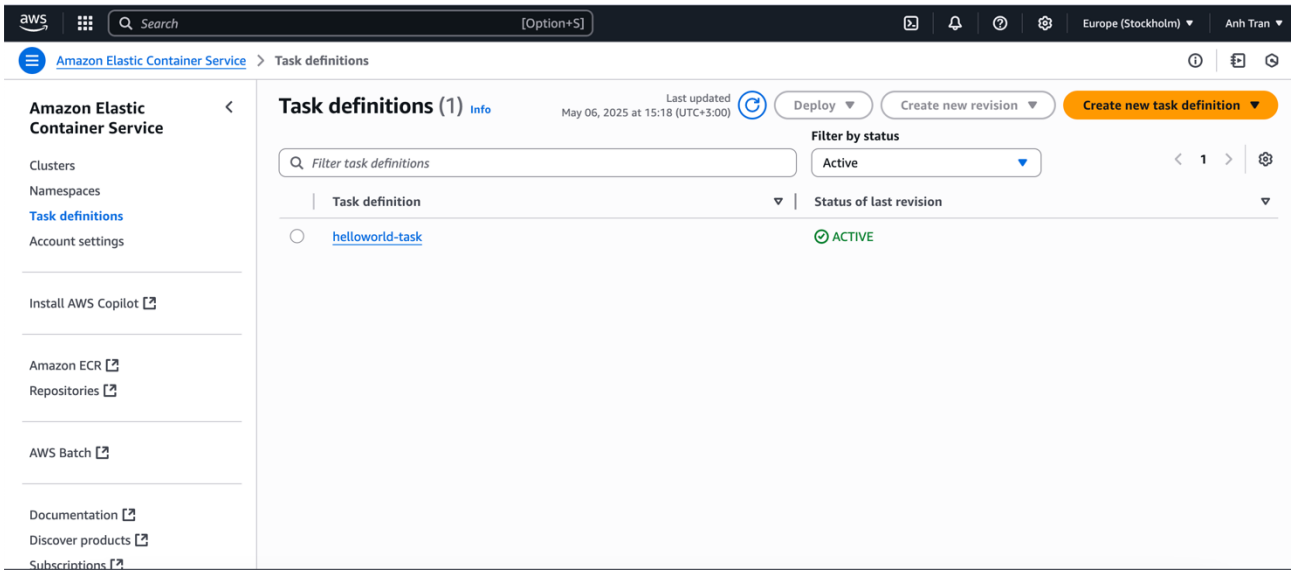


Figure 20. ECS Console with the created “helloworld-task” task definition

In the third phase of deployment and testing, this “helloworld-task” task definition would be later used to launch a service or a task within the “helloworld” cluster.

Configuring Networking

After completing the creation of an ECR repository, an ECS cluster, and a task definition, the following key action in the AWS configuration phase was to set up the networking properly so that

users can have secure access to the deployed Spring Boot application. This point includes setting up the Virtual Private Cloud (VPC), defining the public and private subnets, setting up route tables, and managing security groups for traffic control.

According to Amazon Web Services (2025i), a Virtual Private Cloud (VPC) is an isolated virtual network where developers can easily launch AWS resources such as ECS services or load balancers. A key component of VPC is called a subnet, which is “a group of IP address range” (Bari, Khan, Samrin, & Khare 2024, 1791). They further describe that subnets are classified into two types: public subnets and private subnets. While a public subnet allows access from the Internet via the Internet Gateway, a private subnet restricts access to only internal communication.

Routing tables are responsible for directing the traffic flow between subnets in the VPC and other networks. In AWS, while VPC covers numerous Availability Zones, each subnet must stay within one single Availability Zone (Amazon Web Services 2025i). Each subnet is linked to one routing table holding a set of rules to determine how the network traffic should be routed.

Each AWS account is provided with the default VPC along with its preconfigured networking components, such as subnets, a gateway, and routing. Though AWS offers users the ability to create additional VPCs with custom components, this project utilized the default VPC to simplify the deployment setup process. In the next deployment phase, the ECS service in this project would be deployed in the private subnets to guarantee security. In contrast, the Application Load Balancer (ALB) would be launched in the public subnets so that users can access it online.

The final networking configuration for a smooth deployment on AWS is the use of security groups. These perform as virtual firewalls that mange the inbound and outbound traffic for AWS resources such as ECS services or ALB (Amazon Web Services 2025j). In simple terms, the security groups determine which types of traffic can get access to or exit a resource. They can be imagined as the door guards deciding who can come in and who cannot. This project also handled the security groups through the Amazon Management Console.

To access the EC2 console, visit https://eu-north-1.console.aws.amazon.com/ec2/ in a web browser. Scroll down the menu on the left-hand side to navigate the “Network and Security” section, select “Security Groups”. This led to the “Security Groups” view with the list of created security groups and the option to create a new one.

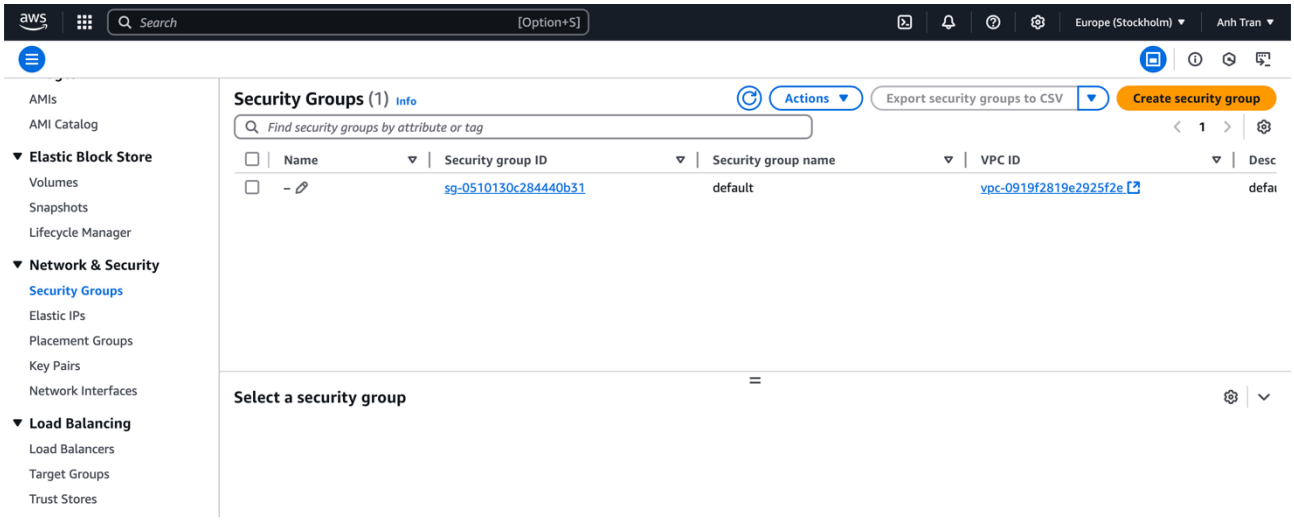


Figure 21. EC2 Console with Security Groups view

The Application Load Balancer (ALB) in this project was associated with the default security group “sg-0510130c284440b31 – default” shown in the Figure 21. One inbound rule was manually created to ensure that the application was accessible over the Internet. Select the security group and navigate to the tab “Inbound rules”. After that, click on the button “Edit inbound rules” on the right-hand side of the tab to add more rules. Click on the “Add rule” button and fill in the details shown in Figure 22 as follows:

- Type: HTTP
- Port range: 80
- CIDR blocks: 0.0.0.0/0

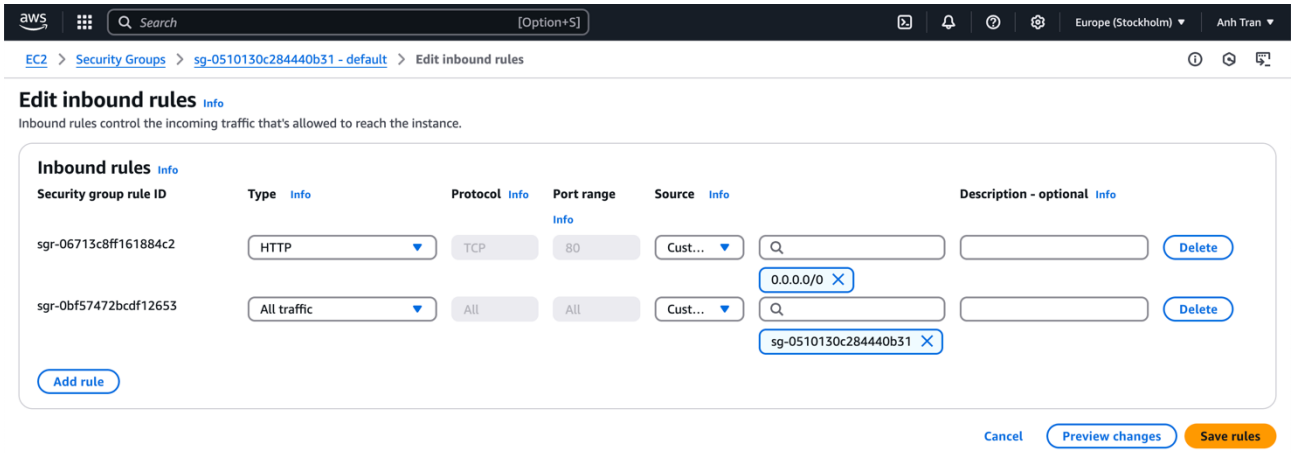


Figure 22. Edit Inbound rules view

This rule set HTTP traffic on port 80 available for users to access the Application Load Balancer (ALB). This default security group was used for the ALB and another security group was created

for the ECS services. In the “Security groups” dashboard shown in Figure 23, select the button “Create security group” and fill in the create security group form with the fields as follows:

- Security group name: ecs-service-sg
- Description: Inbound 8080 from ALB for ECS task access
- Inbound rules: Customer TCP – Port range 8080 – Source as sg-0510130c284440b31 (the Security Group ID of the ALB)
- All other settings were left as default

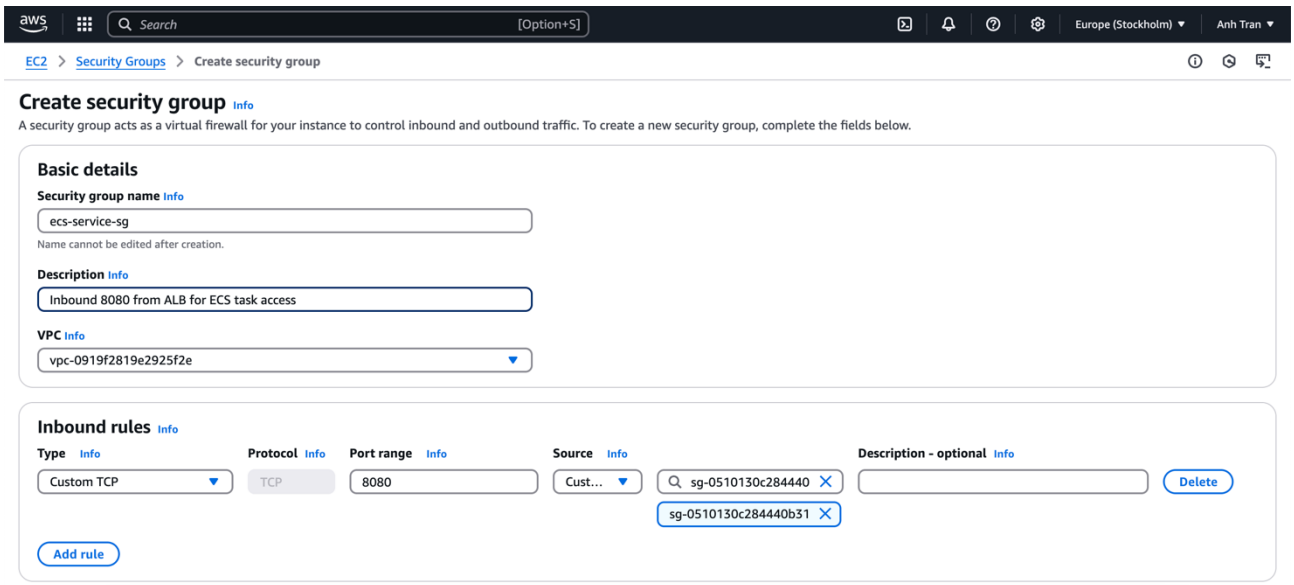


Figure 23. Create security group form

This inbound rule permitted traffic to port 8080 and enabled the ALB to forward HTTP requests to the ECS task executing the Spring Boot application. With these two security groups properly configured, each AWS resource used in the project could be communicated securely.

