Project Report

Project Group 12

Codebase: https://github.com/karanpathak/AWS-Serverless-vs-Serverful-Comparison

We chose the title “Comparing Serverless and Serverful Performance for An Application Using Microservices Architecture” for our project. We used Amazon Web Services as our cloud platform to deploy our application.

System Design

Application

We built and hosted a prediction microservice with one API endpoint (“*/predict*”) for an object-detection task. An image is uploaded in the S3 bucket (bucket name: “*distributedbucket*”), and the image filename is passed to the API endpoint in the request body while making the API call. The API endpoint downloads the respective image from an S3 bucket predicts the bounding box for all the respective objects in the image along with the confidence score, and uploads the resultant image to another S3 bucket (bucket name: “*distributedbucket-final*”). The endpoint returns the S3 public URL of the resultant image.

Components

We containerized the application and stored the container image on AWS Elastic Container Registry (ECR).

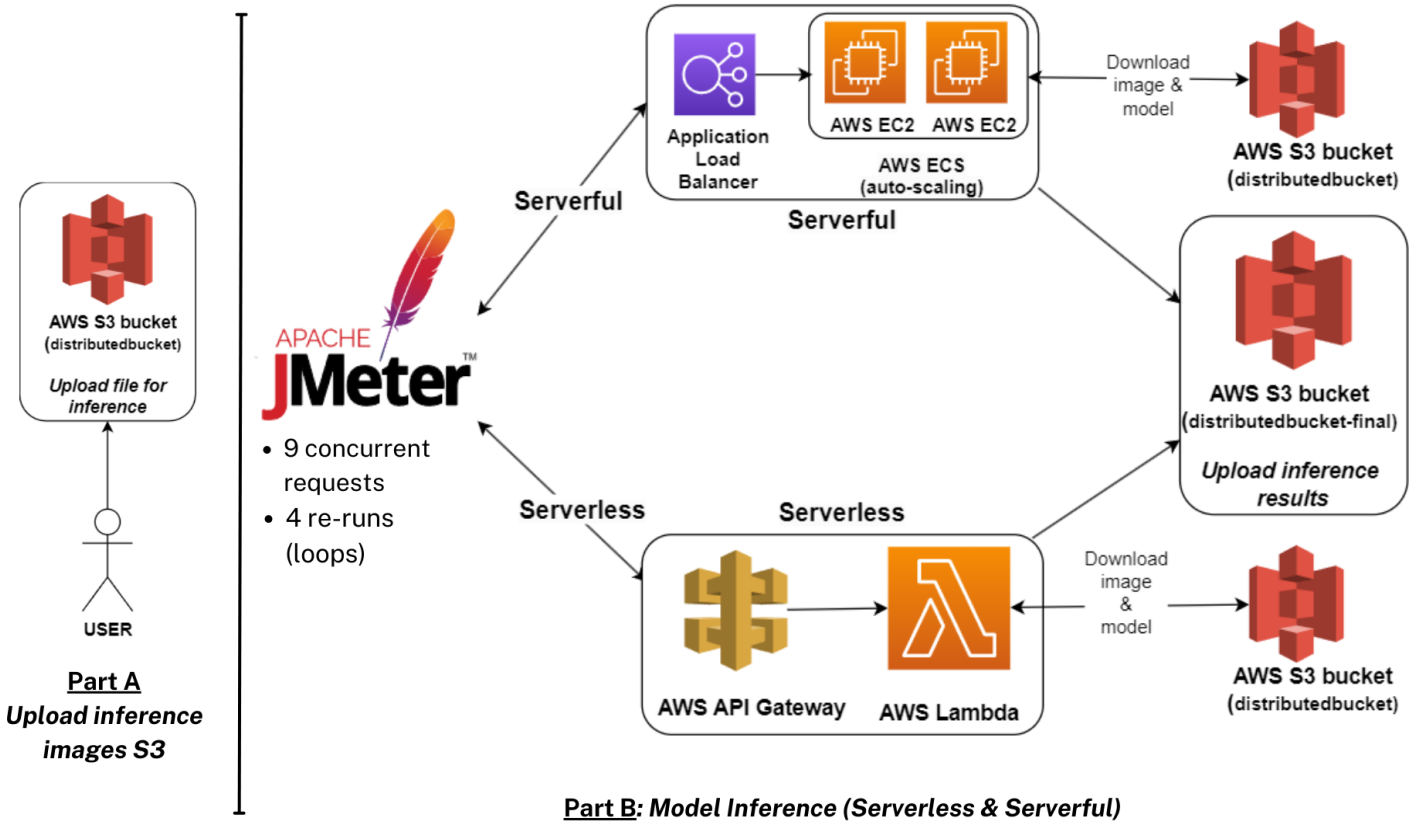

Figure 1: System Design

For serverful, we deployed the containerized application (stored in ECR) on a cluster of EC2 instances. We managed the cluster using AWS Elastic Container Service (ECS) and used auto-scaling for EC2 scale-in and scale-out. We connected an Application Load Balancer (ALB) to the cluster for providing access to the

application endpoint. For serverless, we deployed the containerized application (stored in ECR) onto AWS Lambda service (serverless computing) and connected an AWS API Gateway with the Lambda service for providing access to the application endpoint. We created two S3 buckets - "*distributedbucket*" and "*distributedbucket-final*". "*Distributedbucket"* was used for storing model weight files and input images, and "*distributedbucket-final"* was used to store the output images. Both EC2 instances in the cluster and Lambda functions had access to the respective S3 buckets.

