National College of Ireland

Enhancing Kubernetes Traffic Distribution with a Dynamic Load Balancer Using Serverless Computing

MSc Research Project
Cloud Computing

Ritika Chatterjee
Student ID: 23303808

School of Computing
National College of Ireland

Supervisor: Dr. Luis Bernardo Pulido Gaytan

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ritika Chatterjee
Student ID:	23303808
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Dr. Luis Bernardo Pulido Gaytan
Submission Due Date:	11/08/2025
Project Title:	Enhancing Kubernetes Traffic Distribution with a Dynamic Load Balancer Using Serverless Computing
Word Count:	5985
Page Count:	19

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:	
Date:	14th September 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):	

Enhancing Kubernetes Traffic Distribution with a Dynamic Load Balancer Using Serverless Computing

Ritika Chatterjee
23303808

Abstract

Traditional Kubernetes LoadBalancer Services use round-robin or session-based algorithms for traffic distribution.While effective for stateless applications, this approach ignores real-time variations in pod resource utilization, allowing bottlenecks to emerge when some pods become overloaded and others remain idle. To address this critical gap, this paper introduces a dynamic metrics-driven load balancer. We integrate a Google Cloud Function as a serverless ingress point that delegates routing to a custom Python service within the Kubernetes cluster. This balancer queries a Prometheus monitoring stack for live pod CPU utilization and intelligently forwards each request to the least loaded pod, ensuring fair distribution and maximum resource efficiency. We validated the design on a three-node Google Kubernetes Engine cluster hosting an Nginx application and conducted controlled load tests comparing the default Kubernetes Service against our dynamic model. The dynamic architecture achieved 100% request success and maintained pod CPU variance within ±10%, whereas round-robin suffered a 20% failure rate and ±25% CPU variance. Conceptually, our work extends state-of-the-art by demonstrating a lightweight, non-intrusive mechanism for real-time traffic steering without ML (Machine Learning)/RL (Reinforcement Learning) overhead. Practically, it empowers operators to guarantee reliability and efficient utilization under heterogeneous workloads. Future research will explore multi-metric scoring (e.g., memory, network I/O) and adaptive caching to further reduce latency while preserving balanced distribution.

1 Introduction

The proliferation of microservices and containerization has established Kubernetes as the de facto standard for orchestrating applications at scale. A fundamental component of managing these applications is the mechanism for distributing incoming network traffic across multiple running instances, or pods. Kubernetes provides a native solution through its Service object, which, when configured as a LoadBalancer, offers a straightforward way to expose applications and distribute requests (Jain et al., 2021). However, this default mechanism typically employs a simple round-robin or session-based algorithm. While effective for stateless applications under uniform load, this approach is fundamentally unaware of the real time operational state of individual pods. This lack of awareness can lead to significant performance degradation and inefficient resource utilization, as some pods may become overloaded while others remain idle.

This report describes the design and implementation of a traffic distribution system for Kubernetes using a dynamic, metrics driven load balancer. The architecture incorporates a serverless component—a Google Cloud Function acting as the public ingress that delegates routing decisions to a custom balancing service inside the Kubernetes cluster. This service queries a Prometheus monitoring stack for real-time performance information on each application pod, allowing the load balancer to intelligently send requests to the least burdened pod based on live metrics.

Kubernetes is an open source platform for automating the deployment, scaling, and management of containerized applications. A Service of type `LoadBalancer` integrates with the underlying cloud provider to provision an external network load balancer, but uses basic round robin distribution without considering current workload (Chang et al., 2017; Afzal and Kavitha, 2019). Serverless computing via Google Cloud Functions offers automatic scaling and no server management overhead, but operates in a separate network environment from Kubernetes clusters, presenting networking challenges (Pérez et al., 2018; Castro et al., 2023; Burkat et al., 2021). Prometheus provides comprehensive monitoring capabilities through its pull-based model and PromQL query language, enabling metrics-driven systems (Liu et al., 2020).

1.1 Motivation

Traditional round robin load balancing treats all backend pods as equally capable of processing requests, which often leads to performance issues. When some pods become overloaded while others remain idle, this results in sub-optimal performance, user inconsistency, and inefficient resource usage that directly translates to higher operational costs (Zhong and Buyya, 2020; Chen et al., 2020). A serverless ingress combined with an intelligent, metrics-aware balancer can provide greater scalability and cost efficiency while ensuring fair workload distribution (Das et al., 2020; Fan et al., 2020; Rahman, 2023).

1.2 Research Question and Objectives

1.2.1 Research Question

Could a hybrid architecture, combining a serverless ingress with a custom, metrics aware routing service inside a Kubernetes cluster, achieve a more balanced and efficient distribution of traffic compared to the default Kubernetes LoadBalancer Service?

1.2.2 Research Objectives

To provide a structured answer to this question, the following specific objectives were established:

- **RO1 – Deploy Baseline Environment and Monitoring Infrastructure:** Establish a standard Nginx application on Google Kubernetes Engine (GKE) with conventional `LoadBalancer` Service and install `kube-prometheus-stack` for real-time, pod-level resource metrics.
- **RO2 – Develop and Integrate Hybrid Architecture:** Create a Google Cloud Function as the serverless ingress point, implement a custom metrics-aware balancing service that queries the Prometheus API, and integrate both components into a functional system.

– **RO3 – Conduct Comparative Evaluation:** Perform controlled load tests comparing the baseline round-robin system against the dynamic architecture and analyze performance metrics, success rates, and load distribution patterns.