Setting up Application Load Balancer (ALB)

The last step in phase 2 was to set up the Application Load Balancer (ALB). Reddy and Naidu (2024, 149) suggest that load balancers are critical in managing traffic between clients and servers. More specifically, when HTTP/HTTPS requests are sent from the client, load balancers spread those among multiple servers and ensure no server is overwhelmed or bottlenecked. In this project, the ALB distributed user received requests to the ECS service running the Spring Boot application. In addition, ALB also generated a public endpoint that users can access through HTTP.

A prerequisite before creating an ALB is the creation of a target group. A target group uses a protocol and port that users define to route the client requests to the registered target, such as EC2 instances. (Amazon Web Services 2025k). One target group should be used only with one load

balancer. In this project, the target group oversaw the redirection of HTTP requests from the ALB to ECS tasks running on the private subnet. It also provided a health check path for AWS to monitor the application's health. A new target group was created using the Amazon Management Console. Similarly to setting up security groups, this step initially navigated to the Amazon EC2 console at https://eu-north-1.console.aws.amazon.com/ec2 and scrolled down the left-hand side menu to select “Target groups” in the “Load Balancing" section. This led to a view of created target groups, along with the option to create a new one. Click on the orange button “Create target group” on the right-hand side of the view.

In the “Create target group” form, specify the details as follows:

- Basic configuration: IP addresses (as Fargate tasks do not have EC2 instances IDs, traffic is routed by IP)
- Target group name: helloworld-tg
- Protocol: HTTP
- VPC: select the default VPC
- Health check protocol: HTTP
- Health check path: /index

After all those fields were filled correctly, select the “Next” button and select “Create target group”. As illustrated in Figure 24, once the target group was successfully created, it showed up in the “Target groups” view.

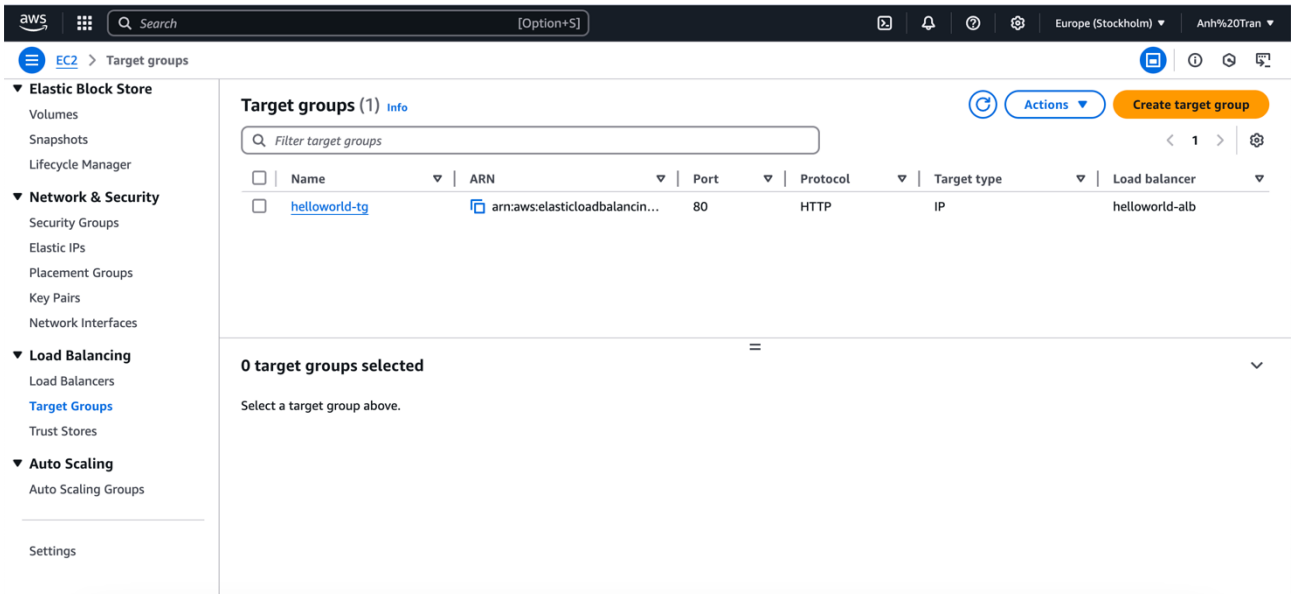


Figure 24. Target group view with the created “helloworld-tg” target group

The target group details could be easily checked by double-clicking the target group name. Next, create a new ALB by navigating to the same “Load Balancing” section and clicking “Load

Balancers". Then, in the "Load Balancers" dashboard, click on the orange button "Create load balancer" in the "Load Balancers" view. Amazon Web Services then took users to the page where they could select a type of load balancer: Application Load Balancer, Network Load Balancer, or Gateway Load Balancer. In this case, choose the button "Create" in the "Application Load Balancer" tab.

The "Create Application Load Balancer" form was displayed, as shown in Figure 25. The configuration fields should be filled as follows:

- Load balancer name: helloworld-alb
- Scheme: Internet-facing
- Load balancer IP address type: IPv4
- Network mapping: select at least two Availability Zones for high availability (eu-north-1a, eu-north-1b, eu-north-1c)
- Security group: select the default security group in the previous step (sg-0510130c284440b31)
- Listener: HTTP – port: 80
- Default action: "helloworld-tg"
- All other fields were kept as default

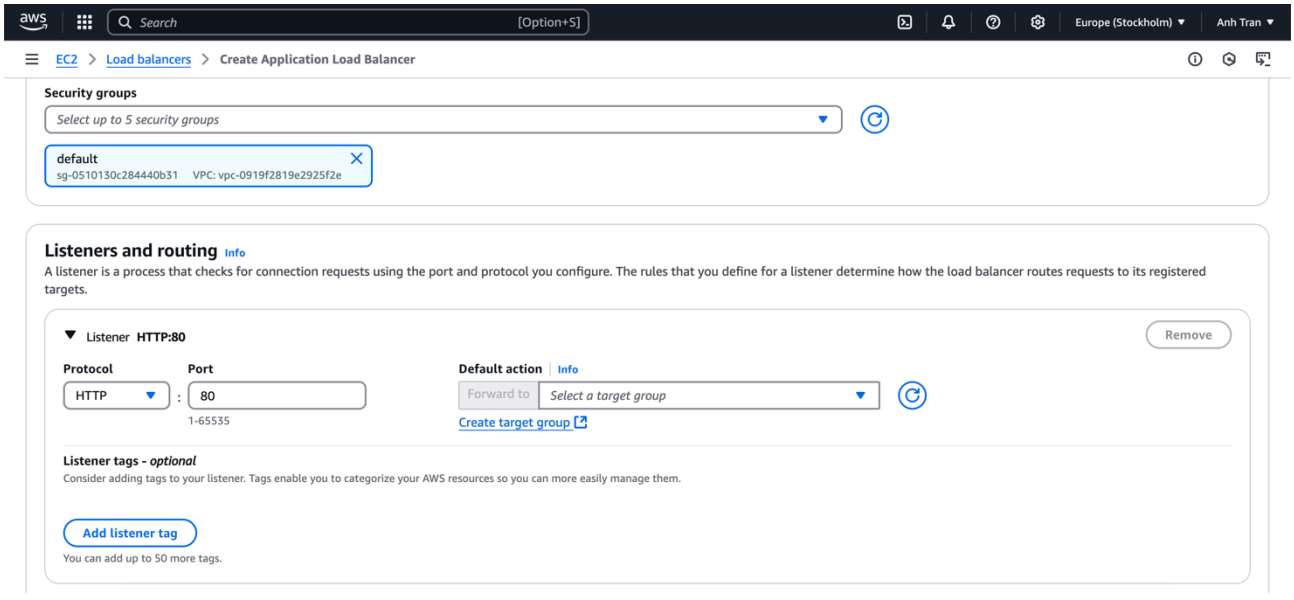


Figure 25. Create Application Load Balancer form

Scroll down to the bottom of the form and click on the button "Create load balancer". After a few moments, a newly created ALB named "helloworld-alb" appeared in the dashboard. This confirmed that the ALB was successfully deployed and prepared to handle HTTP requests.

With this final step of creating ALB, the project was complete with the AWS configuration in the second phase. The project could now move on to the third phase, which concentrated on deploying the containerized Spring Boot application through the defined ECS and ALB environment.

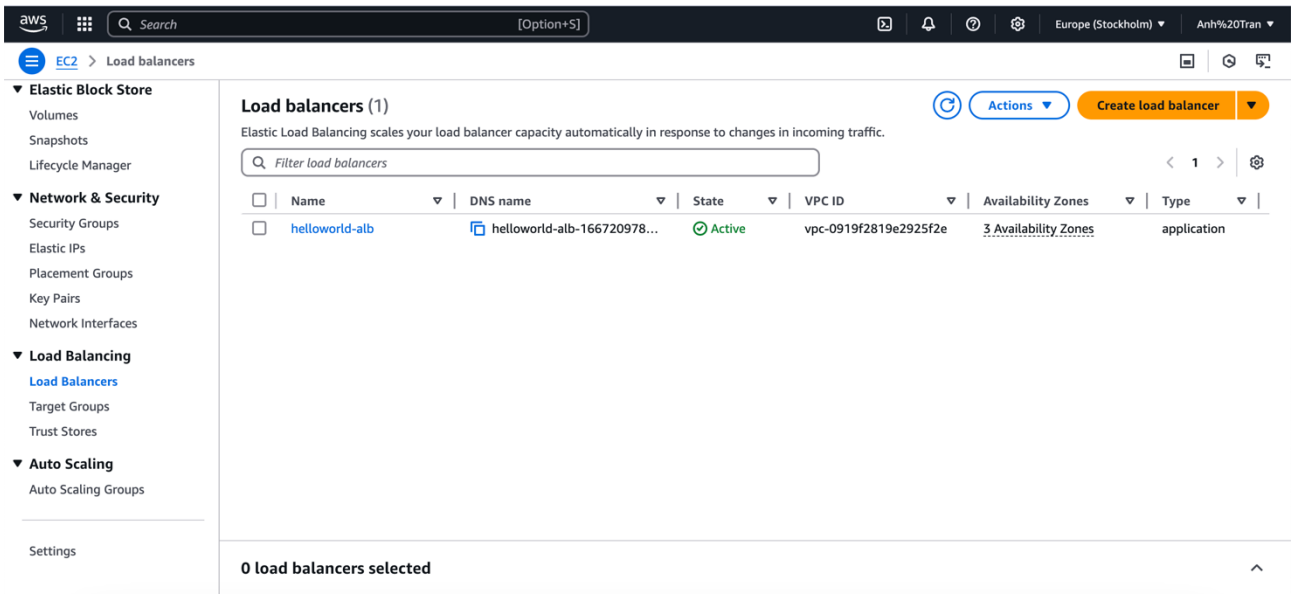


Figure 26. Load balancers view with the created “helloworld-alb”

To conclude phase 2, the following configurations were implemented correctly to prepare the AWS environment for deployment:

- The Docker container image “helloworld” used in this project was successfully built and pushed to Amazon Elastic Container Registry (Amazon ECR).
- An ECS cluster “helloworld” was created to serve as an environment for ECS tasks to run inside.
- A task definition “helloworld-task” was created to define how the Docker container should run within the ECS service.
- The default security group was applied to the Application Load Balancer (ALB), while a separate security group, ecs-service-sg, was created to handle inbound traffic to the ECS service securely
- An ALB, named “helloworld-alb” was deployed to route HTTP requests from users to the containerized Spring Boot application running on ECS.

