National College of Ireland

Efficient Auto-Scaling for Microservices in Cloud Environments

MSc Research Project
MSc Cloud Computing

Ritesh Panaganti
Student ID: X23344563

School of Computing
National College of Ireland

Supervisor: Jorge Mario Cortes Mendoza

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……. ……Ritesh Panaganti……

Student ID: ……X23344563……..……

Programme:…… MSc Cloud Computing…… **Year:** ……2025……..

Module: …… Research Project……..……

Supervisor: ……Jorge Mario Cortes Mendoza……..……

Submission Due Date: ……15/09/2025……..……

Project Title: Efficient Auto-Scaling for Microservices in Cloud Environments……..……

Word Count: ……6589…… **Page Count**……15……..……..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Ritesh Panaganti……

Date: ……15/09/2025……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Efficient Auto-Scaling for Microservices in Cloud Environments

Ritesh Panaganti
X23344563
National College of Ireland

Abstract:

Modern business applications implement microservices architecture as their popular design methodology for developing cloud-ready modular systems. The use of computing resources for maintaining high performance continues to present difficulties in cases where infrastructure capabilities are insufficient. These consist of problems like CPU overload and higher response time at spiky moments of increased traffic. The research focuses on how to apply auto-scaling features, especially horizontal scaling to microservices it deploys via Amazon Elastic Container Service (ECS) with containers launched with the Fargate launch type. The research aims to put into practice horizontal scaling approaches through performance monitoring and system metrics primarily CPU utilization, which depend on Amazon CloudWatch. The research explores resource allocation alteration systems that react to workload with the intention of optimizing cost-performance ratios and avoiding unnecessary resource allocation. In order to simulate dynamic workload and to analyse the autoscaling behavior, Locust performance testing tool was used to create real-time HTTP traffic. The system has implemented two lightweight approaches of load balancing which are Round Robin and Random in order to distribute requests in a balanced manner. Across all the scenarios Round Robin achieved average response time of 221ms compared with Random of 280ms. Under load, CPU briefly peaked 95% to 100% at scale-out and then stabilized around 45% per service once the additional task was healthy with balanced utilization. The research is aimed at providing practical data as to the implementation of scalable microservice components in environments with resource constraints when using clouds environments.

Key Words: Auto-scaling, Microservices, Amazon ECS, Cloud Computing, Containers, Round Robin, Random, Loadbalancing, Locust, Amazon ECR

1. Introduction

The modern business applications tend to build such scalable, cloud ready systems that are modular, can be independently deployed, and easier to maintain using the microservices architecture (Arya, 2024). This architectural style subdivides large applications into smaller self-contained services using standard APIs that are independently controlled for deployments and that can be done so more efficiently in response to changes in business demand (Arya, 2024; Kambala, 2025). Since microservices are meant to work in environments that are dynamic and flexible, cloud environments are the best place for them to be. They can be efficiently deployed so that they have improved resource separation, fault tolerance, and greater portability by containerizing (Matos 2024). While its advantages are evident, microservices are notoriously challenging to implement in resource and infrastructure availability in the clouds. Workload characteristics of each service may vary

and demand and resource usage may vary as well. Services could either over provision by utilizing too much devoted resources or under provision resulting in capacity bottlenecks Marques (2024). For small scale or cost constrained deployments, the cost, infrastructure and cloud costs combine to make or break the feasibility of adoption.

However, such advanced solutions also exist including Machine Learning (ML) based resource schedulers (Hu, 2024) or Deep Learning (DL) based optimization models (Li, 2025) but need complex configurations, expensive infrastructure or in-depth knowledge in applied artificial intelligence. Though widely used as a container orchestration and auto scaling tool, MSARA (Hu, 2024) recommends that kubernetes is resource expensive to set up and operate within minimal environments. As a consequence, there is a gap on the research of lightweight, practical auto- scaling strategies that could be applied in more accessible cloud platforms.

This research is intended to fill that gap for the Amazon Elastic Container Service (ECS) a managed container orchestration service. The other advantage of using Amazon ECS is that it has native auto-scaling capabilities which can be integrated seamlessly with Amazon CloudWatch for real time metric based scaling. The built-in tools by kubernetes are to be used to implement horizontal scaling strategies for containerized microservices so that we can obtain cost effective and adaptive scaling in constrained cloud environments. This approach is taught to be simple, pragmatic and replicable, yet not requiring expensive models or high resources. The other implementation of this project is lightweight Round Robin and Random load-balancing at the API gateway to balance requests and test autoscaling.

What are the ways of auto-scaling design mechanisms in microservices cloud-based applications to enhance efficiency in the utilization of resources?

The study aims to realize how containers and dynamic microservices can be employed to scale its microservices in an attempt to reduce idle resources quota by increasing overall capacity due to the workload nature. The main aim is to implement scale strategies that could be deployed as well as being economical but appropriate on actual cloud environments.

The research provides practical design knowledge about cloud-native applications by establishing an easy-to-implement basic auto-scaling method for microservice applications. Amazon CloudWatch interacts with the native auto-scaling feature of Amazon ECS to conduct metric-based horizontal scaling, it does not need complicated configuration and ML algorithms. This research assesses scaling technologies in order to come up with allocation techniques that are more efficient with regard to system without rendering the system unstable in various workload contexts. Also, lightweight load balancing Round Robin and Random allows even distribution of incoming request to the services. Locust has been deployed as a traffic generator to generate real-time Http requests simulating the traffic to assess the behavior of the system under load and validate the condition of triggering autoscaling. This solution most perfectly serves affordable deployments of small size which operate in programs and developer environment test sites where powerful platforms are not available. This implementation gives system architects and developers with a replicable and affordable solution that works well for deploying microservices throughout modern cloud systems.

2. Related Works

This section provides an overview and the main characteristics of the recent related works Table1.