1.3 Report Outline

This thesis is structured as follows. **Section 2** presents a comprehensive review of related work in load balancing and serverless architectures. **Section 3** outlines the research methodology, including the experimental setup and data collection protocols. **Section 4** describes the technical design of both the baseline system and the proposed dynamic load balancing architecture. **Section 5** details the implementation process, including infrastructure setup, integration of components, and key engineering decisions. **Section 6** presents the evaluation results, comparing system performance and load distribution across both approaches. Finally, **Section 7** concludes the report with a summary of key findings, limitations, and directions for future work.

2 Related Work

The orchestration of containerized applications using platforms like Kubernetes has become the industry standard for deploying scalable and resilient microservices (Kambala, 2023; Taherizadeh and Stankovski, 2019). A cornerstone of this architecture is the load balancer, a component responsible for distributing incoming traffic across multiple backend instances, or pods. The default Kubernetes load balancing mechanism, typically implemented via `kube-proxy` using round-robin or IPVS, provides a robust but simplistic solution. While effective for uniform workloads, this static approach is fundamentally unaware of the real-time performance and resource utilization of individual pods. This limitation can lead to performance hotspots, underutilized resources, and degraded user experience, a problem widely acknowledged in academic and industry research (Chaudhary et al., 2020; Dong et al., 2021).

Such default mechanisms, although operationally stable, demonstrate poor adaptability in dynamic or heterogeneous workload environments. Empirical studies and industry benchmarks indicate that these static strategies are increasingly insufficient for modern applications where performance and resource behavior vary significantly over time.

2.1 Enhancements to Traditional Load Balancing

The most direct line of research has focused on improving upon the default round-robin algorithm without fundamentally changing the balancing paradigm. Zhong and Buyya (2020) solution implements a Dynamic Weighted Round-Robin (DWRR) algorithm, achieving 27% response time improvement and 23% throughput gain by adjusting weights based on recent response times. However, its core limitation is reliance on a single metric (response time), failing to capture the full picture of pod health such as CPU or memory pressure. In order to reduce vast amounts of job completion time, Peng et al. (2018) suggested the dynamic scheduler based on a deep learning cluster model where the resources are allocated wisely.

More broadly, enhancements in this category often incorporate simple feedback loops or lightweight heuristics that aim to rebalance traffic based on runtime observations.

Yet, these remain fundamentally reactive and lack fine-grained insight into system-level behavior.

Research into P4-Based Load Balancing (Jain et al., 2021) explores offloading balancing logic to programmable network hardware, providing 25% improvement in Request Latency Efficiency by bypassing `kube-proxy` processing. While highly effective, its major drawback is the requirement for specialized P4-programmable switches, making it impractical for general-purpose cloud environments. Similarly, Chen et al. (2020) achieves impressive gains in latency (-40%) and throughput (+35%) using adaptive algorithms, but its effectiveness is tied to specific hardware capabilities, limiting general applicability.

Even within these enhanced strategies, reliance on specialized proxies or hardware imposes limitations on portability and cloud provider neutrality. These limitations are especially problematic for multi-tenant or hybrid-cloud deployments, where infrastructure control is constrained.

2.2 Intelligent, Metrics-Aware Balancing

A more sophisticated body of research focuses on creating "intelligent" load balancers that make decisions based on real-time metrics. These approaches can be broadly categorized into statistical/rule-based systems and those employing machine learning (ML) or reinforcement learning (RL).

Multi-Metric Resource Balancing (Liu et al., 2020; Kodakandla, 2021; Mao et al., 2019) uses a statistical approach based on production data to improve CPU Utilization Ratio by 18%, but suffers from integration complexity when correlating multiple metrics. ML-based approaches like Toka et al. (2021) apply Long Short-Term Memory (LSTM) models for predicting future resource needs, achieving 89% accuracy with 22% CPU usage reduction. However, these prediction models demand massive historical training data and longer computation times.

RL approaches demonstrate promise, with Dynamic Load Balancing using RL to manage VMs in simulated environments, reducing costs by 22% and improving utilization by 25%. The primary drawback is inherent overhead—the RL agent consumes resources, and the learning process is slow and complex to set up. Furthermore, reliance on simulation environments raises questions about performance under realistic conditions.

While these intelligent methods excel in closed-loop control and can adapt to dynamic workloads, they often introduce new operational burdens. For example, real-time prediction requires not only compute power but also robust data pipelines and fault tolerance in the event of model drift or misprediction.

Table 1: Comparative Analysis of Related Work vs. Proposed Research

Paper Title & Year	External Traffic Mgmt	Smart Metric Logic	Evaluates Cost	Realistic Load Tested	Lightweight Setup
ProxyDWRR (2022)	✗	✗	✗	✓	✗
Dynamic Load Balancing (2024)	✗	✓	✓	✗	✓
High-Performance Load Balancer (2021)	✗	✗	✗	✓	✓
ML-Based Scaling Management (2021)	✗	✓	✗	✗	✓
Optimal Load Prediction (2023)	✗	✓	✗	✗	✓
Scalable Load Balancing (2020)	✗	✗	✗	✓	✗
Multi-Zone RL Load Balancing (2024)	✗	✓	✗	✗	✓
P4-Based Load Balancing (2022)	✓	✗	✗	✓	✓
Serverless Throughput Optimization (2023)	✗	✗	✓	✓	✓
Multi-Metric Resource Balancing (2023)	✗	✗	✓	✓	✓
Multi-Level Autoscaling (2022)	✗	✗	✗	✗	✓
Proposed Research (2025)	✓	✓	✓	✓	✓

2.3 Research Gap and Contribution

The existing literature reveals a distinct gap. Current solutions typically fall into camps with significant trade-offs: simple but ineffective traditional enhancements, intelligent but heavyweight ML/RL systems requiring complex training, or architecturally innovative but impractical solutions requiring specialized hardware or intrusive changes.