5.3.3 Phase 3: Deployment and Testing

HTTP Deployment

The main goal of phase 3 was to deploy the Spring Boot application and make it accessible over the Internet via AWS with Fargate launch type and an Application Load Balancer (ABL). This was

achieved by launching a service within the ECS cluster “helloworld” created in phase 2, attached to the ALB “helloworld-alb”, and verifying that the traffic was routed successfully to the correct running container.

To begin with creating a new service, first navigate to ECS console and select “Clusters” on the left-hand side menu. As can be observed in Figure 27, after double clicking the cluster “helloworld” that was created previously, the Cluster overview was displayed.

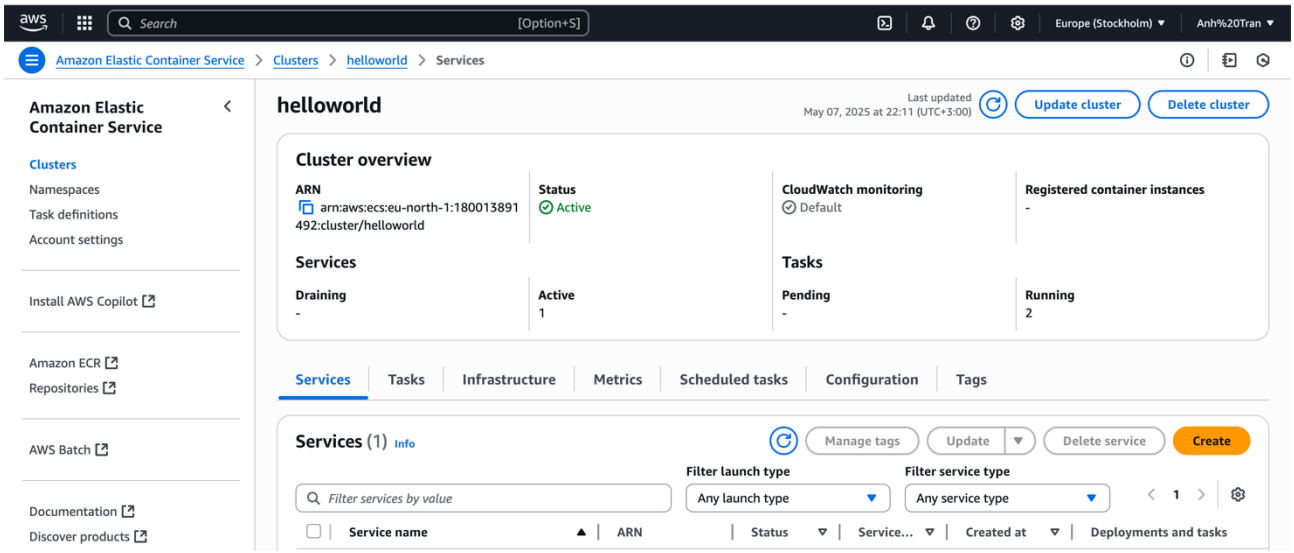


Figure 27. “helloworld” cluster

In the “helloworld” cluster dashboard, navigate to the tab “Services” and click the orange button “Create” to create a new service within this cluster. In the “Create service” form, these following fields should be set up as follows:

- Task definition family: select “helloworld-task” that was created in phase 2
- Service name: helloworld-task-service
- Cluster: helloworld
- Launch type: Fargate (to avoid the need for manual server provision and management)
- Load balancer type: Application Load Balancer
- Use an existing load balancer: enabled
- Load balancer: helloworld-alb
- Listener: HTTP:80
- Use an existing target group: enabled
- Target group name: helloworld-tg
- Health check path: /index
- Health check protocol: HTTP
- All other sections were left as default

After defining the task definition, launch type, and associating the service with pre-deployed ALB “helloworld-alb”, the button “Create” at the bottom of the form was clicked to initiate the deployment process. This triggered the ECS service to start a task defined in the form. The load balancer had a listener on port 80, which routed HTTP traffic to the associated target group helloworld-tg. The service was linked to this target group, allowing ECS to register the task as a target. This launching process could take up to a few minutes to complete. Once the ECS service was initiated successfully, it appeared in the “Services” tab. Double click the service name and check its status as “Active”, and deployment status as “Success” as illustrated in Figure 28.

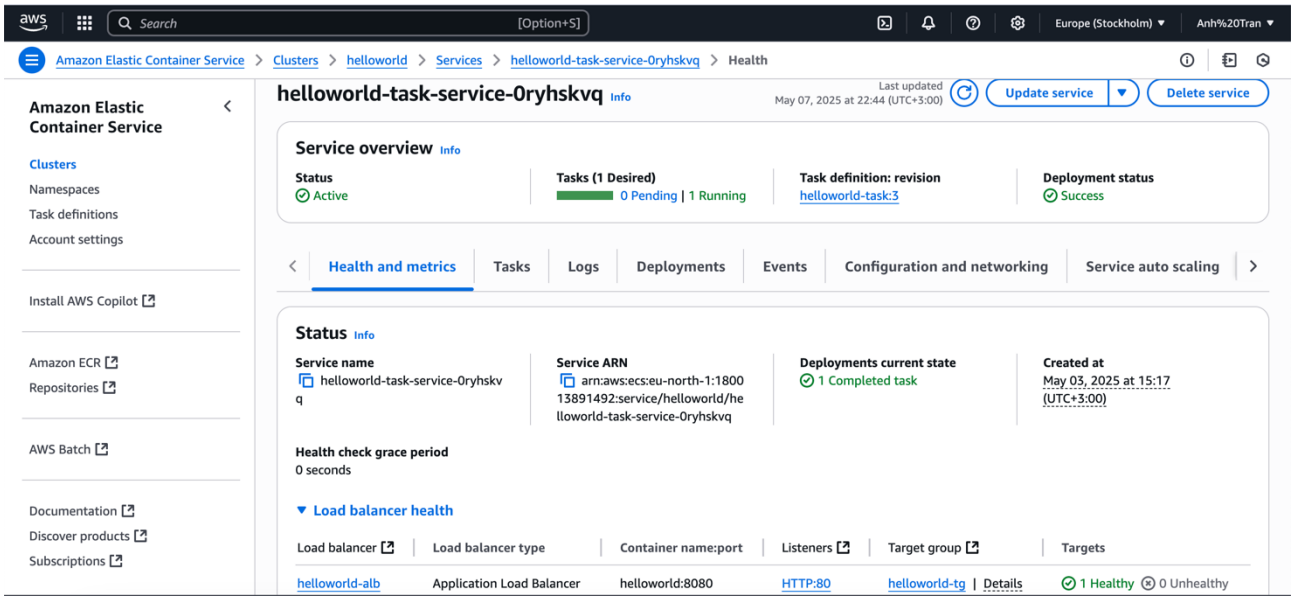


Figure 28. The service “helloworld-task-service”

As the ECS service was running in the private subnet with the listener on port 8080, it is not possible to access it directly. However, the traffic was routed through the Application Load Balancer “helloworld-alb”. This ALB was deployed in the public subnet and set up to direct HTTP requests to the private service. To test whether the containerized Spring Boot application was successfully deployed, the DNS name of the ALB could be used. This DNS name could be retrieved by navigating to the EC2 console and selecting “Load Balancer”. Copy the DNS name from the ALB “helloworld-alb” to start testing on a web browser.

Figure 29 showed that accessing the endpoint /index on the web browser using the ALB DNS “helloworld-alb-1667209789.eu-north-1.elb.amazonaws.com” returned a plain simple message “This is the main page” as expected. Similarly, the other two endpoints /contact and /hello also returned the correct outputs (see Appendix 1 for screenshots of all endpoint responses).

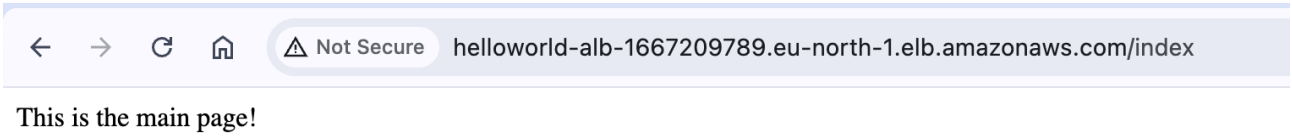


Figure 29. Browser output of the /index endpoint accessed via the ALB DNS

This confirmed that the containerized Spring Boot application was running successfully.

Adding HTTPS Support

Application deployment over HTTPS has been a great practice in web security and protecting sensitive information. By utilizing SSL/TLS certificates, HTTPS enables secure communication between applications and users. “An SSL (Secure Sockets Layer) certificate is a digital document that authenticates a website’s identity and enables an encrypted connection.” (SSL 2024). In Amazon Web Services, this HTTPS deployment can be accomplished by configuring the Application Load Balancer (ALB) to listen on port 443 (HTTPS) and associating it with a valid SSL/TLS certificate from Amazon Certificate Manager (ACM).

