Performance of Microservices when deployed in AWS EC2 versus ECS with Fargate

Elijah Nnorom
York University, Canada
ennorom@yorku.ca

***Abstract*—The deployment of Microservices in Docker containers is a popular pattern for cloud native applications. Amazon Web Services (AWS), as the largest cloud service provider, fully supports deployments in containers. It offers a repository for storing the Docker image, various options for hosting container applications, monitoring tools, cost management tools, and other services. The number of services has made it difficult to make informed decisions when deploying and maintaining software.**

To bridge this gap, we perform an empirical study on a Spring boot application, a common Java framework for building microservices. Specifically, we will compare the performance of this application when deployed in containers on Elastic Container Service (ECS) using Fargate Launch type, a managed service, and deployed in containers on Amazon Elastic Compute Cloud (EC2) a staple virtual machine offered by AWS. It was found that AWS Fargate deployment was more cost-efficient and had better latency, Memory, and CPU utilization. Machine-learning models were built to model and predict the latency that users experience, with good performance that has potential for utilization in predictive scaling.

***Index Terms*—Amazon Web Services, EC2, Elastic Container Service, Microservices, Spring Boot**

I. Introduction

Microservice architecture has been a widely accepted model for building cloud-native applications [1] [2]. It offers advantages such as better cost saving, more fault isolation, improved flexibility, and interchangeability [1]. The loosely coupled design allows for independent scaling. This benefit is enhanced when deployed in cloud environments and combined with containerization technologies. There is a growing number of tools for support containers to aid code version control, streamline deployment, ease scaling, and facilitate monitoring [4] [5]. The overwhelming support and overlapping functionalities of tools offered by Cloud platforms have made the deployment decision-making process complex. Decision-makers must balance performance, cost, and support to meet their applications [7].

Spring Boot is a popular, well-supported Java framework that provides robust configuration options that make it suitable for building microservices. Its widespread adoption for building UI and APIs in enterprise environments makes it ideal for evaluating realistic deployment strategies in cloud platforms like AWS [7].

Among the cloud providers, AWS stands out as the leading cloud platform, offering a variety of deployment models for containerized applications. The models can be categorized into AWS-managed and self-managed options. In the AWS-managed option, AWS is responsible for the operations and infrastructure deployment [8] [9]. AWS tools like ECS (AWS Fargate) fall into this category [10]. In contrast, the Self-managed option requires the cloud customer to handle the selection, operation, and management of the cloud deployment infrastructure in use. By clearly understanding these options, decision-makers can make informed decisions that align with desired performance engineering and strategic goals.

Deployment planning from a performance perspective considers the user traffic load patterns and their relationship to the latency (response time), resource consumption, and deployment infrastructure. To understand this relationship, performance modelling is a crucial component of the decision-making process. Simulation models are proven effective performance models that provide options like specialized modelling tools such as Palladio [11] or Machine learning [12].

This paper contains an empirical study of a Spring Boot microservice deployed in a Docker container on AWS. The same microservice will be deployed on EC2 with a self-managed deployment type and on ECS (AWS Fargate) with an AWS-managed deployment type. Monitoring data will be collected and used to train a machine learning-based performance model to simulate the latency experienced by users in each deployment setup. The primary objective of the study is to evaluate and model the performance of a Spring Boot microservice (PetClinic) across different AWS deployment models which are self-managed (EC2) and AWS-managed (Fargate). The Focus is on understanding the latency, cost and failure tolerance. The goal is to provide insights that can enable decision-making when choosing deployment strategies for cloud-native applications. The specific objectives of the study are to:

- Explore the performance of a Spring Boot microservice when deployed in containers on EC2 versus ECS (AWS Fargate).
- Highlight the cost of using a Spring-boot application deployed in EC2 versus ECS (AWS Fargate).
- Investigate failure handling when a Spring-boot application is deployed in EC2 versus ECS (AWS Fargate).
- Examine the prospects of enhancing the latency experienced by users when using Spring-boot applications deployed in EC2 versus ECS (AWS Fargate).

The remainder of this paper is structured as follows. Section 2 introduces the related works of the study. Section 3 outlines

the approach of the experiment. Section 4 presents the result of the experiment. Section 5 discusses the results of the experiment. Finally, Section 6 contains the closing remarks.

II. Related Work

Ahmed and Islam [13] performed a study comparing the performance of an application when deployed on a Docker container in ECS (with an EC2 launch type) versus directly on an EC2 virtual machine. The authors considered the HTTP response, error percentage and throughput of different user traffic volumes. Based on these considerations, the authors found that deployment on ECS resulted in better performance than deployment on EC2. In contrast, this paper compares a container-based deployment on EC2 versus ECS with an AWS Fargate Launch type. Furthermore, the author's deployment in ECS used the underlying EC2 launch type instead of the AWS-managed Fargate Launch type explored in this paper. Notable, this paper utilized the data obtained from the experiment to train a machine-learning model to simulate users' response times.