References	Strategy	Technique	Main Goal	Metrics	Dataset / Workload	Simulation/ Environment	Configurations
Arya et al., 2024	Microservices adoption	Kubernetes Orchestrated Deployment	efficient deployment	utilization, count, Scaling	own dataset	Minikube and Kubernetes setup	Kubernetes cluster; HorizontalPodAutoscaler
Biegel et al., 2024	Microservice Orchestration	Photogrammetry with OpenDroneMap	deploying and managing microservices	Compute time, Cost-efficiency, Resource usage	own dataset	AWS	AWS Lambda, AWS ECS (Fargate), Docker S3 bucket
Li et al., 2025	Workflow based Microservice Scheduling	GNN	Cost	Deadline violations, Reliability, Cost	CyberShake and Inspiral workflow	DAG Workflow Simulator	VM cluster
Hu et al., 2024	SLO Resource Management	Meta learning and RL	resource consumption and SLO violations	Adaptation speed, Resource cost	own dataset	RL Simulator on Kubernetes	Kubernetes cluster
Miao et al., 2024	Fog computing with LoRa microservice architecture	Containerized microservices on edge servers combining sensors and cameras	smart monitoring and control environments (agriculture)	latency, quality, energy consuption, cost, distance	own dataset	Real deployment on cloud (AWS,Azure)	Microservices, REST APIs, Docker
Lu et al., 2025	two stage load balancing	GWO and SA	container deployment and execution latency	Utilization, task completion time	own dataset	container scheduling simulator	node cluster
Kambala et al., 2025	Cloud-native Microservices Integration	enterprise application design	Integration of microservices with cloud computing and enterprise application design	NA	NA	NA	NA
Matos et al., 2024	Microservice placement optimization	RL Microservice Scheduling	communication latency	Average latency, Microservice placement accuracy	own dataset	kubernetes clusters	Multi-region clusters, Kubernetes v1.22
Ahmad et al., 2025	Reactive and Proactive autoscaling	Threshold based and Prophet ML model	Autoscaling	Over/Under provisioning, Startup time, Latency	own dataset	MAPE and MAPE-KI	Kubernetes, Custom HPA controller in Python;
My Research	Microservice Orchestration	Amazon ECS and Cloudwatch	resource efficiency	utilization, scaling and cost	Locust tool (generate traffic)	AWS	ECS task definition, CloudWatch Docker Compose, Python, Flask

Table 1: Related Works table

Arya et al., (2024) introduces the migration of monolithic to microservices and takes into consideration Kubernetes Horizontal Pod Autoscaler (HPA) as the default scaling measure. It mentions weaknesses of HPC such as slow scaling up period and non-predictive when a system is dynamic.

Biegel et al., (2024) propose a pipeline implemented on Amazon ECS that can be used to transform drone imagery to 3D models to populates digital twins. It also does not analyze autoscaling logic, but it demonstrates on low-cost and scalable way to utilize cloud computing in order to support photogrammetry applications that are compute-intensive.

Li et al., (2025) propose a DL algorithm of Graph Neural Network (GNN), to schedule containerised microservices subject to deadline and reliability calls. It is economically and performance aware because it learns about service dependencies and bases the placement of containers based on them.

Hu et al., (2024) introduce MSARS, a hybrid meta-learning and Reinforcement Learning (RL) framework in which resources are allocated in a dynamic manner according to the Service Level Objectives (SLOs). It is fast to respond to changes in workloads and seeks

to achieve minimal resource waste but at least guarantees in microservice operation.

Miao et al., (2024) proposed a smart agriculture system that is safe and low cost in terms of energy consumption, it is based on a microservice based fog computing architecture with LoRa communication able to sense animal intrusions in the rural locations. It includes cheap PIR detectors and rotating cameras to achieve low latency animal system detection and position forecast at the edge, which requires less bandwidth and energy consumption than camera-only systems.

Lu et al., (2025) proposed an optimized two-stage solution to container deployment, which would improve load balancing and utilization resource usage among the virtual machines. It employs a greedy algorithm to coarse-placed containers and a genetic algorithm in the case of resource allocation on each machine. The hybrid approach substantially improves the existing approaches, such as Grey Wolf Optimization (GWO) and Simulated Annealing (SA), in terms of the balanced resources distribution and shortest task completion time.

Kambala et al., (2025) discusses the topic on how microservices and cloud computing have changed the way enterprises are designed. The agility, resilience and efficiency in the operations provided by the combination of modular microservices with the scalability and flexibility offered by platforms in the cloud. The author also discusses about such technologies as containers, Kubernetes and Continuous Integration and Continues Delivery / Deployment (CI/CD), concerning such problems as service orchestration, fault tolerance and tendencies like AI and edge computing.

Matos et al., (2024) introduces scheduling plans based on RL and are focused on a distributed cloud environment with a geographical coverage in which a microservice dependency has been considered to improve on latency effects. Realistically, at the microservice level, the RL agent will have to place microservices as close as possible to each other minimizing network latency.

Ahmad et al., (2025) propose Smart HPA is a reactive scaling with a balance in real time between microservices, where ProSmart HPA adds predictive ML based scaling to real time. The two solutions will make Kubernetes clusters more efficient and less over/under provisioned.

2.1 Auto-Scaling Strategies in Cloud-Native Microservices

Auto-scaling operates as the main element for maintaining optimal efficiency and response times in cloud-native microservices. Ahmad (2025) developed Smart HPA alongside ProSmart HPA to minimize both over and under provisioning through respective reactive and proactive scaling approaches. The goal of their work was to develop simpler alternatives to standard Kubernetes HPA, using current system metrics along with predictive decision-making processes. Matos (2024) worked on service latency reduction by developing reinforcement learning-based scheduling of microservices for optimized latency and intelligent service deployments. The two studies use different approaches to system performance improvement through automation but they approach it at varying levels of complexity.

The authors Arya (2024) and Kambala (2025) promote cloud-native microservices with agile and modular structures that ensure easy deployment and efficient results. The suggested orchestration systems must demonstrate strong scalability without losing their manageability, even in reduced operational environments.