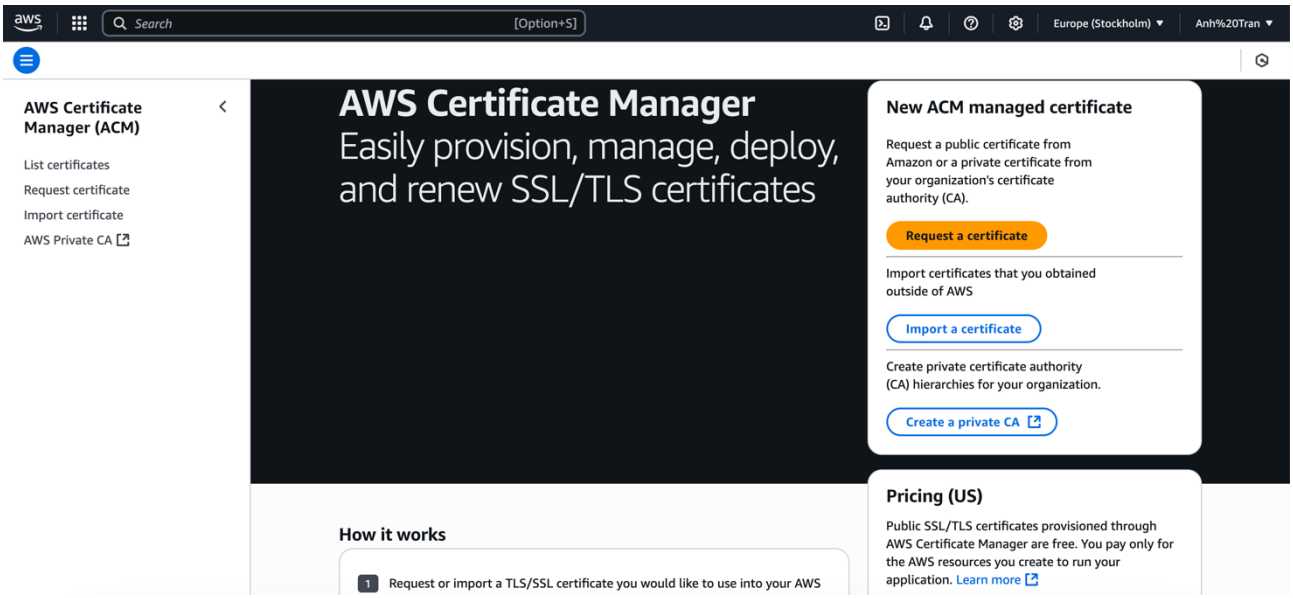


Figure 30. AWS Certificate Manager - Requesting a public certificate for HTTPS

After successful deployment over HTTP on AWS, adding HTTPS support typically involves the following steps:

1. Request a public SSL/TLS certificate from Amazon Certificate Manager (ACM) as shown in Figure 30. This step requires a custom domain such as site.example.com

2. Validate domain ownership with the DNS method. This validation requires adding a DNS record to the domain's DNS management system to verify the ownership.
3. The validation and issuing process is relatively quick, from 5 to 15 minutes. The third step is to adjust the ALB configuration by adding an HTTPS listener on port 443, associating the issue certificate with the listener, and sending the HTTPS requests to the same target group as the HTTP requests.
4. Redirect HTTP to HTTPS so that all traffic uses HTTPS for security purposes.

However, in this project, the HTTPS support was not implemented due to the lack of a custom domain. AWS does not offer SSL certificates for its automatically generated ALB DNS names (helloworld-alb-1667209789.eu-north-1.elb.amazonaws.com) as the domain is not controlled by the user and the user cannot prove his/her ownership. To continue with HTTPS, a domain name could be bought using a third-party registrar (like GoDaddy or Namecheap) or registered directly with AWS Route 53. The domain could then be set up to point to the ALB by a CNAME record, allowing for validation and certificate issuance with ACM.

While HTTPS support is beyond this project's scope, it is considered as a desirable enhancement for future production or work deployment use.

Alternative Approach: Using AWS CLI for Deployment

While this project utilized the AWS Management Console for a clear and better demonstration, all previous steps, such as creating a new ECR, pushing a Docker image to ECR, or creating a new cluster, could also be done by the AWS Command Line Interface. AWS CLI is beneficial in many ways, depending on users and purposes. For those more comfortable with command lines, performing tasks with AWS CLI speeds up the deployment process by avoiding manual clicks in the console. Additionally, writing scripts with CLI can help developers automate repetitive steps and minimize the risk of making errors.

5.3.4 Phase 4: CI/CD pipeline automation

Overview of CI/CD Pipeline

The project has gone through several steps, such as Dockerizing the application and setting up AWS configurations in phases 1, 2, and 3 to accomplish the main goal, which is to make the application accessible via ALB. However, if the project is developed further with more endpoints, will developers need to go through all those steps to re-deploy the application? The process is relatively troublesome and prone to human error. This is where CI/CD pipelines step in to solve the problem. As discussed in the theoretical framework section, CI/CD stands for Continuous Integration and

Continuous Delivery/Continuous Deployment. The ultimate objective of this phase is to automate the deployment process and minimize the manual work using GitHub Actions. Building an efficient CI/CD pipeline for this project comprises five main steps: linking the application to GitHub, creating the IAM User on AWS, generating and storing secrets on GitHub, adjusting the task-definition.json file, and lastly creating a GitHub Actions Workflow (deploy.yml).

Linking the Application to Github

To automate the deployment process with GitHub Actions, it was required that the Spring Boot application was pushed to a GitHub repository. Any changes made to the source code and pushed to GitHub served as a trigger to GitHub Actions workflows to rebuild and redeploy the application.

As the project was initially developed locally, a new repository named “helloworld-deployment-AWS” was created on GitHub. After creating the repository, the following Git command lines as shown in Figure 31 were executed to initialize the project folder, add the source files, and push the commit to the main branch.

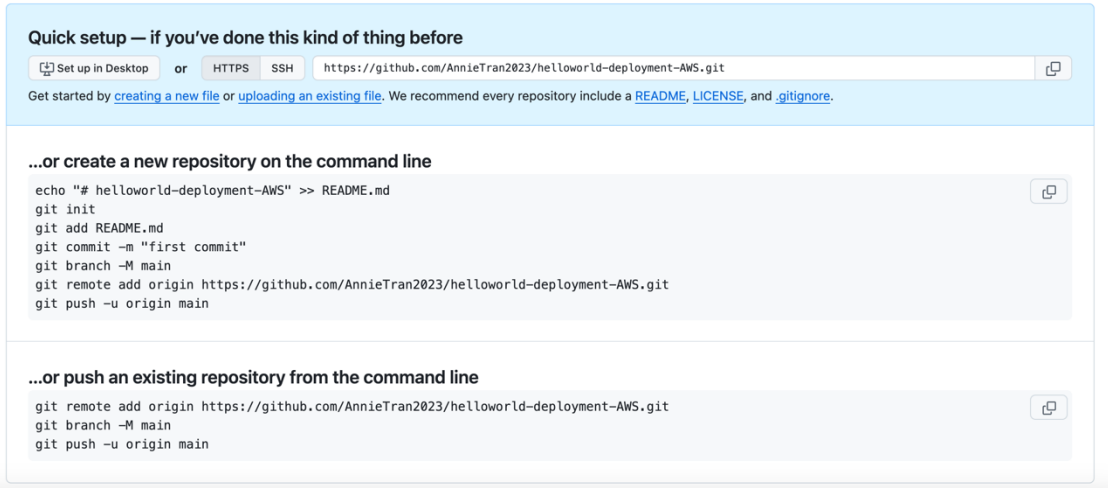


Figure 31. Git command lines for pushing the project to GitHub repository

This step formed a version control and set a solid foundation for CI/CD pipelines using Github Actions. Any future committed changes to this repository would trigger the workflow defined in the .github/workflows/deploy.yml file.

Creating the IAM User on AWS

Amazon Web Services (2025m) defined an IAM User as “an identity with long-term credentials that is used to interact with AWS in an account”. Creating the IAM User on AWS plays a key role in the GitHub Actions Workflow. GitHub Actions needs permission to interact with AWS to execute

certain tasks such as pushing Docker images to ECR, updating ECS services. The IAM User provides access keys that GitHub Actions can utilize securely without interfering with personal credentials.

To create a new IAM User, navigate to the IAM dashboard on AWS Console at https://us-east-1.console.aws.amazon.com/iamv2. Select “Users” in the menu bar on the left-hand side to navigate to the Users view, then click the “Create User” button on the right-hand side of the view. Figure 32 illustrated the form to create a new user with the Username as “github-actions-user”.

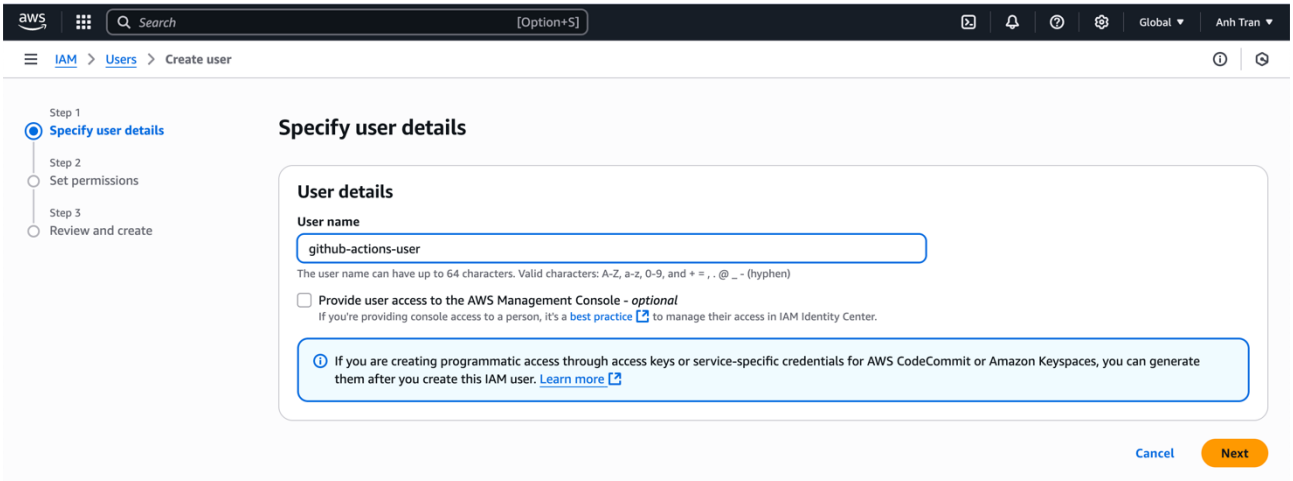


Figure 32. Creating a new IAM user for GitHub Actions in the AWS Console

In step 2 of setting permissions, select the third option, “Attach policies directly”. Granting permission for GitHub Actions Workflows to communicate with AWS requires the following policies:

- AmazonEC2ContainerRegistryFullAccess – to push Docker images to ECR
- AmazonECS_FullAccess – to update ECS services
- CloudWatchLogsFullAccess – to view ECS logs

Finally, step 3 involved reviewing the IAM User setup and creating the user. The IAM User named “github-actions-user” was then displayed in the Users dashboard. Select the user and then select “create access key” in the Summary section to generate a new secret access key.

Since GitHub Actions is a third-party service for CI/CD pipelines, the fourth option, “Third party service” should be selected, as demonstrated in Figure 33. In step 2, fill in the description tag value as “access-key-github-actions” and select “Create access key”.

Once the access key was successfully created, ensure that the access key and secret access key were stored safely for later use in GitHub.