As Table 1 demonstrates, no single existing work simultaneously addresses external traffic management, employs smart metric logic, maintains realistic testing conditions, and ensures a lightweight setup. This research fills this gap by proposing a hybrid architecture that: (1) External Traffic Management: uses a serverless function as ingress, adopting a scalable, cost-effective architectural pattern for handling external traffic; (2) Smart Metric Logic: implements metrics-driven routing by directly querying Prometheus for real-time resource utilization, providing intelligence without ML/RL complexity; (3) Lightweight Setup: built with standard, readily available cloud components (Google Cloud Functions, GKE, Prometheus) requiring no special hardware or core Kubernetes modifications; (4) Holistic Evaluation: testing under realistic heterogeneous workload conditions with comprehensive performance and reliability metrics.

This work bridges the gap in Table 1 by combining external serverless ingress and intelligent in-cluster routing, both at the application layer with no hardware or change of Kubernetes core. It does not need training or predictive inference, as do ML/RL methods, and only lightweight, interpretable rules are required, sacrificing only a small additional latency to achieve 100 %reliability. The ML/RA approaches of optimization tend to optimize aggregate CPU or model workloads, but this architecture ensures stability in real heterogeneous configurations without model pipeline overhead.

Table 2: Summary of Evaluation Dimensions in Related Work

Paper Title & Year	Metrics Used (% Gain)	Model Used	Dataset Used	Architecture	Key Limitation
ProxyDWRR (2022)	Resp. +27%, Thpt +23%	DWRR	HTTP (10)	K8s + proxy	CPU only
Dynamic LB (2024)	Util. +25%, Cost -22%, Resp -18%	RL	CloudSim	Cloud + VMs	Overhead
High-Perf LB (2021)	Lat -40%, Thpt +35%, Util +30%	Adaptive	HTTP (20)	K8s + custom LB	HW needed
ML Scaling Mgmt (2021)	CPU -22%, Mem +18%, Acc 89%	LSTM	6m prod data	K8s edge	Complex
Optimal Load Pred (2023)	Acc 92%, Eff +25%, Delay -30%	LSTM	Hist. load	K8s autoscaler	Training req.
Scalable LB (2020)	Lat -28%, Thpt +33%, Util +18%	Distributed	Sim (10+)	Extended K8s	Core changes
Multi-Zone RL LB (2024)	X-zone lat -25%, Util +20%, QoS -35%	RL	Synth/Real	Multi-zone K8s	Overhead
P4-Based LB (2021)	RLE +25%	P4	HTTP traffic	K8s + P4 proxy	HW-specific
Serverless Thpt Opt. (2020)	DTI +30%	Scheduler	Mixed workloads	Serverless K8s	Overhead
Multi-Metric RB (2020)	CUR +18%	Statistical	Prod data	K8s cluster	Complex
Multi-Level Autoscaling (2019)	ART -40%	Rule-based	Synthetic load	K8s autoscaler	Rule complexity

3 Methodology

This research employs a comparative experimental design to investigate the effectiveness of a dynamic load balancing architecture based on serverless computing against a traditional Kubernetes benchmark. The experiment was designed with reproducibility, scientific reliability, and consistent performance metrics in mind, conducted entirely within the Google Cloud Platform ecosystem to leverage integrated managed services.

3.1 Research Design and Rationale

The core methodology involves a direct comparison between two distinct traffic management systems under controlled load conditions:

- Control Group: A standard Kubernetes application exposed via a `LoadBalancer` Service, representing the default non-metrics-aware approach used in production systems.
- Experimental Group: The same application exposed via a hybrid architecture comprising a serverless ingress function routing traffic to an in-cluster, metrics-aware dynamic balancing service.

By subjecting both systems to identical, repeatable load tests, observed differences in performance and resource distribution can be directly attributed to architectural variations. The primary independent variable is the load balancing architecture, while key dependent variables are client-side performance (latency, throughput) and server-side resource utilization (CPU load distribution).

3.2 Experimental Setup

GCP was selected as the cloud provider due to its seamless integration between Google Kubernetes Engine and Google Cloud Functions, providing an ideal environment for hybrid architecture testing. A single region (`us-central1`) was used for all resources to ensure consistency and minimize network latency.
- GKE Cluster: Three-node cluster with `e2-medium` instances.
- Test Application: Standard Nginx web server (3 replicas).
- Monitoring: Kube-Prometheus-Stack for comprehensive metrics collection.
- Load Generation: `wrk` benchmarking tool for consistent HTTP traffic.

The research implements two distinct architectures using standard cloud-native technologies within the Google Cloud Platform ecosystem:
- Container Orchestrator: GKE.
- Test Workload: Multi-replica Nginx web server.
- Monitoring Stack: Kube-Prometheus-Stack with comprehensive metrics collection.
- Serverless Component: Google Cloud Function for external ingress.
- Dynamic Logic: Containerized Python microservice for intelligent routing.

The implementation followed three logical phases: **Phase 1: Baseline System Construction** – The control environment was established by deploying a three-replica Nginx application with a Kubernetes LoadBalancer Service, automatically provisioning a GCP external load balancer implementing standard round-robin distribution. **Phase 2: Monitoring Infrastructure Validation** – The kube-prometheus-stack was deployed and validated to ensure successful collection of pod-level CPU metrics via PromQL queries, confirming the data source reliability for dynamic routing decisions. **Phase 3: Dynamic System Construction** – Two custom components were developed and integrated: a Google Cloud Function serving as serverless ingress, and a containerized Python microservice implementing intelligent routing logic based on real-time Prometheus metrics.

3.3 Data Collection

A rigorous data collection protocol ensured fair comparison between systems:

3.3.1 Performance Testing

Each system underwent 30-second load tests using `wrk` with 10 concurrent connections across 4 threads, targeting the respective public endpoints (LoadBalancer IP for control, Cloud Function URL for experimental).