The work presented by Ahmad (2025) simplifies Kubernetes environment scaling

logic, yet Matos (2024) shows the importance of adaptive learning-based approaches. The solutions require Kubernetes infrastructure together with extensive resources that make them hard to apply to constrained deployment environments. The shortcomings demonstrate the importance of investigation which seeks to create auto-scaling techniques that boost resource efficiency in microservice-based cloud applications through easier and more beginner-friendly tools. This study compares container orchestration tool with its microservice resource scheduling functions.

In contrast to the existing works, which exist on complex orchestration platforms or predictive models, the proposed research focuses on implementation of native autoscaling in Amazon ECS, which is supported by simple, stateless load balancing algorithms.

2.2 Container Orchestration Platforms and Resource Management

Speed of adaptation and scalability are directly added to a microservice application by the way controllers manage containers. DL Model for Microservice Container Configuration (DeepMCC) graph neural Network model can be optimized to schedule operations of containerized workflow (Li, 2025). In their study, the researchers improve task placement performance by addressing budgets along with time constraints to bring about improved results regarding the operation of clouds. Meta-learning to efficiently generate SLO resource Allocation plans and the use of reinforcement learning with adaptive Scaling of microservice resources (MSARS) represents reinforcement and meta-learning hybrid model of dynamic resource allocation (Hu ,2024). Successful performance improvements were applied in the studies but this required esoteric infrastructural requirements as well as extensive technical prerequisites. The approach in research follows this dimension because it explores a container-based mechanism that simplifies auto-scaling, analyzing the impact of practical limitations on cloud platform scalability regarding microservice deployment. In contrast to DeepMCC and MSARS, which need AI pipelines and orchestration logic, the strategy in this research can leverage ECS-native scaling and simple CPU-based policies-thus being deployable in narrow environments with no particular facilities.

2.3 Resource Conscious Deployment and Practical Constraints

Scaling microservices have operational limitations in the context of cost-based projects. With his work, Biegel (2024) demonstrated that the use of ECS and the leverage of the AWS serverless technology could make dream 3D modelling systems cost-effective and scalable. Miao (2024) carried out the study of reinforcement learning techniques that are applied to the optimization of container scheduling in distributed systems that possess a large amount of resources. Biegel focuses his research on developing user-friendly interfaces of the large-scale process whereas Miao is directed towards achieving better performance capabilities of the complex systems in stack infrastructures.

The different strategies illustrate that superior solutions remain beyond reach for users under performance limitations. By focusing on ECS in his work Biegel provides backing to this research direction of developing simple and practical auto-scaling solutions for commonly used platforms. This research will finish the evaluation of dimensions prior to specifying the research gap.

Further developing the ECS-based strategy suggested by Biegel, this paper does not require serverless technologies and AI models and proposes the integration of ECS-native scaling with low-friction load balancing, workload simulation.

2.4 Research Niche

The analyzed studies reveal beneficial knowledge about automated scaling methods and management systems combined with performance efficiency approaches. The predictive

and learning-based models achieved great potential through advanced techniques such as Smart HPA, DeepMCC and MSARS. The utilization of Kubernetes along with comparable intricate platforms together with expert-level capabilities remains typical for these solutions. A lack of research exists currently about lightweight native solutions for dynamic scalability in containerized microservice deployment systems working with reduced resource needs.

This study examines Amazon ECS with CloudWatch when it comes to the use of metrics in scaling. This study develops an efficient yet affordable method for improving resource usage which specifically addresses the core research question that examines the best approaches for microservice-based cloud application auto-scaling.

With an integration of ECS-native scaling, lightweight load balancing, and real-time simulation of workload, this paper presents a highly repeatable alternative with less complexity compared to AI-based frameworks of autoscaling.

3 Research Methodology

A. Research Procedure

This research proposes an experimental procedure, in which the efficiency of metric-based autoscaling strategies will be tested in a microservices ecosystem, implemented by the use of Amazon ECS with Fargate launch type. The goal is to identify a threshold level of CPU utilization that improve the performance and scale of containerized cloud without impact the execution of the workload. It will have several microservices of which one will serve as the API Gateway and the other two will serve as backend microservices involving performing compute and logging duties. The Docker is used to containerize the services which are deployed on ECS Fargate. To handle the incoming traffic, an Application Load Balancer (ALB) is set up, and it has listener rules to send HTTP requests to target groups related to the microservices (Figure 1). The routing mechanism will provide the redirection of traffic to the right microservice depending on the path of request.