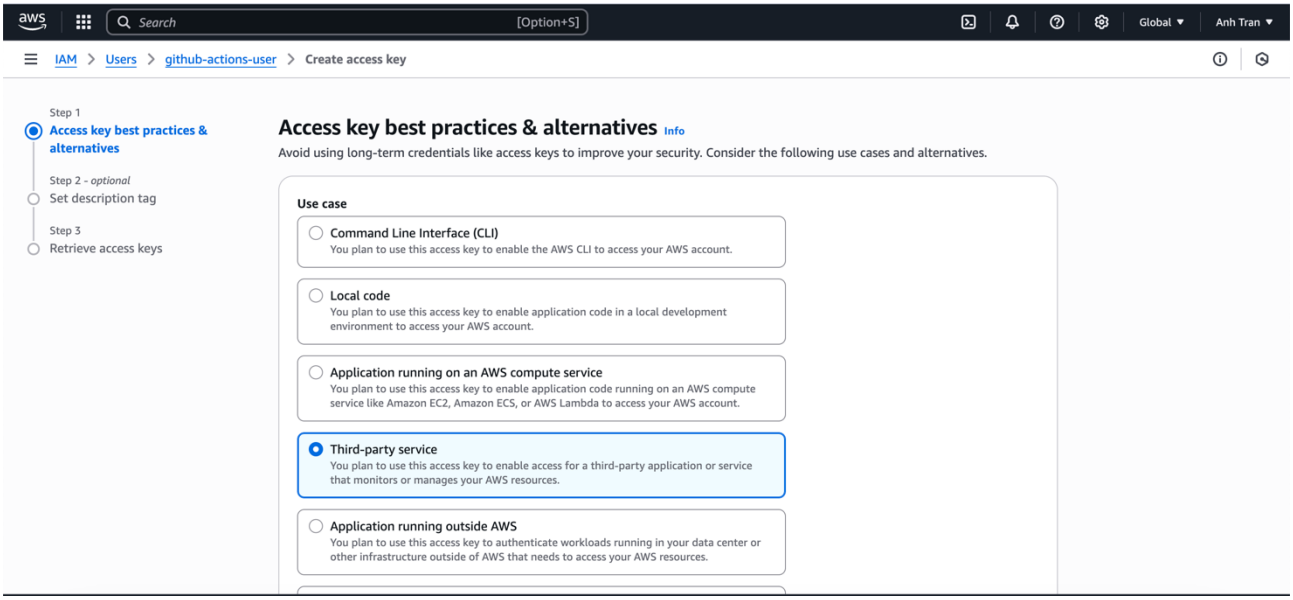


Figure 33. Selecting “Third-party service” when creating an access key for GitHub Actions

Storing Secrets in GitHub Repository

GitHub Actions needs to authenticate and interact securely with AWS to perform automated re-builds and redeployments. To achieve this securely, the access and secret keys must be encrypted and stored as secrets in the GitHub repository. This ensures sensitive information remains protected while allowing the GitHub Actions workflow to access the necessary credentials.

To store the secret, navigate to GitHub repository “helloworld-deployment-AWS” created in the first step and click on “Settings”. In the left-hand sidebar, select “Secrets and variables” under “Security”, then click “Actions”. Click "New repository secret" and enter each of the following secrets:

- AWS_ACCOUNT_ID : ****-****-**** (this could be found in the AWS account dashboard)
- AWS_ACCESS_KEY_ID: ********** (the access key generated earlier)
- AWS_SECRET_ACCESS_KEY: ********** (the secret access key generated earlier)
- AWS_REGION: eu-north-1 (Stockholm – closest region to Finland)
- ECR_REPOSITORY: helloworld
- ECS_CLUSTER_NAME: helloworld
- ECS_SERVICE_NAME: helloworld-task-service

Once these secrets were created, the repository secrets on GitHub should display them, as shown in Figure 34.

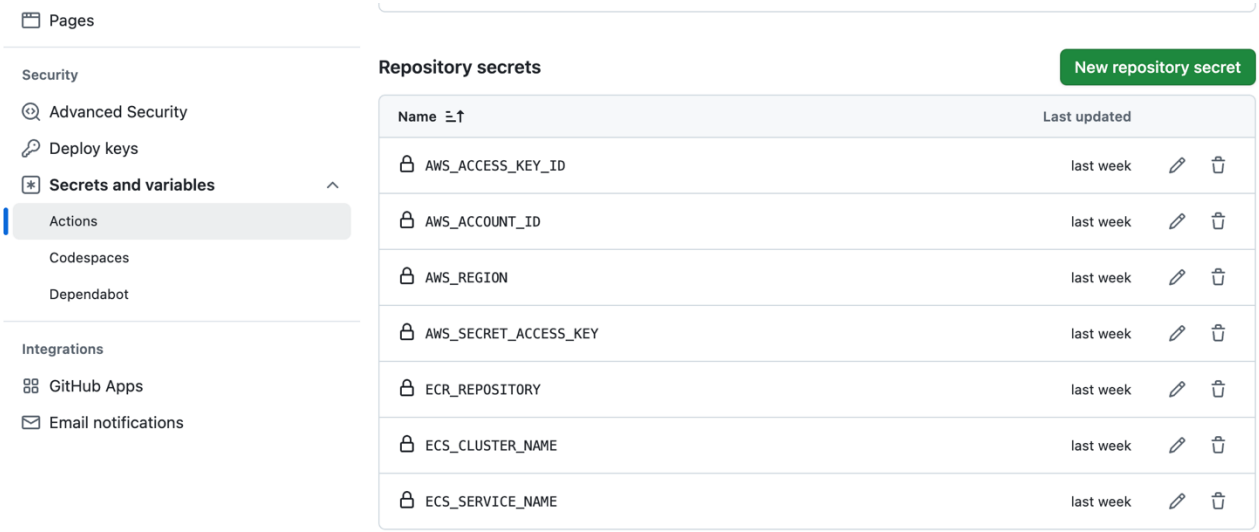


Figure 34. GitHub Repository Secrets Configuration for CI/CD Workflow

Adjusting task-definition.json

In this project, the task definition “helloworld-task” defines how the Docker container “helloworld” should run on the ECS cluster “helloworld”. This JSON-formatted file was required to integrate with GitHub Actions workflow to ensure that any updates to the application were deployed automatically and consistently. To get the JSON task-definition file, navigate to the ECS console, select “Task definitions” in the left-hand sidebar, and click “helloworld-task”. This step led to the view of “helloworld-task” with an overview of information and several tabs. Select the “JSON” tab, copy the full content to prepare for creating a task-definition.json file in the project folder (see Appendix 2 for the ECS Task Definition “helloworld-task” in JSON format).

Create the file task-definition.json under the root project folder and paste the code copied previously into this file. However, unnecessary metadata and system-generated fields must be removed to ensure the task-definition file integrates smoothly with GitHub Actions. The following fields were removed from the JSON file:

- "taskDefinitionArn": This information was automatically generated when registering a new task
- "revision": ECS automatically incremented the revision number when registering a new task
- "requiresAttributes": These are system-inferred capabilities of the task
- "registeredAt" and "registeredBy": Metadata to record by whom and when the task was registered
- "compatibilities": not necessary as "requiresCompatibilities" was already defined.

The clean-up version of task-definition.json can be found in Appendix 3.

Building GitHub Actions Workflow (deploy.yml)

The final step in the CI/CD workflow was to build a new GitHub Actions workflow file named deploy.yml. This YAML file determined the sequence of tasks that needed to be executed automatically when any updated codes were pushed to the repository. Those tasks included building the Docker image, pushing the image to ECR, and updating the ECS service.

To begin this step, create a directory .github/workflows in the root of the project. Within the folder workflows, a new filed named “deploy.yml” was added. The initial structure of this file was demonstrated in Figure 35.

```yaml
name: Deploy to AWS ECS

on:
  push:
    branches:
      - main
jobs:
```

Figure 35. Initial structure of the GitHub Actions workflow file (deploy.yml)

“Name” field defined the description of this workflow, and the “on” section determines that the pipeline was triggered whenever any changes were pushed to the main branch. The YAML file contains several main steps, including:

1. Check out the repository
2. Set up Java environment with JDK17
3. Pack the Spring Boot application into a JAR file using Maven
4. Configure AWS credentials using the access key, the secret access key, and the region stored in the GitHub Secrets
5. Build and tag the Docker image with the Amazon account ID, ECR name, and region from the GitHub Secrets
6. Push the Docker image to ECR with
7. Update the ECS service to deploy the new container version

The full deploy.yml was provided in the Appendix 4 for reference. Once the file was added and configured appropriately, the changes were pushed to GitHub repository to trigger the deployment process. This process could be observed for troubleshooting by navigating to the “Actions” tab inside the repository, as shown in Figure 36.

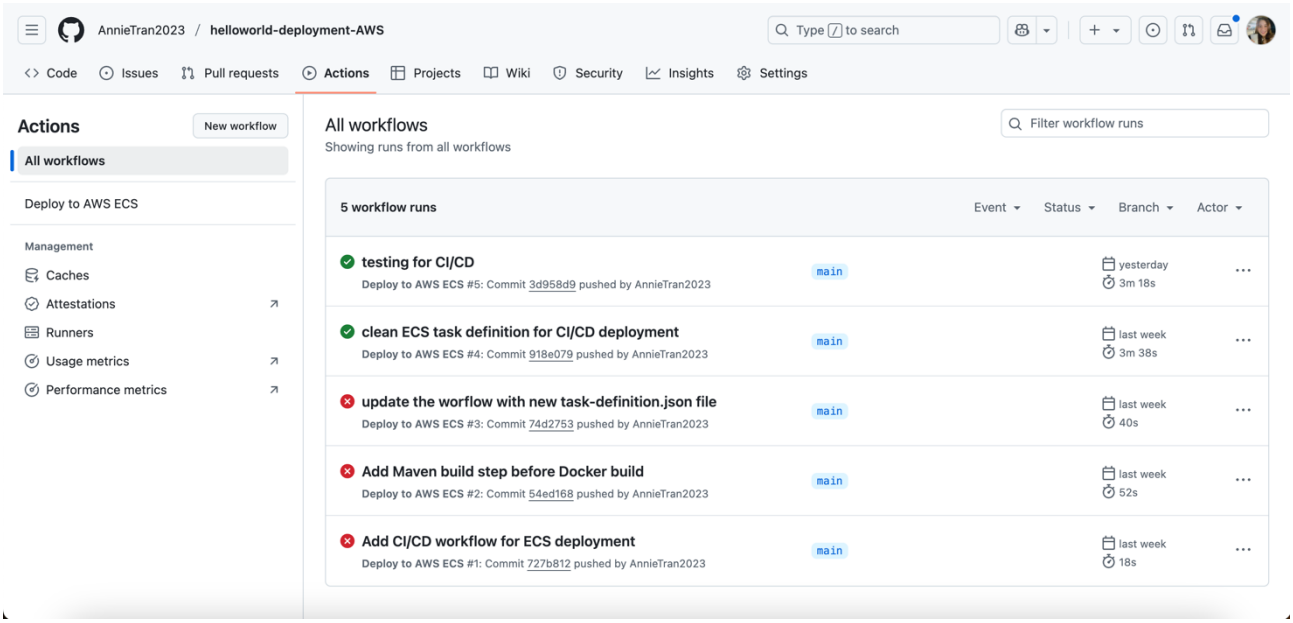


Figure 36. GitHub Actions tab showing the deployment workflow execution

Though the workflow ran successfully, as indicated by the green checkmark, the initial executions failed for two main reasons:

- Insufficient IAM permissions: As shown in Figure 37, the first build failed due to the lack of permissions from GitHub Actions to perform tasks such as pushing Docker images to ECR and updating ECS services. Therefore, it is crucial in the step “Creating IAM User” to assign proper policies, specifically “AmazonEC2ContainerRegistryFullAccess” and “AmazonECS_FullAccess”.