Bagai [14] performed a study comparing three deployment models AWS ECS, SageMaker, and Lambda. The features considered include functionality, integration, scalability, performance metrics, cost implications, and overall strengths and weaknesses during their comparison. The authors found that AWS ECS provided cost benefits for sustained and predictable patterns. ECS also provided performance benefits for computational tasks due to the ample setup options, allowing control over configurations to optimize for computation. The author's comparison is based on AWS specifications, documentation, and case-study-driven performance profile rather than performing an experiment, which this paper pursues.

III. Approach

In this study, a Spring Boot application is deployed using Docker containers under two different setups on AWS: directly on EC2 instances (self-managed) and on ECS with the Fargate launch type (AWS-managed). The application under evaluation, Pet Clinic, consists of eight independent microservices—hereafter referred to as services. The modular nature of the services allowed for their deployment in containers across both deployment setups.

The dual deployment strategy enables comparative analysis across several dimensions, including infrastructure setup, deployment complexity, application performance, and cost implications. A workload generator simulates user traffic to the application, during which these key metrics are collected, aided by AWS Cloudwatch: response time, CPU utilization, and memory consumption are collected. These metrics are processed, analyzed and used to infer the performance and cost of each deployment setup.

Data under varying user loads was used to build machine learning (ML) regression models. These regression models were used to model the application's response time based on the user load. Regression algorithm selection, feature engineering, and hyperparameter tuning were part of the modelling process. The ML models' R-squared (R^2) and Root Mean Squared Error (RMSE) were used to evaluate their performance.

All resources used in this experiment were in the AWS us-east-2 region, except global non-region specific resources. Since the region of deployment correlates with the network latency experienced by users, choosing to deploy all resources in the same region enables a comparison of both deployment setups under similar network constraints associated with that region. However, this does not eliminate network latency differences between the two deployment setups, even though network data was not taken into account in this study.

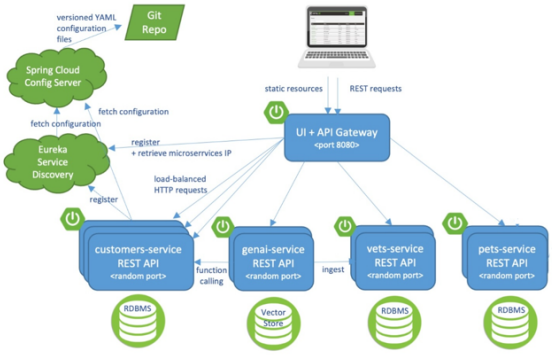

Fig. 1. Architecture overview of the PetClinic application with core services highlighted.

A. Subject System: Spring Boot Microservices Architecture

Figure 2 presents the architecture of the application under evaluation [15]. It is a microservice called Petclinic built with Spring Boot; it consists of eight core services: config-server, discovery-server, api-gateway, customers-service, genai-service, visits-service, vets-service, and admin-server. All services rely on two foundational components: the config-server, which acts as a centralized configuration manager, and the discovery-server, which provides service registration and discovery capabilities through Eureka.

Api-gateway is the entry point to the application, which routes incoming requests to the appropriate downstream secondary service. The include the customers-service, vets-service, and genai-service. The customers-service manages customer and per owner information. The vet-service maintains data on Veterinarians and their specialties. The visit-service tracks and schedules pet visits. The genai-service offers a generative AI chatbot for interfacing with users. The genai-service is dependent on a downstream call to Openai, which is used in the backend of the chatbot. The application utilizes an HSQLDB database that is prepopulated with data upon startup of each service, which was sufficient for our experiment [16].

The end-to-end latency experienced by a given service user was impacted by interservice dependencies, where all other services relied on the config-server and discovery-server. Additionally, the genai-service had both internal and external

dependencies on Openai. In broader terms, the architecture aligns with modern software development practices, making this application ideal for this study.

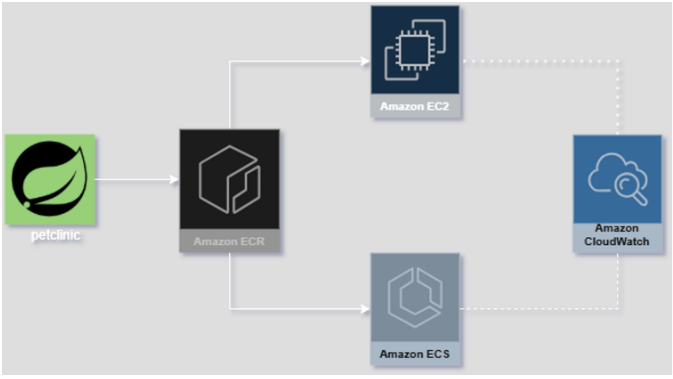

Fig. 2. Architecture overview of the deployment setup in Amazon Web Services, including CloudWatch.