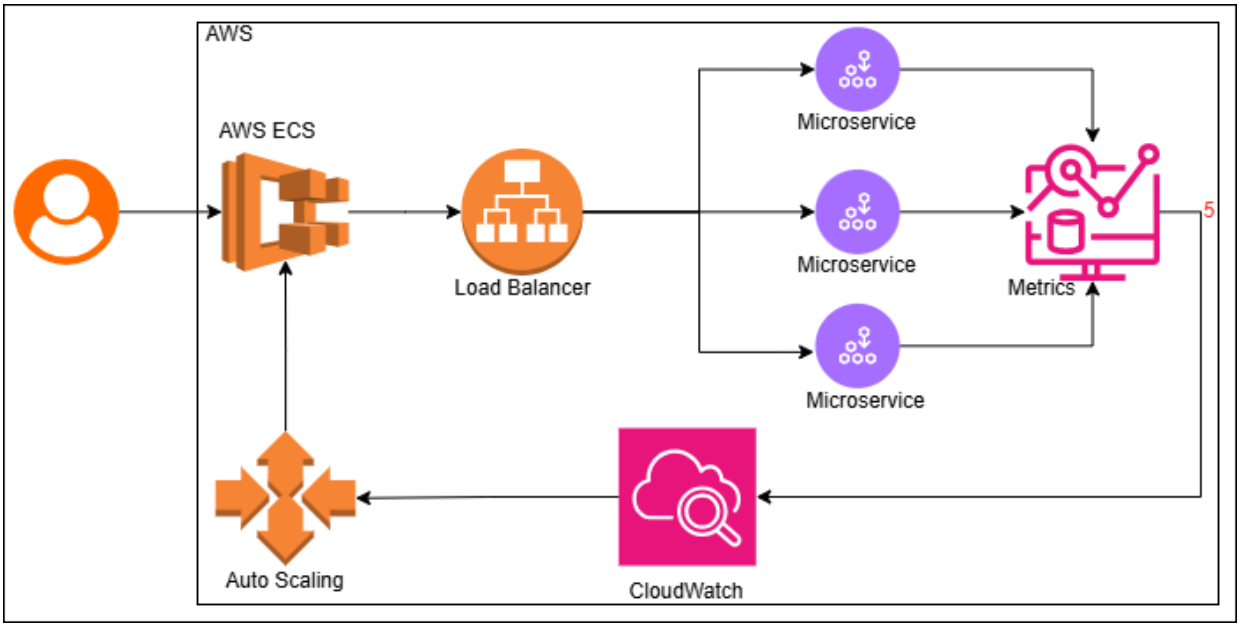


Figure 1: Auto-scaling architecture for ECS based microservices using CloudWatch and auto-scaling

To determine the effect of load balancing strategies on the resource use and scaling, two important test cases are adopted. In the former, there is only one backend service, but there are multiple containers that are running and the requests are managed by Round Robin or Random logic. This will enable the comparison of the performance under scaling when

traffic is put on a single service. In the latter case, both backend services are implemented, and the requests are distributed with the use of the same two methods of load balancing among them via the API Gateway. This multi-service topology is truer to the scenario in the real-world microservice deployment, and enables the assessment of the scaling behavior of heterogeneous services. In each setup, ECS target tracking policies with the different threshold percent CPU utilization are used. The CPU metrics calculated by CloudWatch are checked every minute and autoscaling operations are automatically performed according to the thresholds planned.

B. Tools and Technologies

Various open-source and cloud-native tools are applied in the course of this research. The operation of Amazon ECS with the launch type Fargate is the container deployment and management platform. Each microservice is built and packaged with the help of Docker, and Amazon ECR stores the images. Amazon CloudWatch is a real-time monitoring tool that is incorporated to instigate autoscaling actions on the basis of CPU measures. Target groups are set up on ALB so that their traffic can be routed according to the rules of listeners. The implementation of the microservices is performed in Python and Flask. Locust is a performance testing framework which is open-source and it is used to generate traffic and impose a workload pressure on the system. To configure services and inspect the events of the autoscaling process as well as observing the metrics of performance during the experiments, we will use the AWS Management Console and CloudWatch dashboards.

C. Setup Details

Every experiment is aimed at testing the system within the controlled and repeatable context. The architecture of the deployment is different using single-service and multi-service modes. In each of the setups, distribution strategy is toggled to Round Robin and Random algorithms as a way of modelling the effects of the distribution strategy on CPU consumption and scalability of the performance. The target tracking policy of each ECS service is configured with each with the target tracking at different utilization rates in the CPU and limitations between multiple tasks. All the incoming HTTP requests made to Locust and spawned by the user are forwarded to the target groups on the ALB with the help of the path-based routing. Health checks make sure that only healthy containers get to get traffic. The test cases are run individually, with enough time left to cool-down in between tests so that previous tests do not interfere with any other tests.

D. Data Collection and Analysis

The data on the performance is gathered with the help of a mixture of Amazon CloudWatch values, Amazon ECS service logs, and Locust reports. Amazon CloudWatch has extensive data on the CPU utilization history, number of tasks that have been created over time, and the shift happening in regards to alarm indicating the scale-in or scale-out process. Amazon ECS logs capture the workflow of provisioning and shutting down of tasks and this is useful in computing scale-out delay. Locust will also generate the statistics at the request level as the estimated response time, throughput and probabilities of failures. The results of every test run are combined and statistically analyzed with the help of simple statistic procedures such as calculating the mean and peak usage of the CPU, average response time, the number of scaling event and the time of the scaling. Holding all other variables constant, analysis removes the effects of all other variables so effects of the two variables, threshold and load balancing strategy on system behavior can be obtained.

4 Design Specification

A. System Architecture

The proposed system is modularly built on the microservices architecture on Amazon ECS with the Fargate launch type. It is comprised of several individually containerized microservices, namely api-service, api-round-robin, compute-service, and logger-service. These two services of API Gateway present the client-facing endpoints and apply the request routing logic to the backend services. The microservices include the compute and logger whose roles entail computation and logger handle the backend processing. Each of the services is started as a distinct ECS service with a light weight python 3.9.13 based Docker containers with version 28.3.0 and each task has 0.25 vCPU and 0.5 GB memory assigned (Table 2), this is the minimum configuration. Various services get registered with an ALB that does the routing of the external traffic based on listener rules according to Path. The ALB has been set up with four target groups with each of them referencing one of the microservices and will direct requests according to URL path received. Since round-robin has the benefit of spreading arrivals evenly and reducing hot spots along queues and aligning services, so this architecture offers purity of separation of concerns, scalability and optimum routing across the loosely coupled services.

Services	CPU	Memory	Min Tasks	Max Tasks
api-service	0.25 vCPU	0.5 GB	1	2
compute-service	0.25 vCPU	0.5 GB	1	2
logger-service	0.25 vCPU	0.5 GB	1	2
api-round-robin	0.25 vCPU	0.5 GB	1	2

Table 2: Table showing ECS tasks configurations for the microservices

B. Load Balancing and Routing Logic

The ALB manages traffic to the system by forwarding the requests according to the specified listener rules. The Round Robin service, which represent a custom logic defined in Python to alternate the request between the compute-service and logger-service. The unspecified route will be redirected to the api-service where the incoming requests received will be redirected to the two backend services using Random distribution strategy. This two API model opens the possibility of conducting comparative study of effects of load balancing paradigms on the behavior of autoscaling. As well as, compute and logger are processed by route-based target groups and redirected to the compute-service and logger-service respectively without traversing the API Gateways. With this arrangement, direct access to microservices is possible as well as an API-mediated routing, which makes the architecture appropriate to carefully controlled experimentation and realistic simulation of microservice interactions. The API containers are used to deploy all load balancing logic and no external orchestrators are needed, nor proxies.