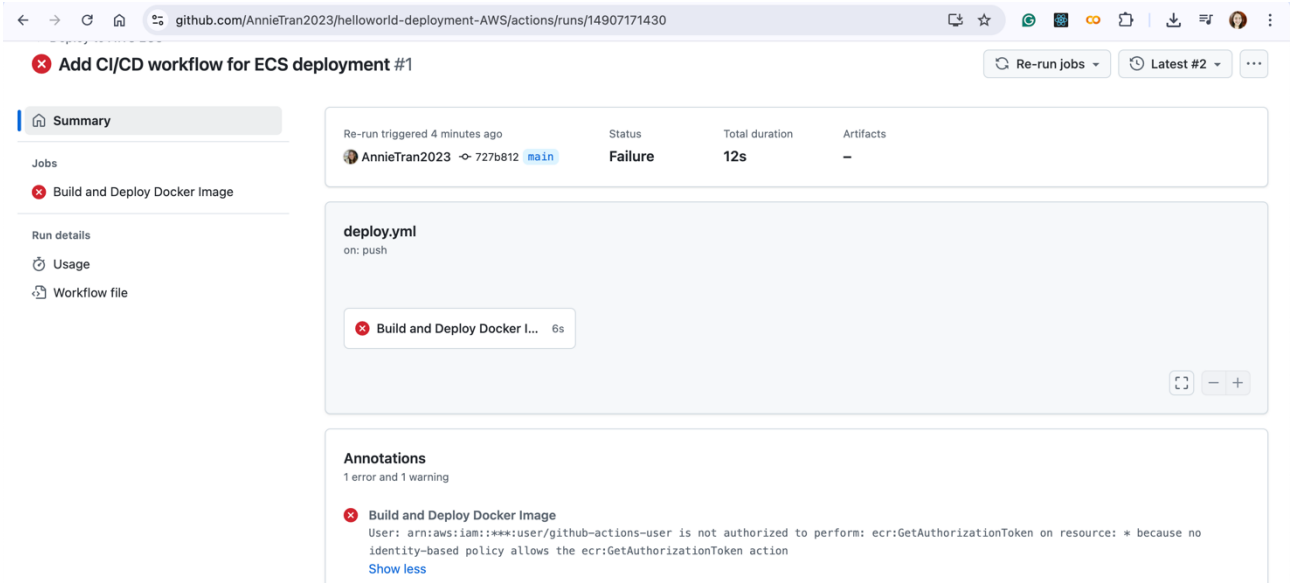


Figure 37. GitHub Actions workflow failed due to missing IAM permissions (ecr:GetAuthorizationToken)

- Inappropriate task definition file: The original task-defintion.json file exported from AWS contained metadata, system-related information, which resulted in the errors during the deployment process, as demonstrated in Figure 38. The workflow was executed appropriately after removing unnecessary fields in task-definition.json (taskDefinitionArn, revision, requiresAttributes, registeredAt, registeredBy, compatibilities).

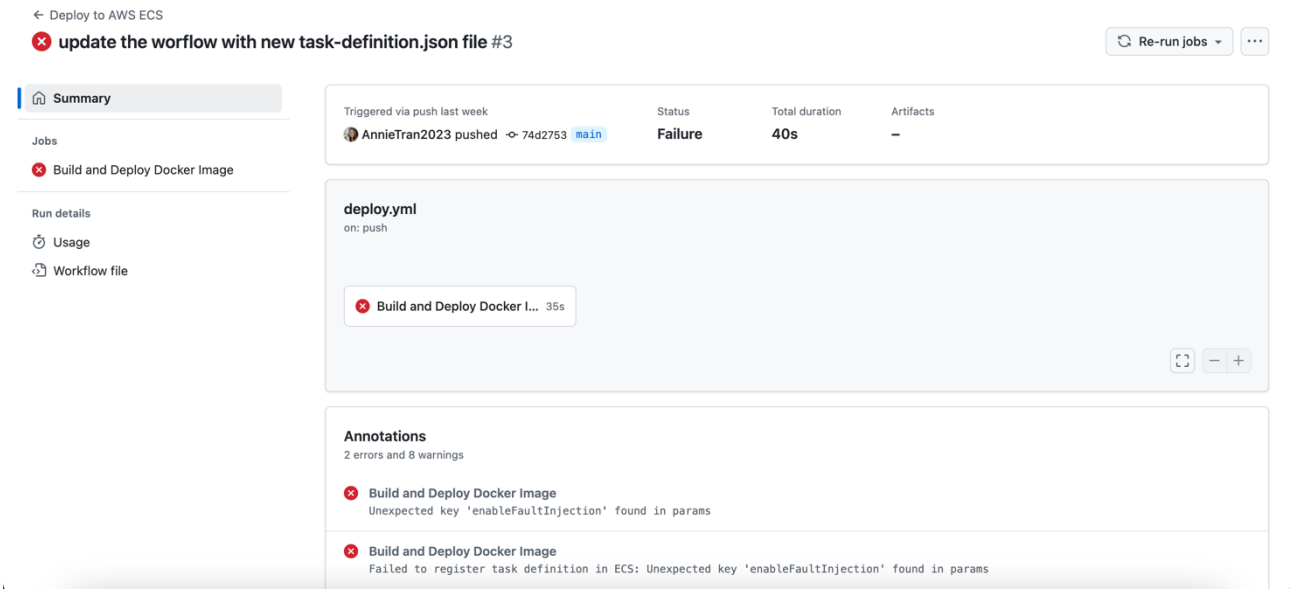


Figure 38. GitHub Actions workflow failed due to invalid fields in task-definition.json

In conclusion, the successful implementation of the CI/CD pipeline using GitHub Actions has fully automated the development to deployment workflow. In other words, any changes pushed to the GitHub repository triggered the rebuild and redeployment on AWS. As a result, developers can save a significant amount of time and minimize human errors.

5.4 Project Result

The first main goal of this project, deploying a Spring Boot application on AWS with Docker, was accomplished throughout the first three phases in the empirical part. In short, these steps included setting up local development, Dockerizing the Spring Boot application, configuring AWS services, and deploying the application. By the end of phase 3, the application was publicly accessible via an Application Load Balancer (ALB) at http://helloworld-alb-1667209789.eu-north-1.elb.amazonaws.com/index. Although formal testing was out of scope, the outcome displayed the result as expected, a plain text message “This is the main page!”. All other end points /contact and /hello also matched expectations (see Appendix 2).

The second primary goal was to automate the entire process from development to deployment. In phase 4, this was achieved by linking the project to a GitHub repository, creating an IAM User,

storing AWS credentials as GitHub repository secrets, and building the GitHub Actions workflow. This means that whenever there was any code update pushed to the GitHub repository, they served as a trigger to the pipeline, where it starts with building a Docker image, pushing the Docker image to ECR, and updating ECS services with the new image without any manual intervention. The ECS service “helloworld-task-service” consistently reached a steady state, as shown in Figure 39, which indicated that the container was redeployed and running successfully. Automating the rebuild and the redeployment not only eliminates a ton of manual work but also prevents any potential human errors.

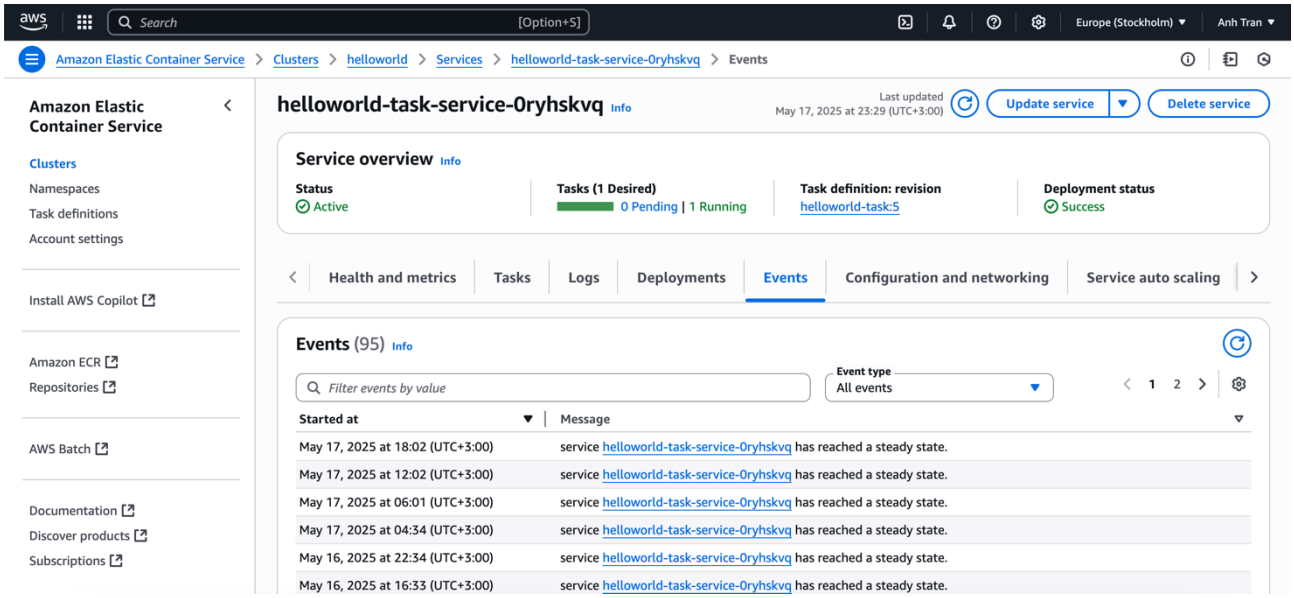


Figure 39. ECS Service reaching steady state after GitHub Actions triggered deployment

These outcomes confirmed that this project has effectively met both key objectives, deploying and automating the pre-built Spring Boot application on AWS with Docker and GitHub Actions.

6 Discussion

6.1 Evaluation of the Project

While AWS services and Docker are widely used in modern software development, there is yet a cohesive guideline for developers to deploy and automate Spring Boot applications using these technologies. As a result, the main objective of this project was to fill in this academic gap. This project aimed at deploying and automating a simple pre-built Spring Boot application on Amazon ECS using Docker and GitHub Actions. This was successfully achieved and validated with the application accessible via Application Load Balancer (ALB) and ECS service reaching steady state after each GitHub Actions trigger. This project may be valuable and informative for various audience groups, including cloud architects, devOps engineers, software developers, companies transitioning to a cloud-based microservices architecture, or simply individuals who are interested in learning new technologies such as AWS services, Docker, GitHub Actions.

While this project falls within AWS Free Tier, some awareness is still necessary regarding cost considerations and cloud billing. Services such as Elastic Container Registry (ECR), ECS with Fargate launch type, and the Application Load Balancer (ALB) may incur charges if usage is beyond the limit. Fargate charges for CPU and memory defined per task and for the time they are running actively. ALB has a fixed hourly rate. While there was no significant charge during the implementation of this project because of the short time and limited usage, it is crucial to consider cost monitoring and budgeting if the application is running for production.

One geographical challenge the project dealt with was Finland's lack of AWS data centers. Consequently, the closest available region for deployment was eu-north-1 (Stockholm). Though there was no significant delay when accessing the application, it is probably not the best option for companies with strict data residency or those desiring to have extremely low latency. This reason could also impact a company in Finland's decision to choose a cloud provider.

6.2 Limitations and Future Improvements

While the project successfully deployed the Spring Boot application and automated it with Amazon ECS, Docker, and GitHub Actions, it still imposed some limitations. First, the goal was to keep the Spring Boot application simple and minimal so that it could concentrate purely on the deployment and automation process. As a result, the guideline may not fully reflect the complications of real-world deployments since the chosen pre-built application did not implement any database integration or RESTful APIs.