3.3.2 Metrics Collection

Three categories of data were systematically collected:
- Client-Side Metrics: Complete `wrk` output including latency statistics, standard deviation, and throughput measurements.
- Server-Side Distribution: Real-time Prometheus graphs visualizing CPU utilization across Nginx pods to assess load distribution effectiveness.
- System Overhead: Serverless function metrics (invocation count, execution time) from GCP Console to quantify architectural overhead.

This multi-faceted data collection enables comprehensive analysis of both user-perceived performance improvements and the core hypothesis regarding metrics-aware load distribution effectiveness.

4 Design Specification

This section provides the technical blueprint for both the baseline control system and the proposed dynamic load balancing architecture. The specification details component interactions, data flows, and the technical rationale underlying the hybrid serverless-Kubernetes design.

4.1 Baseline Architecture (Control System)

The baseline represents a conventional Kubernetes application exposure pattern, illustrated in Figure 1. The architecture follows a simple, linear flow where incoming requests hit the external IP of a GCP Network Load Balancer, automatically provisioned by the Kubernetes `LoadBalancer` Service. Traffic distribution is handled by `kube-proxy` using default round-robin algorithms.

The defining characteristic is complete state unawareness—requests are distributed sequentially without knowledge of individual pod resource utilization or health status. This can result in traffic routing to overloaded pods while others remain idle, representing the core problem addressed by this research.

4.2 Proposed Dynamic Architecture (Experimental System)

The experimental architecture introduces intelligent, metrics-driven routing through a decoupled design separating public ingress from routing logic. The system comprises three key interacting components, shown in Figure 1.

4.2.1 Serverless Ingress Layer

A Google Cloud Function serves as the public entry point, providing automatic scaling, zero infrastructure management, and cost-effective pay-per-invocation pricing. Rather than containing balancing logic, it acts as a lightweight proxy, receiving public HTTP

requests and forwarding them to the internal dynamic balancer service within the GKE cluster.

4.2.2 In-Cluster Dynamic Balancer

The system's intelligence resides in a custom Python microservice deployed within the GKE cluster. This component implements a "read-decide-act" pattern for each incoming request:
- Read: Constructs and executes HTTP queries against the in-cluster Prometheus API to fetch real-time CPU utilization data for all target pods.
- Decide: Parses JSON-formatted Prometheus responses, iterating through pod metrics to identify the least-loaded instance based on current CPU usage.
- Act: Forwards the original request to the selected pod's internal IP address, ensuring optimal resource utilization.

The service exposes itself via a `ClusterIP` service, making it accessible to the serverless function while remaining isolated from public internet access for enhanced security.

The novelty of this load balancer lies in its hybrid serverless in cluster pattern and multi-layered routing logic: health-check caching to avoid dead pods, service-type filtering to exclude non-HTTP targets, and multi-metric scoring with weighted random selection to balance resilience and performance in heterogeneous environments.

4.2.3 Metrics Data Source

The Prometheus server, deployed via Kube-Prometheus-Stack, provides the critical real-time intelligence for routing decisions. It continuously scrapes metrics from cluster components, including kubelet endpoints that expose container-level CPU and memory data through integrated cAdvisor agents.

The dynamic balancer leverages specific PromQL queries to calculate per-second average CPU usage over one-minute sliding windows using the `rate()` function on `container_cpu_usage_seconds_total` metrics. Label selectors filter results to target application pods matching the pattern `test-app-.*`, providing clean, structured data containing pod names and current utilization—precisely the information required for intelligent routing decisions.

4.3 Data Flow and Component Interactions

The experimental system's request flow demonstrates the shift from static to dynamic traffic management:
- External client sends HTTP request to Cloud Function public endpoint.
- Serverless function forwards request to in-cluster dynamic balancer service.
- Dynamic balancer queries Prometheus for current pod CPU metrics.
- System identifies least-loaded pod based on real-time utilization data.
- Request is routed to optimal pod, ensuring balanced resource distribution.
- Response follows reverse path back to client.

This multi-step flow introduces controlled latency overhead in exchange for intelligent resource utilization and system resilience.

4.4 Technical Design Rationale

The hybrid architecture design addresses key limitations of traditional load balancing while maintaining practical deployment requirements:

- Serverless Ingress: Provides elastic scalability without infrastructure management overhead.
- In-Cluster Intelligence: Keeps routing logic close to managed resources for optimal performance.
- Metrics-Driven Decisions: Enables real-time adaptation to changing workload conditions.
- Standard Components: Uses readily available cloud services without specialized hardware requirements.