C. Autoscaling Design

In order to allow performance flexibility to changing workloads, all ECS services have target tracking-based horizontal autoscaling policies. In Amazon CloudWatch, the monitoring of real-time CPU use is made with the triggers of autoscaling being established on different thresholds. Services are also set with a minimum desired and Maximum task counts

respectively. CloudWatch alarms assess the metrics every 1-minute, as this is the minimum time to access the metrics and initiate scale-out or scale-in actions on the basis of the changes in the CPU usage. Using this set up, each of the services has the ability to scale independently with increased demand and prevent over-provisioning. The autoscaling configuration is not based on models of predictive or ML in any way, however, it focuses on being simple, responsive, and native to ECS. Although no official functional and non-functional requirements were placed, the system was deliberately constructed to be light-weight, maintainable, and cost-responsible perfect to experiment upon the real-world limitations.

5 Implementation

The deployment is a containerized microservices application that is fully hosted on AWS along with the use of Amazon ECS Fargate. The solution consists of four loosely coupled microservices that are api-service api-round-robin, compute-service and logger-service. All the services were developed using Python with Flask framework which was containerized by Docker. The images generated were pushed to the Amazon ECR through the AWS Command Line Interface (CLI) so that the imaged would be deployed versioned and reusable. Deployment was done via the AWS Management Console and every microservice had the individual ECS Fargate service configuration. Each service had its own definition of tasks the limits of resources were defined at 0.25 vCPU and 0.5 GB of memory. The services were made available through an ALB, which had four path-based listener rules. These listener rules were used to allocate incoming requests to specific target groups /compute to the compute service, /log to the logger service, / to the API Gateway with Random logic and /round to the API Gateway with Round Robin logic. This configuration would have enabled direct access of backend services as well as an ability to have indirect routing via API Gateways in order to test load balancing strategies experimentally. System and resource metrics were tracked through the Amazon CloudWatch that monitors the performance of the system. Autoscaling policies target tracking were established in each ECS services with CPU thresholds to be measured. These policies allowed flexible scale-out and scale-in dynamics according to real-time workloads by each task where the maximum scaling is constrained between the tasks.

The outcomes of this implementation consisted of microservice execution, target group configurations that were in use under the ALB, metric-monitoring of the task scaling that was being done through CloudWatch, ECS service logs, and its corresponding deployment and scaling activities. The system architecture was stable, reproducible, and matched the best practices of scalability and modularity of cloud-native solutions. Such implementation was used as a foundation of performance evaluation and analysis in later parts of the research.

6 Evaluation

The evaluation compares two traditional load balancing strategies that is round-robin load balancing and a random load balancing, over a back-end composed of a compute service and a logger service deployed on Amazon ECS using Fargate. All tasks have 0.25 vCPU and 0.5 GB of memory and service will be limited with a minimum of one task and a maximum of two tasks. Auto-scaling uses CloudWatch target-tracking on CPU utilization taking the minimum time that is one-minute metric periods, a scale-out cooldown of 60 seconds, and a scale-in cooldown of 300 seconds. The CPU objective is set to 60 %, 70 %, or 80 % depending on the scenario. To interpret scaling behaviour, two timings are used consistently

the scaling reaction time, defined as the elapsed time from the first sustained CPU reading at or above the objective until a new task appears in Pending state, and the capacity readiness time, defined as the elapsed time from that Pending event until all desired tasks are Healthy and serving traffic. For every run the analysis records throughput (RPS), latency percentiles, error rate, and per-service CPU.

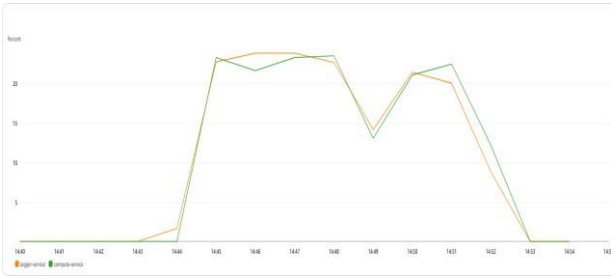
6.1 Identifying the best strategy

Exp.	Strategy	Auto-scaling	CPU target	Average response (ms)	RPS	Requests	Errors (%)
1	Random	No	No	320.82	8	3,388	0
2	Round-robin	No	No	239.12	8	2,925	0
3	Random	Yes	60%	255.96	48	26,012	0.02
4	Round-robin	Yes	60%	199.61	48	25,799	0.02
5	Random	Yes	70%	301.85	38	27,700	0
6	Round-robin	Yes	70%	237.28	50	30,958	0
7	Random	Yes	80%	240.08	45	32,372	0
8	Round-robin	Yes	80%	209.15	37	24,028	0.04

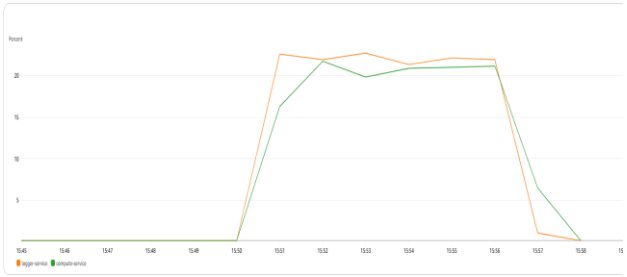
Table 3: Comparison between Round Robin and Random Load Balancing

The average response time of the round-robin load balancer is significantly faster than the random load balancer and both have the same throughput and produce no errors. The average response time is calculated by adding processing time, queuing time and network time. This provides deterministic alternation a built-in latency improvement over the deterministic alternation prior to the addition of auto-scaling dynamics. With target-tracking enabled at a 60 % objective (Figure 2), round-robin reduces average response from 255.96ms to 199.61ms at comparable throughput and negligible error rates (Table 3). Timings have been captured which can be observed as per the controller design. Reaction occurs after several one-minute evaluations above the objective, and readiness is achieved roughly one minute later when the additional task becomes healthy. At a 70 % objective (Figure 2) the advantage persists.
Round-robin delivers 237.28ms compared with 301.85ms for random, while maintaining higher throughput and a zero-error rate in both cases. CPU traces indicate that utilization of the two services remains closely aligned under round-robin, limiting single-service saturation near the trigger point.