B. AWS Deployment Configurations

Figure 2 presents the platforms used, which were AWS EC2 and Fargate. EC2 instances are virtual machines where the AWS administrator is responsible for managing the resources including choosing the instance type, setting up scaling, software installation and updates. Fargate is serverless offering where the underlying infrastructure is managed by AWS including provisioning, EC2 instance selection and scaling.

1) ECR Image: A private repository was created on the Elastic Container Registry (ECR) in AWS, which simplified the maintenance of the container image versions [17]. Subsequently, the container images for each service were built and pushed. This ensured that the same image was deployed in the two different deployment setups, where both benefited from the optimal configuration of pulling the images from a private repository on AWS rather than an external repository.

2) EC2 Deployment: The EC2 instance type and image were chosen based on research that considered the CPU and memory needs of the application [18]. The Docker containers were deployed on a T3.medium instance with 2 vCPU; each deployed container had a memory allocation of 512 MiB. An IAM role was attached during setup to enable EC2 access to the ECR repository to retrieve the container image. Then, the instance's security group was configured to allow inbound traffic. After the instance was provisioned, Docker was manually installed. Finally, Docker containers were launched on the EC2 instance using the Docker compose file referencing the latest image in ECR.

For monitoring, CPU and memory utilization metrics were forwarded to CloudWatch [19], while the docker stats command was used to obtain container-level insights. Since the EC2 instance (T3.medium) provides 2 vCPU shared across all containers, and ECS allocates a fixed 0.25 vCPU per container, normalization was necessary for a direct comparison. To estimate CPU utilization under 0.25 vCPU allocation, the EC2 container-level CPU utilization was adjusted. The resulting normalized value reflects how each EC2 container would perform if constrained to the ECS vCPU limits. This process involves two steps where TOTAL_VCPU is 2:

$$\text{used\_ec2\_vcpu} = \left(\frac{\text{ec2\_percent}}{100}\right) \times \text{TOTAL\_VCPU} \quad (1)$$

$$\text{normalized\_ec2\_vcpu} = \left(\frac{\text{used\_ec2\_vcpu}}{0.25}\right) \times 100 \quad (2)$$

Normalization was not required for memory allocation, as 512 MiB was directly allocated to each container in the docker Compose file. The memory utilization from the docker stats command was used for comparison.

3) ECS Deployment using Fargate launch type: A cluster was created to manage the services for the ECS (Fargate launch type) deployment [21] [21]. The required network configuration included the selection of the VPC and subnet, followed by creating a security group that allowed traffic into the application. A Namespace was created in AWS Cloud Map to allow interservice communication [22]. Each service had a DNS name that resolved to the internal IP, which enabled routing by the api-gateway via Eureka. For IAM setup, a role was created to allow the ECS agent to perform the setup of each service, and a second role enabled the service to interact with other AWS tools, such as CloudWatch. Finally, an aws-log group was created to allow log aggregation.

The docker compose file was converted to task definition files for each service that specified the resource allocation, container image to deploy from ECR, launch type (Fargate), environment variables, log configurations and IAM configurations. Finally, the task definition file was used to deploy the container service in the cluster with the specified network configuration.

For monitoring, the application logs were forwarded to the aws-log tag created for each service in CloudWatch. Moreover, the CPU and memory utilization were captured based on the service's usage; it was eventually be retrieved for the analysis phase of our experiment.

C. Workload Simulation and Performance Metrics

In this study, the workload simulation was designed to evaluate the performance of two distinct deployment setups in response to user traffic. The usage scenarios were meticulously planned, making the use of synthetic benchmarking suitable for this analysis. A fixed work benchmarking strategy was employed to highlight the performance difference between setups. During the traffic generation phase, multi-threading was employed to simulate real-world conditions with non-uniform requests. For the use case scenario, a user lands on the pet clinic's home page and then interacts with the chatbot to get directions using the genai-service. Based on the directions, the user registers as a Pet Owner using the customers-service, then proceeds to look for Veterinarians that match their needs using the vets-service.

A variety of performance metrics were collected during the simulation including latency, throughput, CPU utilization, and memory utilization. Latency is the response time experienced by a user when they perform a specific action on the Pet Clinic application, measured in seconds. Throughput refers to the quantity that successfully processed with a time interval. The api-gateway, as the entry point for the application, manages the distribution of traffic to secondary services. For each user request, we collected the CPU and Memory utilization of the api-gateway and the secondary service fulfilling the request.

D. Performance Model Construction

Next, data from the simulation was processed and used to build a performance model. The models were trained on key features: Number of users, CPU allocation per service, Memory allocation per service, Endpoint type, and Deployment type. The various Endpoints and Deployment setups were encoded as integers to use them as features. Endpoint types corresponding to secondary services responding to user requests were encoded as follows: */chatclient* = 1, */owners* = 2, and */vets* = 3. Deployment setups indicating the deployment environment in AWS were encoded as *ecs* = 1 and *ec2* = 2, respectively.