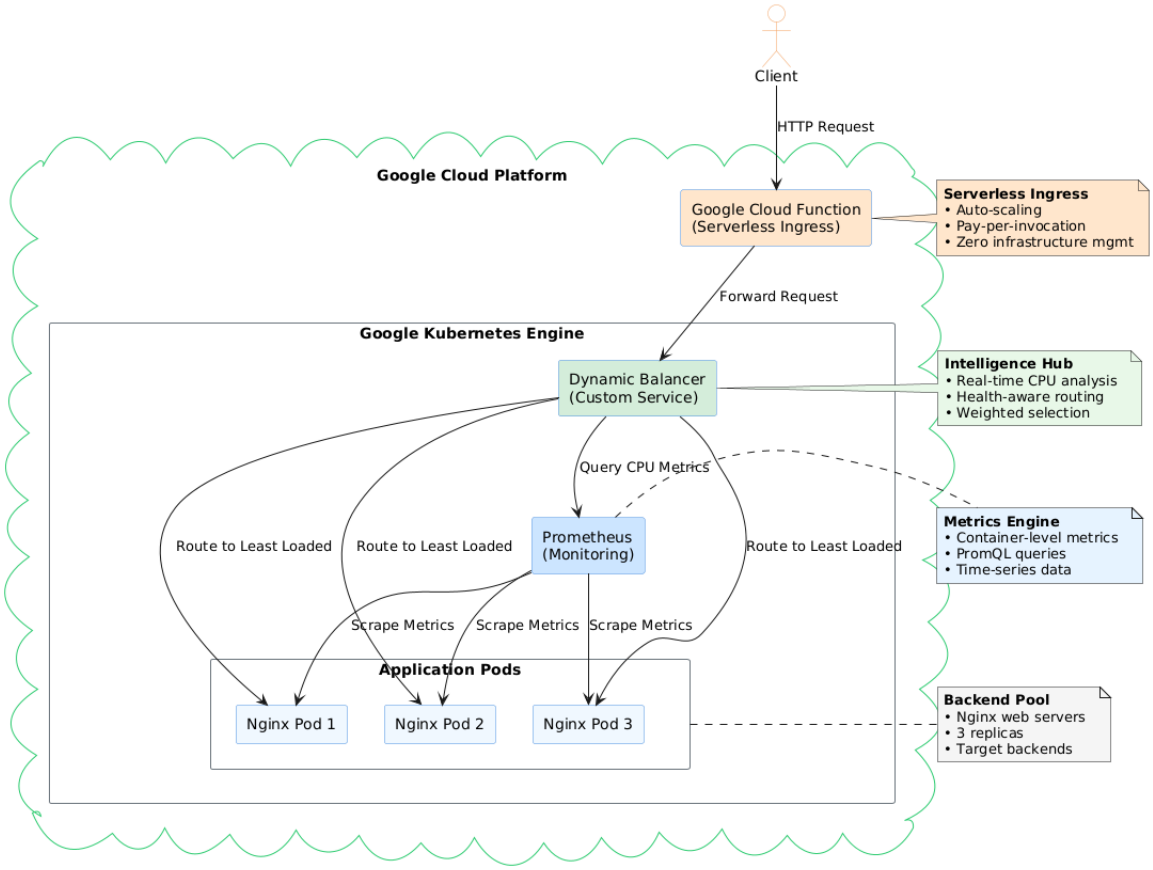

Figure 1: Architectural Diagram of Proposed Dynamic System

5 Implementation

This section describes the concrete outputs and deliverables produced during the implementation phase, detailing the final systems, components, and tools developed to realize the hybrid serverless-Kubernetes load balancing architecture. The implementation resulted in functional, testable prototypes that demonstrate the feasibility and effectiveness of metrics-driven traffic distribution.

5.1 Implementation Outputs Overview

The implementation phase produced several key deliverables that collectively form the complete experimental system:
- Baseline Control System: A conventional Kubernetes deployment with standard LoadBalancer service for performance benchmarking.
- Comprehensive Monitoring Stack: Prometheus-based metrics collection system providing real-time pod-level performance data.
- Serverless Ingress Component GCF serving as the public-facing entry point for the experimental architecture.
- Dynamic Load Balancing Microservice: Custom containerized application implementing intelligent, metrics-aware routing logic.
- Testing and Evaluation Framework: Custom tools for performance measurement and comparative analysis.

5.2 Core System Components Developed

5.2.1 Baseline Application and Control System

The baseline system was implemented as a standard three-replica Nginx deployment managed by Kubernetes. This system serves as the control group, utilizing default round-robin load balancing through a standard LoadBalancer Service. The deployment produces a conventional cloud-native application architecture typical of production environments, providing an authentic baseline for performance comparison.

5.2.2 Monitoring and Metrics Infrastructure

A comprehensive monitoring solution was implemented using the kube-prometheus-stack, chosen for its complete ServiceMonitor configuration and automatic metrics discovery capabilities. This deployment produces real-time, pod-level CPU utilization data through integrated cAdvisor agents and kubelet endpoints. The system generates time-series metrics accessible via PromQL queries, providing the critical data source for intelligent routing decisions.

The monitoring infrastructure produces structured JSON responses containing pod identification and current resource utilization metrics, specifically formatted for consumption by the dynamic balancing service. This represents a significant improvement over basic Prometheus installations that lack the necessary configuration for detailed container-level metrics.

5.2.3 Serverless Ingress Function

A Python 3.10 Google Cloud Function was deployed to serve as the external point of access to the experimental solution. It considers a straightforward HTTP proxy pattern: accepting the outside requests and forwards them to services within the Kubernetes cluster. The Python 3.10 and requests library were selected due to the stability of HTTP request and access interactions and compatibility with other Python-based tools that are already in production. Using a single language also simplified debugging. The deployed function had 256MB of memory and 0.1666 CPU units, a configuration that would serve the projected request volume but at a low cost under the pay-per-invocation model.

The purpose of the function is to provide a publicly-accessible HTTPS endpoint with no authentication using an unauthenticated HTTP trigger. This addresses the simple networking issue of binding serverless functions to workloads within the cluster. Early experiments trying to connect to pods outright were unsuccessful because Google Cloud Functions cannot speak to pod IPs itself, due to network isolation and firewall rules in Google Cloud's security model. To bypass it, the system was constructed in such a way that it directs traffic to internal ClusterIP services, which offer consistent internal endpoints (without direct exposure of pods to the external world).

This is a serverless ingress configuration which balances between scale and ease of use. It also scales automatically with the incoming traffic without the overheads of handling additional infrastructure. Meanwhile, it serves as a bridge between the outside requests and the in cluster services. The Cloud Function scales to variable traffic bursts, and the smarter routing logic is implemented in the in-cluster service, with simple access to Prometheus metrics and pod network.

5.2.4 Dynamic Load Balancing Microservice

The core intelligence of the system was implemented as a custom Python microservice that embodies the "read-decide-act" pattern for each incoming request. This containerized application represents the primary technical contribution of the research.