Second, the project could not implement HTTPS support to secure the application due to a lack of a custom domain. This step is highly needed when the application is being used in production. Lastly, the project created only one CI/CD pipeline with GitHub Actions, which builds the Docker image, pushes it to ECR, and updates the ECS service. However, in reality, companies typically use more complex pipelines than that. They can include testing steps, rollback capabilities, and human approval. These advanced features were also out of the scope of this project.

All discussed limitations could also provide plenty of room for this project to grow and improve. This project can expand in the future by getting the SSL certificate from Amazon Certificate Manager (ACM) with a custom domain and implementing HTTPS support to secure the application efficiently. Moreover, the pre-built Spring Boot application can be more complex with database integration or external APIs. Finally, the CI/CD pipeline could be improved with additional features such as automated testing stages, deployment to multiple environments (e.g., staging and production), rollback mechanisms, and manual approval steps.

6.3 Personal and Professional Development

Upon completing the project, the author has gained valuable knowledge and hands-on experience with cloud services, AWS, Docker, and CI/CD implementation with GitHub Actions. Since these technologies are in high demand in the job market, they are crucial for growing a future career either as a software developer or a DevOps engineer. In addition, deploying and automating the Spring Boot application offers insights with real-life troubleshooting, such as incompatible architecture when using a Mac OS, permission issues, or pipeline errors. However, the author considers them not only obstacles during writing this paper but also significant advantages since all the issues encountered could be documented and help readers save time fixing or avoiding them.

Looking from a broader perspective, cloud service has always been a hot topic and a highly demanded skill for developers. Therefore, it is critical for the author to have a better understanding of the cloud deployment cycle. This paper could also benefit the author’s future career as it demonstrated knowledge of modern technologies and hands-on experience.

References

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. 2010. A view of cloud computing. Communications of the, 53, 4, pp. 50–58.

Amazon Web Services (AWS) 2025a. What is Continuous Integration?. URL: https://aws.amazon.com/devops/continuous-integration/. Accessed: 23 March 2025.

Amazon Web Services (AWS) 2025b. The Amazon EC2 instances for your Elastic Beanstalk environment. URL: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.ec2.html. Accessed: 1 April 2025.

Amazon Web Services (AWS) 2025c. What is AWS Elastic Beanstalk?. URL: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html. Accessed: 1 April 2025.

Amazon Web Services (AWS) 2025d. Elastic Load Balancing. URL: https://aws.amazon.com/elasticloadbalancing/. Accessed: 2 April 2025.

Amazon Web Services (AWS) 2025e. Amazon EC2 Auto Scaling. URL: https://aws.amazon.com/ec2/autoscaling/. Accessed: 2 April 2025.

Amazon Web Services (AWS) 2025f. Amazon Elastic Container Registry. URL: https://aws.amazon.com/ecr/. Accessed: 3 April 2025.

Amazon Web Services (AWS) 2025g. Amazon ECS clusters. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/clusters.html. Accessed: 5 April 2025.

Amazon Web Services (AWS) 2025h. Amazon ECS task definitions. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html. Accessed: 6 April 2025.

Amazon Web Services (AWS) 2025i. What is Amazon VPC?. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html. Accessed: 10 April 2025.

Amazon Web Services (AWS) 2025j. Control traffic to your AWS resources using security groups. URL: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html. Accessed: 10 April 2025.

Amazon Web Services (AWS) 2025k. Target groups for your Application Load Balancers. URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html . Accessed: 11 April 2025.

Amazon Web Services (AWS) 2025l. AWS Fargate Features. URL: https://aws.amazon.com/fargate/features/#topic-0 . Accessed: 1 May 2025.

Amazon Web Services (AWS) 2025m. IAM users. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html. Accessed: 11 May 2025.

Bari, A., Khan, I., Samrin, R., & Khare, A. 2024. VPC & Public Cloud Optimal Performance in Cloud Environment. Educational Administration: Theory and Practice, 30, 6, pp. 1789–1798.

Bashari Rad, B., Bhatti, H., & Ahmadi, M. 2017. An introduction to Docker and analysis of its performance. IJCSNS International Journal of Computer Science and Network Security, 17, 3, pp. 228-235.

Docker 2025a. Docker overview. URL: https://docs.docker.com/get-started/docker-overview/#what-can-i-use-docker-for. Accessed: 23 Feb. 2025.

Docker 2025b. Build, tag, and publish an image. URL: https://docs.docker.com/get-started/docker-concepts/building-images/build-tag-and-publish-an-image/. Accessed: 5 April 2025.

Dinu, F. & Ninawe, S. 23 April 2025. 20+ Best CI/CD Tools for DevOps in 2025. Spacelift. URL: https://spacelift.io/blog/ci-cd-tools#top-cicd-tools-and-platforms. Accessed: 30 March 2025.

Douglis, F., & Nieh, J. 2019. Microservices and containers. IEEE Internet Computing, 23, 6, pp. 5-6.

Fowler, M. 25 March 2014. Microservices. Martin Fowler Blog. URL: https://martinfowler.com/articles/microservices.html. Accessed: 24 February 2025.

Github 2025. Understanding GitHub Actions. URL: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions. Accessed: 30 March 2025.

Jani, Y. 2023. Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development. International Journal of Science and Research (IJSR), 12, 6, pp. 2984-2987.

Mell, P., & Grance, T. 2011. The NIST definition of cloud computing. National Institute of Standards and Technology. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. Accessed: 20 February 2025.

Mhatre, A.L. 2023. Microservices Architecture Using Docker and Kubernetes. International Journal for Multidisciplinary Research (IJFMR), 5, 5, pp. 1–4.

Microsoft Azure 2025. What is cloud computing? URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing. Accessed: 21 February 2025.

Microsoft Learn 2025. CI/CD baseline architecture with Azure Pipelines. URL: https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture?view=azure-devops Accessed: 1 April 2025.

Mistry, A. 2018. Expert AWS Development: Efficiently Develop, Deploy, and Manage Your Enterprise Apps on the Amazon Web Services Platform. 1st ed. Packt Publishing. E-book. Accessed: 30 April 2025.

Mudasir 6 Oct 2024. Amazon Elastic Container Registry (ECR): Simplifying Container Image Management. Medium. URL: https://medium.com/@mudasirhaji/amazon-elastic-container-registry-ecr-simplifying-container-image-management-64da472ffaff . Accessed: 3 April 2025.

Murugesan, S., & Bojanova, I. 2016. Encyclopedia of cloud computing. 1st ed. John Wiley & Sons, Incorporated. U.K. E-book. Accessed: 22 February 2025.

Newman, S. 2021. Building microservices: Designing fine-grained systems. 2nd ed. O'Reilly Media. California. E-book. Accessed: 24 February 2025.

Oracle 2025. JAR File Overview. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html. Accessed: 5 April 2025.

ProsperOps. July 2024. EC2 vs. Fargate: Understanding the Differences and Choosing the Best for ECS. URL: https://www.prosperops.com/blog/ec2-vs-fargate/ . Accessed: 1 May 2025.

Prakash, M. 2022. Build automation tools for software development: A comparative study between Maven, Gradle, and Bazel. CS & IT - CSCP 2022, 12, 6, pp. 73-89.

Reddy, N.A. & Naidu, W.T. 2024. Aws Load Balancing Used In Deployments. International Journal of Advances in Engineering and Management, 6, 4, pp. 147–159.

Saini, R. & Behl, R. 2020. An Introduction to AWS—EC2 (Elastic Compute Cloud). International Conference on Research in Management & Technovation, India, pp. 99–102.

Singh, N., Patel, D., Raj, A., Shubham, & Kour, S. 2023. CI/CD Pipeline for Web Applications. International Journal for Research in Applied Science & Engineering Technology (IJRASET), 11, V, pp. 5218-5226.

SSL 2024. What is an SSL/TLS Certificate?. URL: https://www.ssl.com/article/what-is-an-ssl-tls-certificate/. Accessed: 25 April 2025.

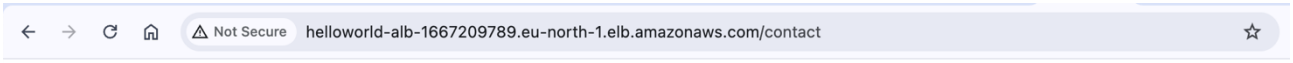
Spring Boot 2025. Microservices with Spring. URL: https://spring.io/microservices. Accessed: 25 February 2025.

Reddy, N.A., & Naidu, W.T. 2024. Aws Load Balancing Used In Deployments. International Journal of Advances in Engineering and Management (IJAEM), 6, 4, pp. 147–159.

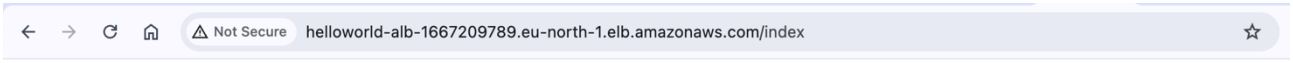
Yepuri, V., Polamarasetty, V. K., Donthi, S., & Gondi, A. K. R. April 2023. Containerization of a polyglot microservice application using Docker and Kubernetes. arXiv.org. URL: https://arxiv.org/abs/2305.00600. Accessed: 20 March 2025.

Appendices

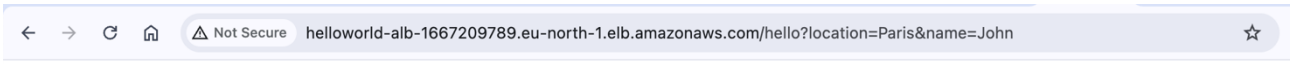
Appendix 1. Browser Output for Deployed Endpoints



This is the contact page



This is the main page!



Welcome to the Paris, John!

Appendix 2. ECS Task Definition “helloworld-task” (JSON Format)


```
{
  "taskDefinitionArn": "arn:aws:ecs:eu-north-1:180013891492:task-definition/helloworld-task:3",
  "containerDefinitions": [
    {
      "name": "helloworld",
      "image": "180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld",
      "cpu": 0,
      "portMappings": [
        {
          "name": "helloworld-8080-tcp",
          "containerPort": 8080,
          "hostPort": 8080,
          "protocol": "tcp",
          "appProtocol": "http"
        }
      ],
      "essential": true,
      "environment": [],
      "environmentFiles": [],
      "mountPoints": [],
      "volumesFrom": [],
      "ulimits": [],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/helloworld-task",
          "mode": "non-blocking",
          "awslogs-create-group": "true",
          "max-buffer-size": "25m",
          "awslogs-region": "eu-north-1",
          "awslogs-stream-prefix": "ecs"
        },
        "secretOptions": []
      },
      "systemControls": []
    }
  ],
  "family": "helloworld-task",
  "taskRoleArn": "arn:aws:iam::180013891492:role/ecsTaskExecutionRole",
  "executionRoleArn": "arn:aws:iam::180013891492:role/ecsTaskExecutionRole",
  "networkMode": "awsvpc",
  "revision": 3,
  "volumes": [],
  "status": "ACTIVE",
  "requiresAttributes": [
```