This study utilized the Random Forest Regression, Gradient Boosting Regressor, and XGBoost Regressor models. Based on previous research, the selected models showed potential to be effective for the workload, data type, and non-linear relationships between variables in this study [29]. The Random Forest Regression model constructs multiple decision trees during training and returns the average prediction of those trees as its output [30], which is known to be effective when trained on smaller datasets. Gradient Boosting Regressor is an ensemble algorithm that is effective on datasets with both numerical and categorical features. Its prediction is based on repeatedly building models and correcting errors to improve accuracy [31]. An improved boosting regressor is XGBoost (Extreme Boost), which uses regularization and computational improvement techniques to enhance performance [32].

To evaluate the model's predictive performance, the R^2 and RMSE are used as the evaluation metrics. R^2, a dimensionless metric ranging from 0 to 1, indicates the proportion of variance in latency that is explained by the input features. Higher R^2 values suggest stronger explanatory power of the model. In contrast, RMSE, expressed in seconds, which is consistent with the units of the latency variable, quantifies the average prediction error. Lower RMSE values imply greater accuracy.

Ultimately, assuming comparable CPU and memory allocation and depending on the deployment setup, the latency experienced by users for secondary services routed to by the api-gateway was modelled using the performance models.

IV. Results

A. Experiment setup

For this study, a Spring Boot application called Petclinic was employed. The application comprises eight services; however, for the purpose of this experiment, focus was on six services that represent a realistic usage scenario. These services include the config-server, discovery-server, api-gateway, genai-service, customers-service, and vets-service. The study investigation considered two distinct configuration setups: deploying on an EC2 instance and deploying within ECS (AWS Fargate), an AWS-managed service. The deployment process involved building a Docker container for each microservice, followed by the deployment of these containers. In the case of EC2, an instance is selected, provisioned, and configured prior to the application deployment, which entails the installation of Docker dependencies and other configurations. Conversely, in ECS (AWS Fargate), an ECS cluster was established along with a Namespace in Cloud Map, utilizing the task definition file for deploying the service in Docker containers.

To gather data for this research, a workload of users interacting with the application under the two different deployment configurations were generated. The test began with a workload of 5 users, followed by 10, 20, and 50 users sending requests to the application. Each user commenced their journey by accessing the api-gateway home page, subsequently interacting with the genai-service, then the customers-service, and finally the vets-service, which are categorized as secondary services. Metrics were collected during the routing of user requests to each of these three secondary services.

The experiment yielded results that provide valuable insight into the application's performance across various configurations, enabling the exploration of the following research questions:

- RQ1: What is the performance of a Spring-boot application when deployed in EC2 versus ECS (AWS Fargate)?
- RQ2: What is the cost of using a Spring-boot application deployed in EC2 versus ECS (AWS Fargate)?
- RQ3: How is infrastructure or deployment failure handled when a Spring-boot application is deployed in EC2 versus ECS (AWS Fargate)?
- RQ4: What are the prospects of enhancing the latency experienced by users when using Spring-boot applications deployed in EC2 versus ECS (AWS Fargate)?

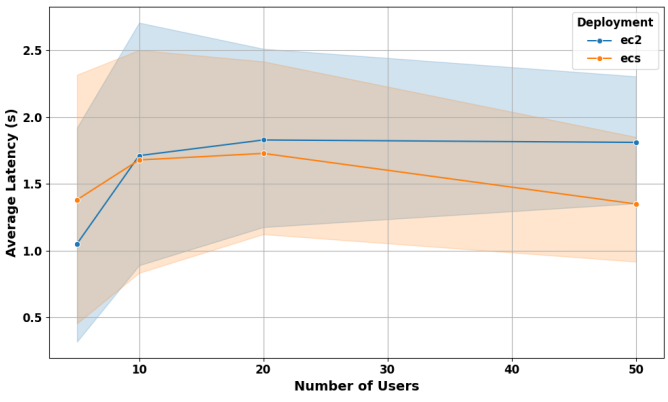

Fig. 3. Representation of the average latency experienced by users when completing the user journey at various traffic volumes

TABLE I
PERFORMANCE METRICS BY DEPLOYMENT TYPE AND USER LOAD

Users	Deployment	CPU Utilization (%)	Memory Utilization (%)	Latency (s)
5	ec2	0.65	83.56	1.05
5	ecs	0.63	68.16	1.38
10	ec2	10.58	83.57	1.71
10	ecs	0.63	68.16	1.68
20	ec2	12.56	83.61	1.82
20	ecs	0.61	68.16	1.72
50	ec2	37.87	83.65	1.81
50	ecs	0.60	68.16	1.35

B. Performance of AWS-managed versus Self-managed

RQ1 primarily investigates the average latency experienced by users when accessing the different deployment configurations. Table I presents the average latency for varying user traffic groups associated with each deployment setup. In the context of EC2 deployment, an observable spike in latency corresponds with increasing user traffic; however, average latency stabilizes as traffic rises to 50 users. In ECS deployments, there is an initial increase in latency, yet as illustrated in Figure 3, latency begins to decline once the simulation reaches 50 users. Furthermore, Table I indicates an increase in CPU utilization for EC2 deployments as user traffic escalates, while the ECS deployment does not demonstrate a significant rise in utilization. This finding suggests that ECS deployments handle increases in workload more effectively than EC2, indicating that the AWS-managed deployment provides a level of resilience during traffic spikes, even without enabling auto-scaling. The management of the underlying infrastructure for the application deployment is handled by ECS (AWS Fargate), whereas in EC2 deployments, the researcher is responsible for selecting the underlying virtual machine (T3.medium), which provides adequate CPU and memory resources for the containers.