Implementation

Application

We used the official COCO dataset pre-trained PyTorch-based YOLO V7 (YOLOv7-W6 model version) object-detection model. We developed a microservice with the *"/predict"* endpoint in the BentoML framework. The microservice downloads the input image (given in the request body) and model from "*distributedbucket"* S3 bucket, runs the YOLO V7 model, uploads the output to the "*distributedbucket-final"* S3 bucket, and returns the public URL of the output image (stored in the S3 bucket) in the response body. We containerized the application using the Docker tool and stored the containerized application in AWS ECR.

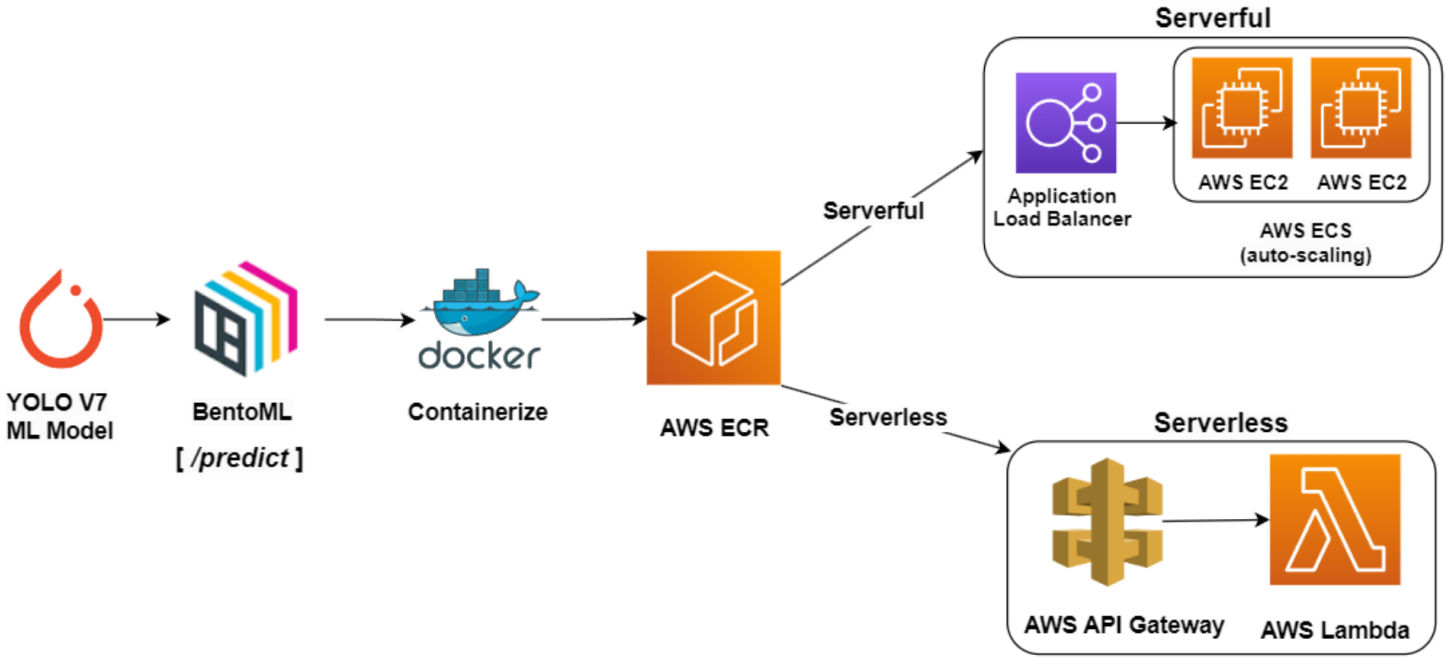

Figure 2: Microservice Implementation

Serverful Deployment

We used the AWS UI to deploy our containerized application on a cluster of EC2 instances and the cluster was managed using ECS. The configured auto-scaling in ECS with a minimum of 1 and a maximum of five EC2 instances. We set the scale-out criteria "*CpuPercentageUsage"* to 70%, and for scale-in, we used idleSystemTime of an EC2 instance i.e. delete an EC2 instance if idle for more than 1 minute. We configured and connected an Application Load Balancer (ALB) to the cluster for providing access to the application endpoint. We used the t3a.xlarge general-purpose EC2 instance type for our experiments.

Serverless Deployment

We used Bentoctl (a BentoML command-line tool) to deploy the containerized application onto AWS Lambda and connected an AWS API Gateway with the Lambda service for providing access to the application endpoint. We set the Lambda function memory to 3000 MB and the timeout to 30 secs.

Experiment & Analysis

We used response time as our comparison metric and used the JMeter tool to simulate load testing.