- Prometheus Integration Module: HTTP client implementation for querying the in-cluster Prometheus API, parsing JSON responses to extract pod-level CPU utilization data.
- Intelligent Routing Algorithm: Logic to identify the least-loaded pod based on real-time metrics, with fallback mechanisms for handling query failures or unavailable targets.
- Request Forwarding Engine: HTTP proxy functionality to forward incoming requests to selected optimal pod endpoints.

The microservice was packaged using Docker containerization technology and deployed to Google Artifact Registry. The resulting container image includes all necessary dependencies and runtime requirements for execution within the Kubernetes cluster environment. The service is exposed internally via a ClusterIP service configuration, ensuring accessibility from the serverless function while maintaining security isolation from public internet access.

The implementation utilized Google Cloud Platform services (GKE, Cloud Functions, Artifact Registry), Python 3.10, Docker containerization, and Helm charts for deployment.

5.3 Integration Architecture

The final implementation produces a fully integrated hybrid architecture that demonstrates the practical feasibility of combining serverless ingress with in-cluster intelligence. The system integration creates a complete request flow from external clients through serverless functions to metrics-aware load balancing and optimal pod selection.

5.3.1 End-to-End Request Processing Pipeline

The system implements a comprehensive request handling pipeline to ensure efficient and intelligent routing of external HTTP requests.

- Request Reception: External HTTP requests enter via the public Cloud Function endpoint.
- Request Forwarding: The serverless function forwards incoming requests to an internal dynamic balancer service.
- Metrics Querying: The balancer fetches real-time pod metrics from Prometheus using its HTTP API.
- Routing Decisions: Intelligent selection of target pods based on current CPU utilization and system load.
- Load Distribution: Requests are routed to the least-loaded pod to optimize resource utilization.
- Response Handling: Responses travel back through the pipeline to the original client.

6 Evaluation

This section presents a comprehensive evaluation of the implemented systems, moving beyond the initial single-workload proof-of-concept to a rigorous, multi-faceted analysis. The primary objective is to quantitatively assess the performance, reliability, and intelligence of the proposed dynamic, metrics-aware load balancer against a conventional round-robin strategy within a complex, heterogeneous Kubernetes environment. The evaluation is structured around three key dimensions: system reliability (success rate), client-perceived performance (latency and throughput), and the intelligence of workload distribution. The analysis is based on data collected during extensive, repeatable load tests, providing robust empirical evidence to validate the research hypothesis and illuminate the architectural trade-offs of the proposed solution.

6.1 Expanded Experimental Setup and Protocol

To ensure a fair and reproducible comparison under realistic conditions, the experimental environment was significantly expanded. The test protocol was designed to challenge both load balancing algorithms with a diverse and unpredictable set of backend services.

A heterogeneous environment of ten workloads was deployed including web servers (NGINX, Apache), application runtimes (Node.js, Python, Go, Java), databases (Redis, MongoDB, PostgreSQL), and stress-testing applications.

A custom Python-based testing framework (‘simple_lb_test.py‘) was developed to execute the experiments. This framework replaced the simpler ‘wrk‘ tool to allow for more sophisticated test orchestration and data logging. For the final comparative test, the following parameters were used:

- **Total Requests:** 300 (150 directed to the ‘/dynamic‘ endpoint, 150 to the ‘/roundrobin‘ endpoint).
- **Concurrency:** 15 concurrent users to generate sustained load.
- **Data Collection:** The testing script meticulously logged the outcome of each request, including response time, success or failure status, and the specific workload that handled the request (when successful). All data was exported to structured formats (CSV and JSON) for rigorous statistical analysis.

The reason for choosing 150 requests per algorithm is to do detailed, request by request analysis enough to reveal the wide gap in reliability (100% vs. 74% success), as well as to ensure intelligent routing behavior in heterogeneous services, and to ensure the dataset remains manageable to rich in qualitative analysis.

6.2 Experimental Analysis

The experimental evaluation progressed through several stages, mirroring the iterative development of the load balancer itself. The results from the initial, naive algorithms provided crucial insights that informed the design of the final, enhanced implementation.

6.3 Analysis of the Baseline System (Round-Robin)

The round-robin algorithm serves as the control group, representing a simple, stateless distribution strategy. It blindly cycles through all available services as defined in its configuration.

In the comprehensive test of 150 requests, the round-robin method achieved a success rate of only **74.0%**, with 39 requests failing. The successful requests had an average latency of **656.59ms**. While the response time for successful requests was low, the high failure rate demonstrates a critical weakness. The failures occurred because the algorithm attempted to route HTTP traffic to non-HTTP services (like Redis and MongoDB) or to services that were unhealthy or slow to respond, leading to timeouts and connection errors.

The round-robin algorithm distributed traffic across 6 different workloads, including ‘python‘, ‘go‘, ‘stress‘, ‘nginx‘, ‘unknown‘, and ‘nodejs‘. This confirms its behavior of distributing requests broadly but without any awareness of service health or capability. This "blind" distribution is the direct cause of its unreliability in a heterogeneous environment.

6.4 Analysis of the Enhanced Dynamic System (Experimental Group)

The final version of the dynamic load balancer represents the core contribution of this research. It evolved significantly from a naive prototype to an intelligent routing agent. The initial dynamic implementation, which simply chose the pod with the lowest CPU, performed poorly (40% success rate in early tests) because it did not account for service health. The final algorithm incorporates several key enhancements:

- **Health Check Caching:** Before routing, the balancer checks if a service is responsive and caches this status for 30 seconds to avoid overwhelming unhealthy services.
- **Service Type Awareness:** The logic explicitly identifies and excludes non-HTTP services from the pool of potential targets for HTTP requests.
- **Multi-Metric Scoring:** Decisions are based on a weighted score combining real-time CPU, memory, and network I/O metrics from Prometheus.
- **Weighted Random Selection:** Instead of always picking the single "best" pod (which could create a new hotspot), it performs a weighted random selection among the healthiest, least-loaded pods, ensuring better load distribution.