Additionally, Table 4 illustrates the average latency experienced by users for requests made to the secondary services across different deployment configurations. Notably, latency was significantly higher for users in both deployment configurations when requests were routed by the api-gateway to the genai-service. This phenomenon is attributable to the genai-service's reliance on a third-party API for the chatbot functionality. Due to quota limitations imposed by the OpenAI API, requests were throttled once the limit was exceeded, resulting in increased latency. In contrast, for the other two endpoints, users experienced improved latency in ECS compared to EC2. This observation further reinforces the advantages associated with AWS-managed deployments over self-managed deployments.

TABLE II
AVERAGE LATENCY PER ENDPOINT FOR EC2 AND ECS DEPLOYMENTS

Endpoint	EC2 Avg Latency (s)	ECS Avg Latency (s)
customer/owners	0.08	0.07
genai/chatclient	5.55	5.32
vet/vets	0.10	0.07

C. Cost analysis

To address RQ2, a cost analysis was performed comparing the deployment of the application on EC2 versus ECS. In general, the cost of AWS services is determined by the specific region in which the resources are deployed, except for global resources [25]. For this reason, all our resources were deployed in the US East (Ohio) region (US-east2), allowing for a direct cost comparison in the same region.

For the EC2 deployment, we utilized a T3.medium EC2 instance, with an hourly rate of $0.0416 when CPU utilization remains below 20% and $0.05 when it exceeds this threshold [26]. We normalized the cost to reflect only the duration of our experiment in that deployment configuration, resulting in a total cost of $0.0026 for the 3.71 minutes of operation on EC2.

In contrast, ECS cost structure is based on allocated vCPU and memory for each task, and we deployed one task per service. Precisely, the hourly costs for vCPU and memory in ECS are represented as $0.04048 and $0.004445, respectively [24]. After summing the allocated vCPU and memory for the services, the ECS deployment incurred a cost of $0.0085 for the 1.21 minutes of operation.

Based solely on operational usage, the research findings suggest that ECS deployment is 3.27 times more expensive than EC2, even though EC2 execution time is 3.07 times longer than that of ECS. These results indicate that EC2 is more cost-effective for sustained, long-term deployments characterized by steady workloads, which effectively maximizes the utilization of EC2 instances. Conversely, while ECS (specifically the Fargate variant) may incur higher costs, it can prove more financially advantageous when its scalability is leveraged to manage more unpredictable workloads.

Although the pricing structure generally remains the same, there are several factors that could lead to AWS changing the pricing rate. Therefore, the AWS cost calculator should be referenced to get updated price rates for future cost analysis [23].

D. Infrastructure or deployment failure handling

To address RQ3, the research examined resilience and failure handling, including the delegation of responsibilities for deployments on EC2 compared to ECS. For the EC2 setup, which required to manage instance selection and maintenance, the application was deployed in containers on a T3.medium instance. Since this is a self-managed approach, it is one's

TABLE III
ACTUAL AND PREDICTED LATENCY BY ENDPOINT AND DEPLOYMENT

Users	Deployment	Customer service (avg. latency in seconds)				GenAI service (avg. latency in seconds)				Vet service (avg. latency in seconds)			
		Actual	RF	XGB	GB	Actual	RF	XGB	GB	Actual	RF	XGB	GB
5	EC2	0.0666	0.0670	0.0672	0.0675	3.5003	3.5407	3.5217	3.5196	0.0788	0.0785	0.0765	0.0763
5	ECS	0.0746	0.0745	0.0740	0.0743	4.0038	3.9880	3.9891	3.9895	0.0661	0.0660	0.0701	0.0687
10	EC2	0.0728	0.0726	0.0747	0.0734	4.9804	4.8762	4.9783	4.9807	0.0834	0.0828	0.0887	0.0870
10	ECS	0.0757	0.0752	0.0750	0.0778	4.8798	4.8774	4.8821	4.8797	0.0837	0.0829	0.0759	0.0770
20	EC2	0.0797	0.0800	0.0765	0.0752	5.3251	5.3446	5.3105	5.3121	0.0831	0.0836	0.0905	0.0883
20	ECS	0.0726	0.0726	0.0728	0.0753	5.0372	5.0228	5.0515	5.0501	0.0767	0.0769	0.0737	0.0740
50	EC2	0.0867	0.0867	0.0897	0.0896	5.9887	5.9954	5.9937	5.9927	0.1102	0.1100	0.1049	0.1066
50	ECS	0.0729	0.0732	0.0716	0.0704	5.9172	5.9548	5.9083	5.9105	0.0698	0.0697	0.0738	0.0731

responsibility to properly handle failures and patch the infrastructure. The health of the EC2 instance is tracked by AWS and provided to the AWS customer via CloudWatch to facilitate monitoring and failure handling.