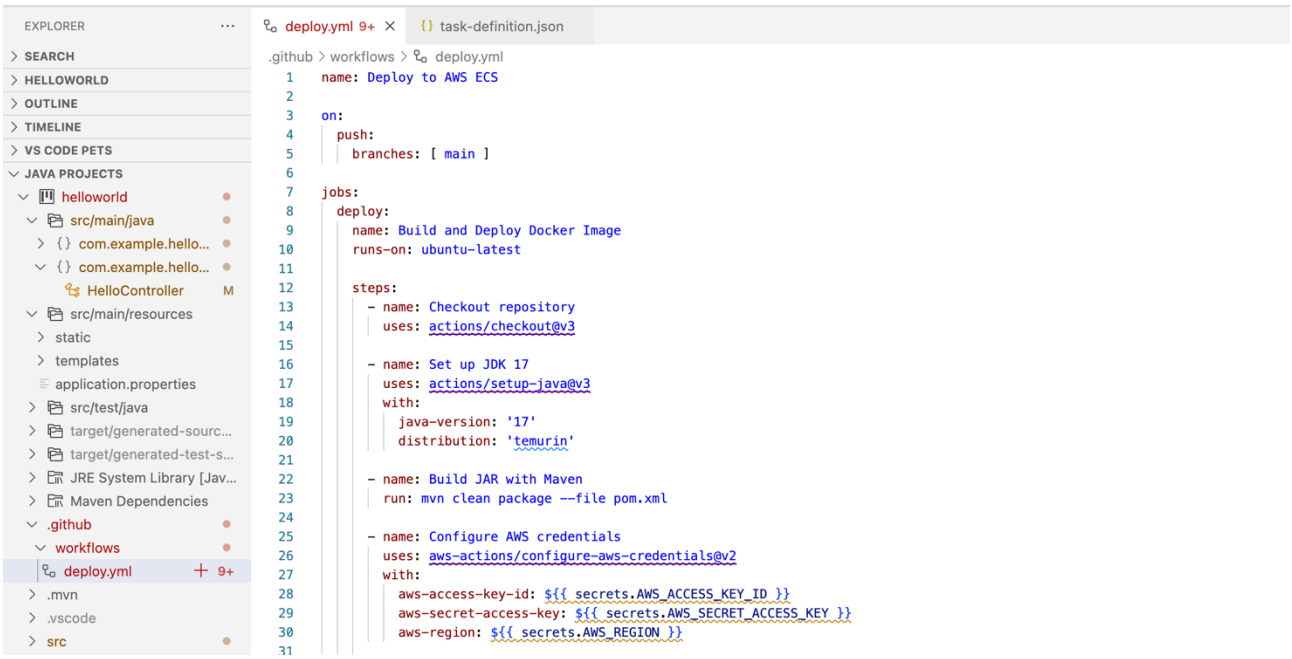
```
"requiresAttributes": [
  {
    "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
  },
  {
    "name": "ecs.capability.execution-role-awslogs"
  },
  {
    "name": "com.amazonaws.ecs.capability.ecr-auth"
  },
  {
    "name": "com.amazonaws.ecs.capability.docker-remote-api.1.19"
  },
  {
    "name": "com.amazonaws.ecs.capability.docker-remote-api.1.28"
  },
  {
    "name": "com.amazonaws.ecs.capability.task-iam-role"
  },
  {
    "name": "ecs.capability.execution-role-ecr-pull"
  },
  {
    "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
  },
  {
    "name": "ecs.capability.task-eni"
  },
  {
    "name": "com.amazonaws.ecs.capability.docker-remote-api.1.29"
  }
],
"placementConstraints": [],
"compatibilities": ["EC2", "FARGATE"],
"requiresCompatibilities": ["FARGATE"],
"cpu": "1024",
"memory": "3072",
"runtimePlatform": {
  "cpuArchitecture": "X86_64",
  "operatingSystemFamily": "LINUX"
},
"registeredAt": "2025-05-03T11:38:40.017Z",
"registeredBy": "arn:aws:iam::180013891492:root",
"enableFaultInjection": false,
"tags": []
```

Appendix 3. Adjusted task-definition.json for GitHub Actions Deployment

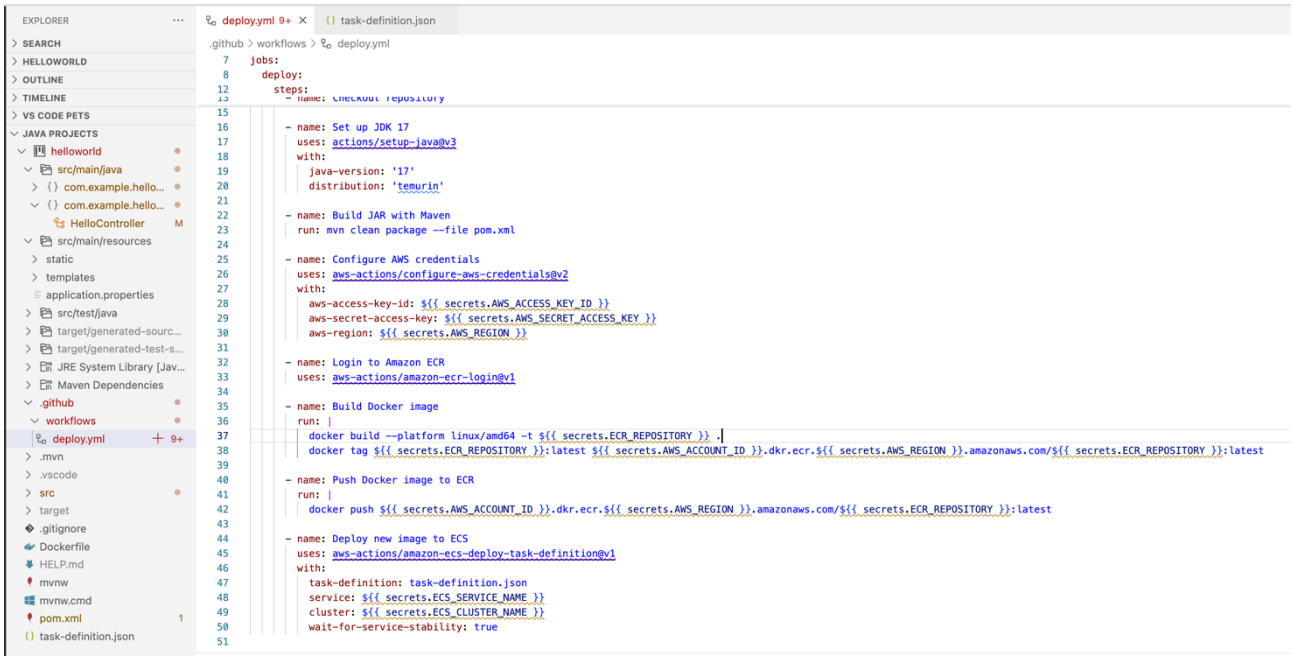
application.properties | deploy.yml 9+ | task-definition.json | Dockerfile | HelloController.java M | pom.xml 1

task-definition.json > ...

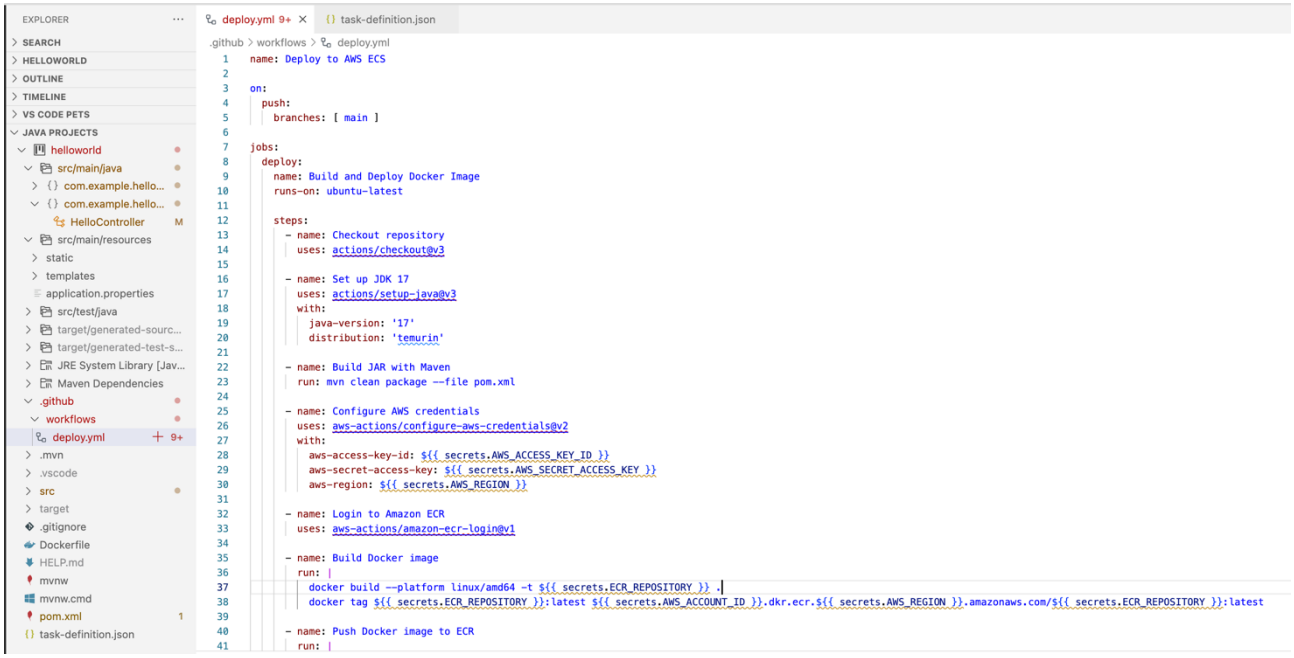
```
{
  "family": "helloworld-task",
  "containerDefinitions": [
    {
      "name": "helloworld",
      "image": "180013891492.dkr.ecr.eu-north-1.amazonaws.com/helloworld:latest",
      "cpu": 0,
      "portMappings": [
        {
          "name": "helloworld-8080-tcp",
          "containerPort": 8080,
          "hostPort": 8080,
          "protocol": "tcp",
          "appProtocol": "http"
        }
      ],
      "essential": true,
      "environment": [],
      "environmentFiles": [],
      "mountPoints": [],
      "volumesFrom": [],
      "ulimits": [],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/helloworld-task",
          "mode": "non-blocking",
          "awslogs-create-group": "true",
          "max-buffer-size": "25m",
          "awslogs-region": "eu-north-1",
          "awslogs-stream-prefix": "ecs"
        },
```



Appendix 4. GitHub Actions Workflow File: deploy.yml



```yaml
jobs:
  deploy:
    steps:
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Build JAR with Maven
        run: mvn clean package --file pom.xml

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build Docker image
        run: |
          docker build --platform linux/amd64 -t ${{ secrets.ECR_REPOSITORY }} .
          docker tag ${{ secrets.ECR_REPOSITORY }}:latest ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}:latest

      - name: Push Docker image to ECR
        run: |
          docker push ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}:latest

      - name: Deploy new image to ECS
        uses: aws-actions/amazon-ecs-deploy-task-definition@v1
        with:
          task-definition: task-definition.json
          service: ${{ secrets.ECS_SERVICE_NAME }}
          cluster: ${{ secrets.ECS_CLUSTER_NAME }}
          wait-for-service-stability: true
```



```yaml
name: Deploy to AWS ECS

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    name: Build and Deploy Docker Image
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Build JAR with Maven
        run: mvn clean package --file pom.xml

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build Docker image
        run: |
          docker build --platform linux/amd64 -t ${{ secrets.ECR_REPOSITORY }} .
          docker tag ${{ secrets.ECR_REPOSITORY }}:latest ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}:latest

      - name: Push Docker image to ECR
        run: |
```