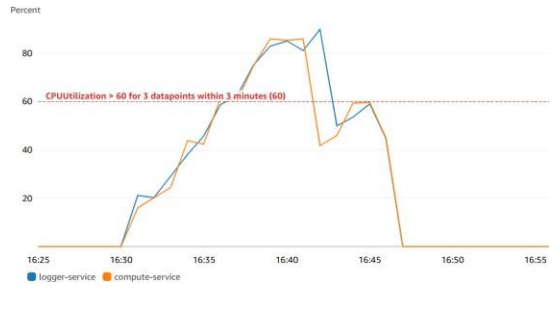
At an 80 % objective (Figure 2), round-robin again provides the lowest average response 209.15ms and 240.08ms with negligible error rates (Table 3). This setting also reduces the number of scale events relative to lower objectives while preserving acceptable latency once the extra capacity is ready. Considering the traditional and all auto-scaled cases (Table 6.1) together, round-robin is identified as the preferred load balancing strategy, it consistently yields lower latency, sustains throughput, and keeps the two services balanced.



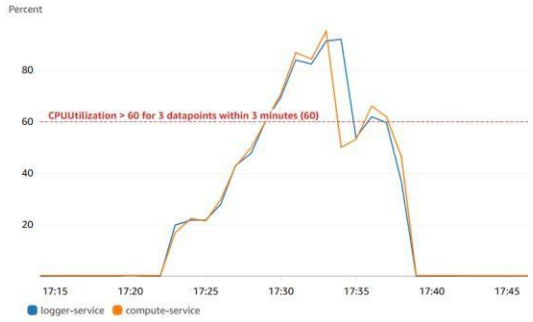
Random Load balancing



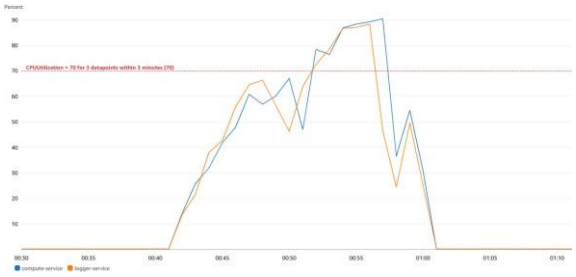
Round Robin



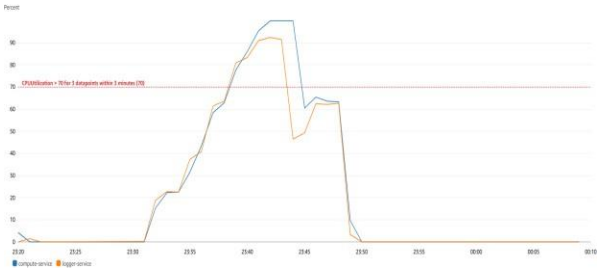
Random at 60%



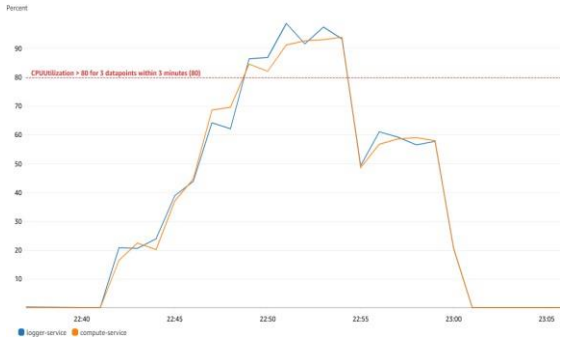
Round Robin at 60%



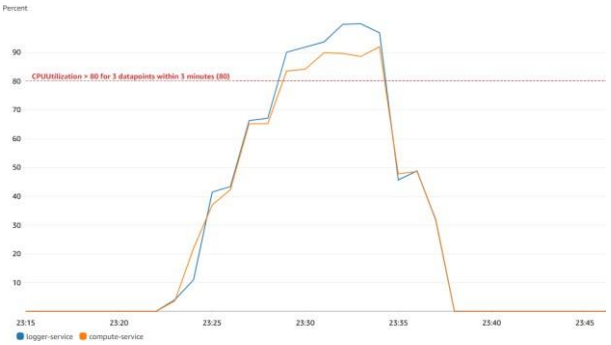
Random at 70%



Round Robin at 70%



Random at 80%



Round Robin at 80%

Figure 2: CPU Utilization of Round Robin and Random Load Balancing

6.2 Round Robin Load Balancing Validation Under Different Workloads

Workload	Load	Avg RPS	P50 (ms)	P95 (ms)	Errors (%)
Steady	60 users, 15 min	34.79	150	300	0.02
Step ramp	10→40→80→120 users, 5-min steps (20 min)	28.92	180	1700	0.003
Bursty spikes	40 baseline; 180-user spikes for 60 s every 5 min (25 min)	22.33	160	4300	0.003
Soak	80 users, 30 min	36.5	140	1100	0.02

Mixed idle/peak	three cycles of 5 min idle → 5 min peak (30 min)	31.18	950	5000	0.02

Table 4: Different workload results of Round Robin Load Balancing

The steady workload (Figure 3) demonstrates predictable scaling behaviour and stable latency after the additional task becomes effective (Table 4), with throughput near 35 RPS, P95 of 300 ms, and an error rate of 0.02 %. CPU usage levels approximately 40% on both services, and that can be interpreted as pretty evenly loaded under round-robin.