Conversely, the ECS deployment with the Fargate launch type provides some resiliency and failure benefits as an AWS-managed service. Firstly, there is no single point of failure since each service is deployed independently in the ECS cluster. AWS orchestrates the deployment of each service on the appropriate EC2 instances, and it handles failovers. The health of each service is monitored by AWS, and automatic reallocation, restart, or redeployment is triggered as needed. Recovery is observable via CloudWatch logs. This effect is observed during the application deployment, where the task running the config-server was stopped to simulate a scenario where the service was unexpectedly shut down. ECS recognized that the task had been shut down and automatically deployed a new task instance for the config-server, registering it in Cloud Map. This demonstrates ECS's built-in guarantee of availability and self-healing in the event of service interruptions.

TABLE IV
PERFORMANCE COMPARISON OF REGRESSION MODELS

Model	Average RMSE	Mean R^2 Score
Random Forest Regressor	1.0061 ± 0.1524	0.8513
Gradient Boosting Regressor	1.0039 ± 0.1521	0.8522
XGBoost Regressor	1.0030 ± 0.1525	0.8525
Support Vector Regressor	1.0313 ± 0.1948	0.8469

E. Performance modelling and the prospect of performance enhancement

To address RQ4, machine-learning models were deployed to predict and model the average latency experienced by users interacting with a specific service. As seen in Table III, all three ML models performed strongly based on their R^2 and RMSE scores. XGBoost was shown to perform the best, achieving high accuracy and striking a good balance between bias and variance, with an R^2 value of 0.8525 and an RMSE of 1.0030 seconds. The GBR performed slightly worse but almost identically to the XGBoost regressor, with R^2 value of 0.8522 and RMSE of 1.0039 seconds, further cementing the effectiveness of the boosting regression algorithms. Random forest regression performed relatively worse with an R^2 value of 0.8513 and an RMSE of 1.0061 seconds, although it performed strongly in its own right. The R^2 shows that approximately 85% of average latency, considering all three models, can be accounted for by the features that were used. This indicates that any of the models can be considered for real-world deployment, depending on the specific performance needs.

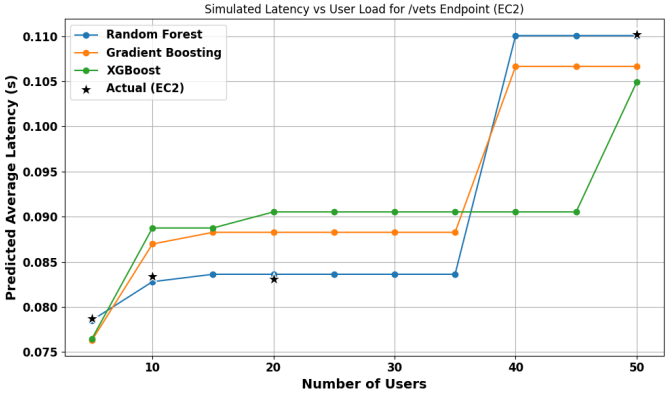

Fig. 4. Overlaying the actual average latency on the predicted latency, the three ML models for vet service in EC2 deployment setup.

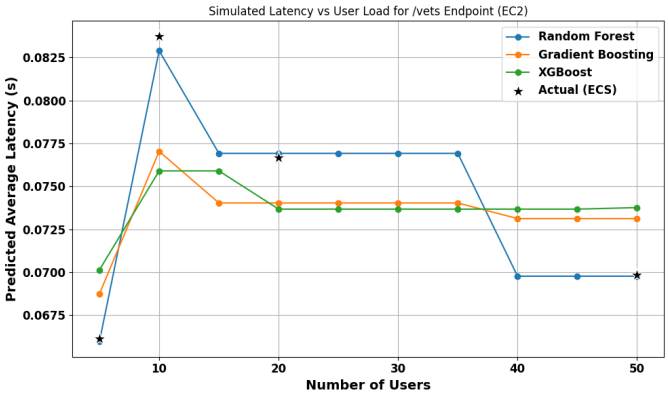

Fig. 5. Overlaying the actual average latency on the predicted latency, the three ML models for vet service in ECS deployment setup.