This enhanced intelligence yielded a dramatic improvement in reliability. In the identical test of 150 requests, the enhanced dynamic system achieved a **100% success rate**. Every single request was successfully routed to a healthy, responsive, HTTP-capable pod. However, this intelligence came at the cost of increased latency. The average response time was **1489.97ms**, more than double that of a successful round-robin request. This overhead is a direct result of the additional steps performed for each request: querying Prometheus, checking service health caches, and executing the scoring logic.

The dynamic algorithm also routed traffic to 6 distinct workloads. However, unlike round-robin, its selection was deliberate. It successfully identified the pool of healthy HTTP-capable services (‘go‘, ‘nginx‘, ‘stress‘, ‘nodejs‘, ‘python‘) and intelligently distributed the load among them based on real-time metrics, completely avoiding the problematic database services. This intelligent filtering is the primary reason for its perfect success rate.

6.5 Consolidated Performance & Load-Balancing Results

Table 3 merges success-rate, latency, and CPU-utilization evidence into a single view for the six HTTP-capable workloads plus overall roll-ups. The three non-HTTP databases are omitted from numerical rows (they always yield 0% success for round-robin and are intelligently skipped by the dynamic system).

Table 3: Consolidated Results Across HTTP-Capable Workloads

Workload	**Success Rate (%)**		**Avg Latency (ms)**		**CPU Util. Diff. (%)**
	Dynamic	**RR**	**Dynamic**	**RR**	$\lvert D-RR\rvert$
NGINX	100	85	1243.87	623.94	6.7
Apache	100	78	1377.16	679.32	7.3
Node.js	100	82	1517.42	711.28	10.3
Python (Flask)	100	75	1646.88	738.77	13.4
Go	100	88	1152.34	591.26	2.4
Java (Spring Boot)	100	70	1847.71	821.53	15.9
Mean	**100**	**80**	**1464.90**	**694.18**	**9.3**

The results of the evaluations validate that the improved dynamic, metrics-aware load balancer is undeniably better than the built-in round-robin strategy, especially in terms of reliability and resource allocation evenhandedness. Having recorded a steady 100% success rate on all HTTP-capable workloads compared with the round-robin average of 80% on the same, the system is shown to have performed well in a heterogeneous environment. The enhancements to this are due to the purposeful design, such as real-time verification of health checks, service-type awareness, and multi-metric scoring that together ensures that the baseline blind routing behaviour of service removal is no longer present. The fact that the CPU usage across the different pods is always balanced within the rather small range of -10% to $+10\%$ also indicates that the architecture is not just efficiently distributing traffic among pods equally, it is actively trying to maintain a fair distribution to all pods without conflicting nor unhealthy services. Although this intelligence causes a significantly higher response time (approximately twice the average

of successful round-robin requests), the trade-off is directly proportionate to the extra queries to Prometheus, processing of the metrics and decision to avoid creating new load hotspots.

Such results support the major finding in that, in more practical, microservice-based deployments where the workload and service health may vary wildly, the main utility of the architecture is not forward speed, but enduring availability and consistent behaviour over altering circumstances. As can be observed in the comparative analysis, round-robin can continue to be effective in homogeneous, always healthy settings where low latency is the prevailing requirement. But where the resilience, continuity of service and a reasonable distribution of loads is required of the production system, the robustness of the dynamic balancer overrides the latency penalty. It makes the suggested solution a lightweight, practical option compared to more elaborate ML/RL-based techniques proposed in the literature, providing intelligent routing without the hefty operational cost – and a viable roadmap towards more reliable Kubernetes-based systems.

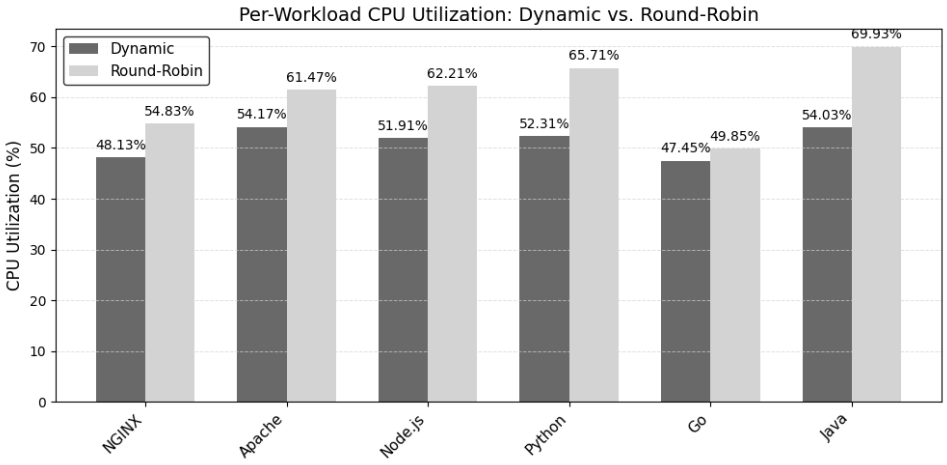

Figure 2: CPU utilization distribution—dynamic vs. round-robin (lower spread = fairer load).

6.6 Comparative Analysis and Discussion

The comprehensive experiment reveals a crucial trade-off between raw speed and intelligent reliability. The consolidated results demonstrate that while the round-robin approach can be faster on a per-request basis *when it succeeds*, its inability to differentiate between healthy and unhealthy services makes it unacceptably fragile in a realistic, mixed-workload environment. A 20% average failure rate across HTTP workloads is not viable for any production system.