The step-ramp workload (Figure 3) raises demand in discrete stages and reveals the controller's timing. The CPU objective is crossed during the growth phase, the service reacts after the sustained-threshold condition is met, and readiness is achieved when the new task becomes healthy. The transient increase in latency during the step transitions subsides once capacity is in place, resulting in an overall P95 of 1.7 s with an almost zero error rate.

The bursty workload (Figure 3) probes short peaks that last only sixty seconds. The peaks increase the tail of the latency distribution P95 ≈ 4.3 seconds while leaving the median low and the error rate negligible (Table 4). This agrees with the sustained-threshold requirement of the target-tracking policy that spikes are short-lived and will not be sufficient to elicit other successive actions, therefore the available capacity can buffer them without challenging correctness.

The soak workload (Figure 3) examines stability over thirty minutes at a constant demand level. After an early reaction and readiness interval the system maintains a flat performance envelope, with P95 around 1.1 s and a low error rate, a scale-in occurs at the end of the run after the prescribed cooldown, indicating absence of oscillation.

The mixed idle/peak workload (Figure 3) alternates five-minute idle and peak periods. The first peak triggers a scale-out and readiness shortly. later peaks remain below the sustained threshold, causing temporary increases in P95 during peaks that recede in the subsequent idle intervals. Errors remain around 0.02 % across the full run. Taken together, the five profiles (Table 4) confirm that round-robin at an 80 % CPU objective offers robust performance across steady, growth, burst, long-running, and cyclic conditions with predictable scaling and balanced utilization.

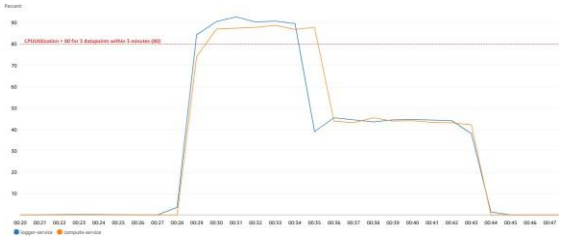


Fig : Steady Load

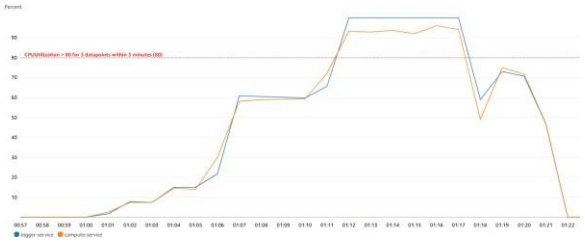


Fig : Step Ramp

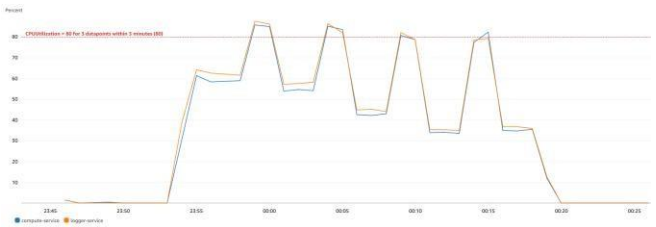


Fig : Bursty spikes

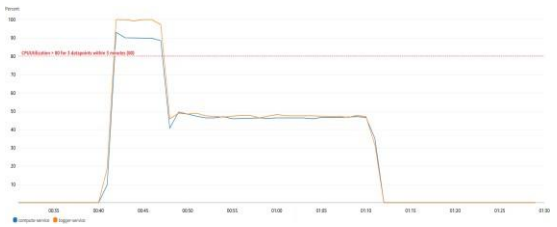


Fig : Soak

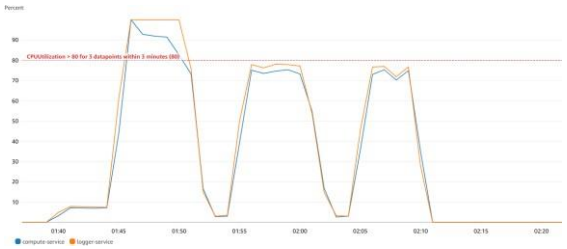


Fig: Mixed idle/peak

Figure 3: CPU Utilization of Round Robin Load Balancing at Different Workloads

7 Discussion

The study positions a lightweight, ECS-native approach to auto-scaling and request distribution against a literature that is dominated by Kubernetes controllers and learning-based schedulers. Prior work highlights both the pervasiveness of Kubernetes and limitations of its default HPA that is notably slow reaction in dynamic settings along with proposals to augment HPA with reactive and predictive logic such as Smart/ProSmart HPA to reduce over/under-provisioning. This work instead uses Amazon ECS with CloudWatch target tracking, a one-minute metric period, and conservative cool-downs. The justification is the simplicity of the mechanism to work on constrained environments but reproducible and cloud-native as well. The “lightweight, practical” niche articulated in the research motivation is therefore intentional and aligned with the gap identified in the related work section.

Compared to the resource managers grounded on learning, MSARS and DeepMCC, the current design exchanges the best under the complex objectives to deployability and understandability by the operator. MSARS and DeepMCC focus on rapidly optimizing adaptation and global placement choices, however, which presupposes non-trivial pipeline and orchestration complexity in the algorithm. The fact that the ECS target-tracking controller presented here only can be based on CPU telemetry and native alarms, it would be simpler to implement in small teams or in otherwise cost-constrained deployments. The validation runs confirm that such a simple controller can still meet stability goals, reaction occurs after a few one-minute evaluations above the target, capacity becomes ready one minute later, and error rates remain negligible across steady, step, bursty, soak, and mixed workloads. These observations empirically support the claim that “ECS-native scaling with simple CPU-based policies” can be viable where AI-based pipelines are impractical.