Table III shows the actual latency alongside the predicted latency values for each of the three models, indicating that all models made close predictions. Interestingly, RF predictions closely matched the actual values. Figure 4 and 5 show the

result of overlaying the actual average latency on the predicted average latency for the vet service only. That graph models the average latency predictions for users, assuming the number of users is increased in increments of five, from 5 to 50 users. In both graphs, RF consistently predicted an average latency that was almost identical to the actual average latency. GB followed a similar trend to RF, while XGB's predictions were less closely aligned with the actual latencies. This observation contrasts with what was expected based on the R^2 and RMSE values, where XGB had the best performance. Prior work has shown that RF models tend to overfit the training data, but may not generalize to unseen data [33].

The accurate prediction of average latency for a given service can inform decisions regarding deployment strategies that facilitate effective scaling. Although both AWS deployment environments offer scaling capabilities, the service-level auto-scaling option in ECS provides more flexibility. ECS's service auto-scaling feature permits horizontal scaling, dynamically adjusting the number of task instances corresponding to containers for a particular service. Conversely, the EC2 deployment requires that all services be housed within containers on the same EC2 instance. This limits the options for scaling since a complete redeployment to a different instance is necessary, rather than allowing for the selective scaling of individual services. In summary, although predictive scaling was not included in this experimental design, the results suggest that using the predictive machine learning model in conjunction with the ECS deployment configuration could yield highly effective results.

V. Discussion

The experiments demonstrate the performance differences between deploying a Spring Boot application using a self-managed EC2 setup versus an AWS-managed ECS Fargate configuration. It explores different topics that directly contribute to understanding performance, potentially improving it, and correlating performance to cost. These findings can inform infrastructure decision-making between the two deployment options.

Data from requests to the different deployment configurations was collected to compare performance. Considering the different deployment intricacies, there was a need to normalize the CPU and memory usage to ensure that the utilization percentages for services in EC2 could be compared to those in ECS. Both deployment configurations received the same workload, and the ECS experiment was completed faster than the EC2 experiment. Latency, CPU, and memory utilization were higher in EC2 compared to the ECS deployment, indicating that ECS enabled better performance. Thus, it is apt to conclude that the AWS-managed deployment option optimizes service deployment in ways that correlate to better performance.

The cost analysis showed that EC2 was more cost-effective than ECS in this experiment. Based on the AWS cost structure, EC2 cost is based on EC2 instance running time in hours, while ECS cost is based on the vCPU and memory utilization of the deployed services. Generally, EC2 is more cost-effective for steady workloads, while ECS could be more cost-effective for unpredictable workloads.

The AWS-managed deployment using ECS Fargate demonstrated inherent resiliency and self-healing. It monitored task deployments and redeployed when a service was terminated or stopped, improving service availability. Although ECS offered built-in failure recovery, both ECS and EC2 allowed for additional configurations to improve fault tolerance, including using CloudWatch logs.

All three performance models trained on the experimental data show strong latency prediction capabilities. They had R^2 higher than an RMSE of approximately 1 second. Although predictive scaling was not implemented in this experiment, the model's predictions could be used in proactive scaling decisions in future work.

VI. Conclusion

This paper evaluated two deployment setups on AWS to compare the performance of a Spring Boot application (Petclinic) deployed directly on Ec2 versus on ECS using Fargate, a fully managed service on AWS. To assess the performance, the study analyzed the latency experienced by users, cost implications, and resilience to failure across both deployment setups. Additionally, an ML model was built to model the performance of the application in both setups.

The results demonstrated that users achieved lower average latency when interacting with the ECS environment, deployed via Fargate (AWS-managed service) compare to the EC2 deployment. Cost analysis revealed that EC2 and ECS had distinct pricing structures, with EC2 being more cost-effective overall, specifically for steady traffic volume, while ECS might be beneficial for unpredictable traffic volume. Furthermore, using the Fargate launch type allows AWS to manage the deployment process and infrastructure, which includes the benefit of automatic failure handling and redeployments. Finally, Random Forest, Gradient Booting, and XGBoost models were trained and used for performing modelling. The ML models' performance result shows the feasibility of incorporating them into a predictive scaling solution in the future.

VII. Reference

References

[1] Oyeniran, O. C., Adewusi, A. O., Adeleke, A. G., Akwawa, L. A., and Azubuko, C. F. (2024). Microservices architecture in cloud-native applications: Design patterns and scalability. https://fepbl.com/index.php/csitrj/article/view/1554

[2] Jelani, U., Perveen, K., and Edward, E. (2024). *Cloud-Native Architectures: Building and Managing Applications at Scale*.

[3] Waseem, M. (2024). Containerization in Multi Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation. *DOI: 10.13140/RG.2.2.21439.32165*.

[4] Amazon Web Services. (Accessed 2025). *Implementing Microservices on AWS*. https://docs.aws.amazon.com/pdfs/whitepapers/latest/microservices-on-aws/microservices-on-aws.pdf

[5] Blinowski, G., Ojdowska, A., and Przybylek, A. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*, 10, 1–1. *DOI: 10.1109/ACCESS.2022.3152803*.