Testing Methodology

For our experiment, we uploaded 20 random images from the COCO dataset to "*distributedbucket*" S3 bucket for testing purposes (*Part A, Figure 1*). We spawned 9 concurrent requests in one batch of requests and re-ran the batch 4 times for both serverless and serverful.

Results & Analysis

For serverful (the EC2 cluster), we obtained the optimal results with 2 t3a.xlarge EC2 instances. We started with 1 default EC2 instance which auto-scaled itself to add one more EC2 instance of the respective type to handle the application load.

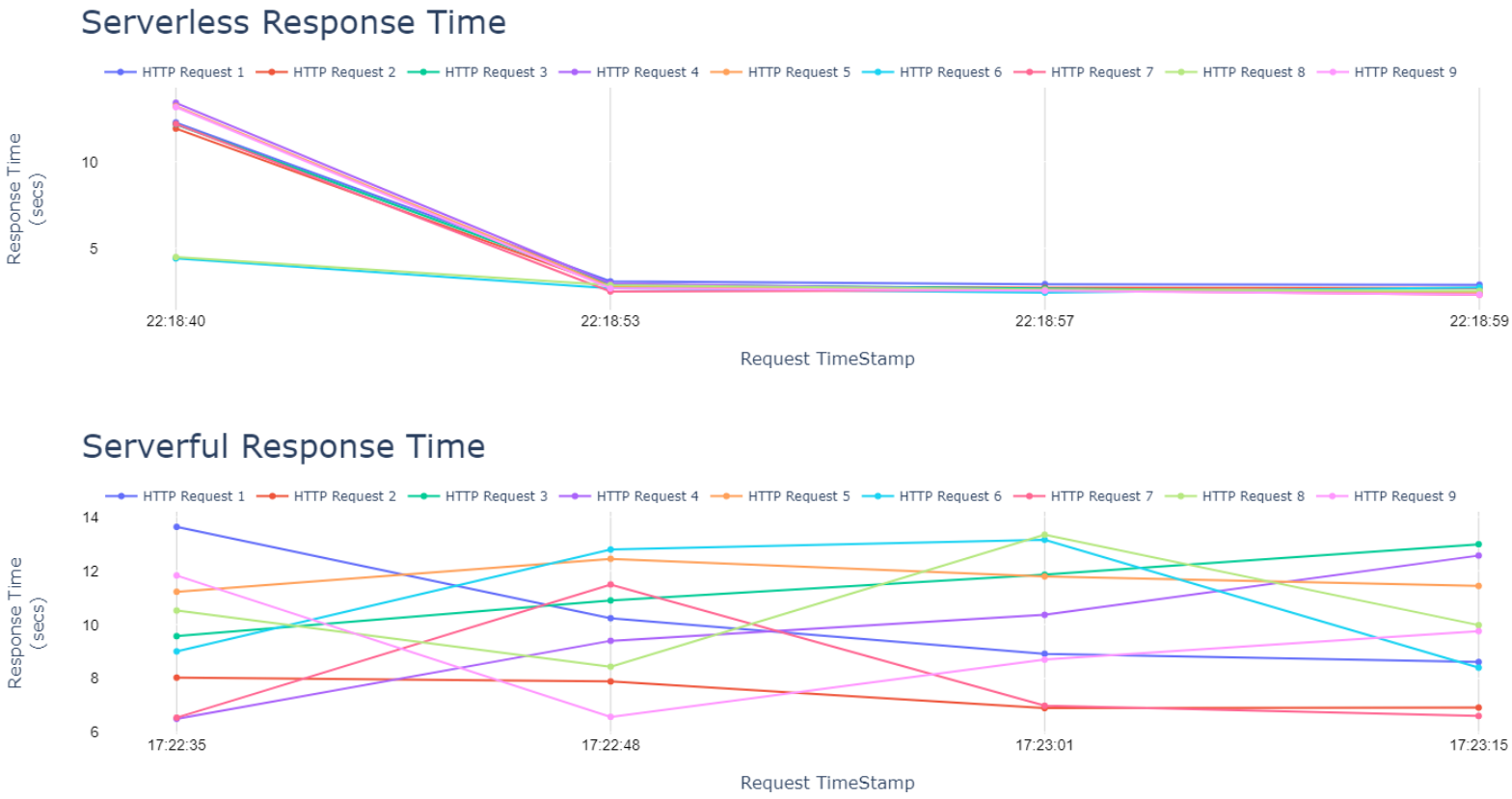

Figure 3: Response Time Comparison (Serverless vs Serverful)

We chose an ML application so that we could evidently showcase a significant difference between the serverless and serverful response times. For the serverless, we saw the problem of cold-starting as the first batch of requests (timestamp: *22:18:40*) had an average 11 sec response time but the subsequent requests had less than 3 secs response time. Moreover, AWS Lambda uses a default in-built caching mechanism which is not present in the EC2 cluster which could also lead to lower response times in Lambda functions as compared with EC2 cluster. For the serverful, we saw that all the requests across various batch runs had a consistent response time i.e. response time was between 6 to 14 secs. The reason behind such a response time range for serverful is that current requests have to wait for existing requests to be completed.