On the request-distribution side, experiments isolate the effect of load balancing policy at the API gateway. The placement strategies aim to equalize load at the cluster level, whereas this work shows that even at the service ingress, a deterministic round-robin load balancer materially improves CPU symmetry and average response time compared with uniform random selection. In the strategy identification phase, round-robin dominates random across baseline and all three CPU objectives, in the workload validation, the same policy retains stability under diverse traffic shapes. This is complementary to algorithmic placement work, since it demonstrates that gateway-level policies can have a significant effect on tail latency and the smoothness of scaling despite not making container placement decisions.

The results also echo the practical orientation of prior ECS case studies that emphasize low-cost, repeatable pipelines over sophisticated controllers. Where earlier ECS-based work demonstrates cost-effective orchestration for compute-intensive tasks, the present study extends that evidence by quantifying scaling and latency behaviour under controlled

traffic and by reporting timing measures that matter operational. These results combined indicate that a fully managed orchestration service with native autoscaling can be used to get reliable elasticity at scale without creating new operational overhead of a custom scheduler, so long as the pattern of workload is well understood, and the purpose of the scaling goal is selected carefully.

The limitations can be put into perspective with the aid of the comparative analysis. Learning-based schedulers and predictive HPAs have the potential in principle to respond quicker to short spikes and to optimize multi-objective SLOs. The bursty profile in this case reveals that the short surges which extend less than sixty seconds do not fulfill the sustained-threshold criterion and hence it will not activate further tasks. In situations where such bursts would break hard SLOs the literature partly indicates two directions which include predictive admission/ scale-ahead as apparently done by ProSmart-HPA-style controllers or placement/scheduling that minimizes queuing delay as it is done during the RL/GNN approaches. However, the 80% CPU target chosen to validate is a practical tradeoff that scales down the frequency of scaling yet maintains acceptable latency after the capacity is available, as is in line with the aim of the study to produce a simple and reproducible procedure adjusted to limited environments.

Compared with Kubernetes-centric and AI-assisted controllers, the ECS/CloudWatch approach evaluated here offers a lower-complexity path to elasticity that still achieves balanced utilization and low error rates. The primary empirical contribution is the demonstration that, for CPU-bound microservices behind an API gateway, round-robin load balancing plus ECS target tracking at 80% CPU delivers the best overall trade-off among latency, throughput, and stability across heterogeneous workloads, a result that complements, rather than contradicts, more sophisticated scheduling work by showing where a minimal mechanism suffices.

8 Conclusion and Future Work

Throughout the strategy identification and workload validation experiment, random load balancing was consistently outperformed by round-robin load balancing, both without the implementation of any dynamically scalable objectives, and in the autoscaled runs using different thresholds. In round-robin, the use of compute and logger were close to each other and latency was less in any of the cases and the error rates were also negligible. The result nature of using ECS target-tracking at an objective of 80% CPU provided a practical operating point, controller reaction followed a few minutes later with one minute evaluation of the target, capacity getting ready within a few minutes later, and steady, step load, bursty, soak as well as the mixed traffic supported constant performance. These results show that a light, ECS-native setting can guarantee reliability of the microservices elasticity when the gateway chooses the requests in a deterministic way.

Further work is needed to raise the two-task limit to measure tail-latency improvement under surges, generalize the controller beyond a CPU-only range into multi-metric goals that incorporate the request latency or queue size and consider faster response to brief bursts through shorter time profiles, step-scaling, or foresight scale-ahead. Replaying production-like traces, longer soaks with failure injection and comparison to learning-based or graph-aware schedulers should be used to establish where minimal per-target tracking is enough reaching the point where more complex control makes a difference.

9 References

Ahmad, H., Treude, C., Wagner, M. and Szabo, C., 2025. Towards resource-efficient reactive and proactive auto-scaling for microservice architectures. *Journal of Systems and Software* 213: 112772.

Lu, C., Zhou, J. and Zou, Q., 2025. An optimized approach for container deployment driven by a two-stage load balancing mechanism *PLOS ONE* 20(1): e0317039.

Biegel, G., 2024. Leveraging cloud compute and open source software to generate 3D models from drone photography. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLVIII-4-2024: 73–80.

Miao, J., Rajasekhar, D., Mishra, S., Nayak, S.K. and Yadav, R., 2024. A microservice-based smart agriculture system to detect animal intrusion at the edge. *Future Internet* 16(5): 196.

Saboor, A., Mahmood, A.K., Hassan, M.F., Shah, S.N.M., Hassan, F. and Siddiqui, M.A., 2021, July. Design pattern based distribution of microservices in cloud computing environment. In *2021 International Conference on Computer & Information Sciences (ICCOINS)* 1-6.

Hu, K., Wen, L., Xu, M. and Ye, K., 2024. MSARS: A Meta-Learning and Reinforcement Learning Framework for SLO Resource Allocation and Adaptive Scaling for Microservices, *Journal of Systems Architecture* 150: 103258.

Kambala, G., 2025 Integration Of Microservices And Cloud Computing: A Paradigm Shift In Enterprise Application Design.

Li, W., Li, X., Chen, L. and Wang, M., 2025. Microservice Workflow Scheduling with a Resource Configuration Model Under Deadline and Reliability Constraints. *Sensors* 25(4): 1042.

Arya, S., Chauhan, D., Anand, S. and Sharma, O., 2024, August. Beyond Monoliths: An In-Depth Analysis of Microservices Adoption in the Era of Kubernetes. In *2024 1st International Conference on Advanced Computing and Emerging Technologies (ACET)* 979–986.

Matos, G.H.M., Carvalho, M. and Macedo, D.F., 2024, November. Container-Based Microservice Scheduling Using Reinforcement Learning in Distributed Cloud Computing. In *2024 IEEE Latin-American Conference on Communications (LATINCOM)* 1-6.