[6] Khan, B., and Faiz, M. (2016). Spring Boot and Microservices: Accelerating Enterprise-Grade Application Development. *DOI: 10.13140/RG.2.2.14266.50886*.

[7] Koneni, V. (2024). Microservices Architectures Using Spring Boot: Embracing Containerization and Observability. *International Innovative Research Journal of Engineering and Technology*, 11, 384.

[8] Amazon Web Services. (Accessed 2025). *AWS Managed Services*. https://aws.amazon.com/managed-services/

[9] Salim, S., Banerjee, D., and Verma, N. O. (2023). *Build a Cloud Automation Practice for Operational Excellence*. AWS Blog. AWS Blog Post

[10] Amazon Web Services. (Accessed 2025). *Developer Guide - AWS Fargate for Amazon ECS*. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html

[11] Palladio Simulator. (Accessed 2025). https://www.palladio-simulator.com/tools/

[12] Jindal, A., Podolskiy, V., and Gerndt, M. (2019). Performance Modeling for Cloud Microservice Applications. https://research.spec.org/icpe_proceedings/2019/proceedings/p25.pdf

[13] Ahmed, K. R., and Islam, M. M. (2022). *A Comparative Analysis of AWS Cloud-Native Application Deployment Model*. https://www.researchgate.net/publication/364137703_A_Comparative_Analysis_of_AWS_Cloud-Native_Application_Deployment_Model (Accessed: April 22, 2025)

[14] Bagai, R. (2024). *Comparative Analysis of AWS Model Deployment Services*. https://arxiv.org/pdf/2405.08175 (Accessed: April 22, 2025)

[15] Spring Petclinic Microservices. GitHub Repository. (Accessed 2025). https://github.com/spring-petclinic/spring-petclinic-microservices?tab=readme-ov-file

[16] Spring Petclinic Microservices. GitHub Repository. (Accessed 2025). https://github.com/spring-petclinic/spring-petclinic-microservices?tab=readme-ov-file

[17] Amazon Web Services. (Accessed 2025). *Amazon ECR Public Repositories*. https://docs.aws.amazon.com/AmazonECR/latest/public/public-repositories.html

[18] Amazon Web Services. (2025). *Specifications for Amazon EC2 General Purpose Instances*. https://docs.aws.amazon.com/AmazonECR/latest/public/public-repositories.html (Accessed: April 22, 2025)

[19] Amazon Web Services. (2025). *CloudWatch Metrics That Are Available for Your Instances*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html

[20] Amazon Web Services. (2025). *AWS Fargate for Amazon ECS*. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html

[21] Amazon. (2025). *AWS Fargate: Serverless Compute for Containers*. https://aws.amazon.com/fargate/

[22] Amazon Web Services. (2025). *AWS Cloud Map Namespaces*. https://docs.aws.amazon.com/cloud-map/latest/dg/working_with_namespaces.html

[23] Amazon Web Services. (2025). *AWS Pricing Calculator*. https://calculator.aws/#/addService

[24] Amazon Web Services. (2025). *AWS Fargate Pricing*. https://aws.amazon.com/fargate/pricing/

[25] Amazon Web Services. (2025). *Pricing Assumptions*. https://aws.amazon.com/calculator/calculator-assumptions/

[26] Amazon Web Services. (2025). *Amazon EC2 On-Demand Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/

[27] Amazon Web Services. (2025). *What is Amazon Elastic Container Service?*. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

[28] H. Mohamed and O. El-Gayar. (2021). *End-to-End Latency Prediction of Microservices Workflow on Kubernetes*. Proceedings of the 54th Hawaii International Conference on System Sciences (HICSS). https://scholarspace.manoa.hawaii.edu/bitstreams/ba00be83-fe9a-47bf-9212-d289785a2b03/download

[29] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. M. Elfadel. (2022). *Proactive Random-Forest Autoscaler for Microservice Resource Allocation*. IEEE Access. https://www.researchgate.net/publication/366841998_Proactive_Random-Forest_Autoscaler_for_Microservice_Resource_Allocation

[30] AnalytixLabs. (2022). *Random Forest Regression — How it Helps in Predictive Analytics*. Medium. https://medium.com/@byanalytixlabs/random-forest-regression-how-it-helps-in-predictive-analytics-01c31897c1d4

[31] Data Science on Medium. (2023). *Gradient Boosting Regressor, Explained: A Visual Guide with Code Examples*. Medium. https://medium.com/data-science/gradient-boosting-regressor-explained-a-visual-guide-with-code-examples-c098d1ae42

[32] F. Omarzai. (2024). *XGBoost Regression In Depth*. Medium. https://medium.com/@fraidoonomarzai99/xgboost-regression-in-depth-cb2b3f623281

[33] Mwiti, D. (2023). *Random Forest Regression: When Does It Fail and Why?*. Neptune.ai. https://neptune.ai/blog/random-forest-regression-when-does-it-fail-and-why