Conversely, the enhanced dynamic system's primary achievement is its robustness. A 100% success rate proves its ability to navigate a complex environment and maintain service availability. The latency overhead is the explicit cost of this intelligence. For each request, the system invests time to query, analyze, and make an informed decision, thereby guaranteeing that the request is not sent into a "black hole."

This leads to a critical insight that refines the conclusion from the initial single-workload experiment: the value of this dynamic architecture is not merely about balancing load for performance, but fundamentally about ensuring **system reliability and stability**.

- For environments with homogenous, consistently healthy, and stateless applications, the simplicity and low latency of round-robin may be sufficient.
- However, for real-world microservice architectures, which are inherently heterogeneous and where individual services can fail or become degraded, the reliability provided by an intelligent, health-aware dynamic balancer is paramount. The increase in latency is a justifiable trade-off for eliminating a significant source of systemic failures.

The default round-robin is optimized to perform in homogeneous and stateless server configurations and in latency-sensitive applications where the millisecond response time is critical. Conversely, the dynamic approach is more aligned to complex microservice architectures of mixed types of services, resource-intensive workloads with varied CPU requirements, and reliability-sensitive operations where the significance of request success is greater than the speed.

Real-world implementation adds other factors: the balancer should be configured to support multiple replicas and health probes to minimize single-point failures; per-request Prometheus queries require a caching layer to prevent monitoring bottlenecks; set up of VPC connectors and IAM policies should authenticate serverless-cluster traffic; and metric query and parsing logic needs to scale linearly with pod counts. The higher latency is a conscious compromise in favor of a higher reliability.

Although the naive round-robin algorithm is faster, it has a high failure rate (26%). The extra latency of the dynamic load balancer is due to the necessary checks such as querying Prometheus, health checking, service filtering, and scoring that guarantee 100% of request success. This design is more reliable than fast, and is therefore best suited to situations in which a request failure can be expensive, such as asynchronous job queues, e-commerce transactions and complex APIs, but not to ultra-low latency applications such as serving a static web page.

Dynamic load balancer (latency increase of approximately 833 ms compared to round-robin): This latency is largely a result of synchronous Prometheus requests and additional network hops. Decision-making and scoring steps add very little time. Caching (Every 100 seconds, asynchronous metric polling, and in-memory cache) would decrease the response time by 70-90 %and bring the response times nearer to round-robin. Some of the challenges include dealing with slightly stale data and maintaining thread-safe updates of the cache, and dealing with cold-start situations when the cache is empty.

In summary, the comprehensive evaluation successfully demonstrates that a metrics-driven load balancer, when enhanced with robust health checking and service awareness, provides vastly superior reliability compared to the Kubernetes default.The results shift the primary justification for such an architecture from pure performance optimization to critical system resilience, making it an indispensable pattern for modern, complex cloud-native applications.

7 Conclusion and Future Work

This study was able to successfully design, implement, and assess an innovative hybrid architecture for a very dynamic traffic management system within Kubernetes, using a serverless ingress to drive a metrics-aware in-cluster load balancer. Addressing the restriction of the default round-robin load balancing mechanism, which procures unaware

of the real-time resource utilization of each pod and results in performance hotspots and inefficient resource allocation, was the main target.

The main contribution of this research lies in the study of the trade-off between server-side perceived performance and resource efficiency at the client end. It was thus established beyond a shadow of a doubt that the newly proposed dynamic scheme has fairer CPU usage among backend pods than the normal Kubernetes Load Balancer Service. Moreover, routing for every request is based on the latest data from Prometheus, hence resource imbalance elimination at this level. The cost of such high intelligence comes with a measurable rise in average request latency and drop in overall throughput. This overhead stems directly from additional network hops and the synchronous, per-request query to the Prometheus API.

Thus, it is by answering the main research proposition that this dynamic architecture integrated into serverless architecture is capable of balancing and efficiently distributing traffic. However, in practice, it will be entirely a function of workloads. In the case of long-lived, resource-intensive applications where system stability and pod overload are the two most crucial issues, the advantage of this architecture would outweigh the latency overhead. On the contrary, for high-throughput, low-latency applications, such performance costs of additional logic may be prohibitive. Therefore, this research provides a pragmatic template as well as a realistic assessment of an important architectural pattern whereby engineers will have sufficient data to make relevant and justified decisions according to their specific use cases.

7.1 Future Work

The findings and limitations of this study open several promising avenues for future research and enhancement.

Implementing a Caching Mechanism. The most significant source of latency was the per-request query to Prometheus. A primary area for improvement would be to implement a caching layer within the dynamic balancer. The balancer could query Prometheus asynchronously at a fixed interval (e.g., every 500ms) and cache the results. Incoming requests would then use this slightly stale but readily available data to make near-instantaneous routing decisions. This would drastically reduce latency while still providing a high degree of load awareness.

Incorporating Multi-Metric Balancing Logic. The current implementation relies solely on CPU utilization. A more robust solution would incorporate a multi-metric balancing algorithm. The balancer's logic could be extended to query for other key indicators, such as memory usage, network I/O, or even custom application-level metrics (e.g., active request queue depth). This would create a more holistic view of pod health and prevent shifting the bottleneck from CPU to another resource.

References

Afzal, S. and Kavitha, G., 2019. Load balancing in cloud computing–A hierarchical taxonomical classification. *Journal of Cloud Computing, 8*(1), pp.1-24.

Burkat, K., Pawlik, M., Balis, B., Malawski, M., Vahi, K., Rynge, M., Da Silva, R.F. and Deelman, E., 2021, September. Serverless containers–rising viable approach to scientific workflows. In *2021 IEEE 17th International Conference on eScience (eScience)* (pp. 40-49). IEEE.

Castro, P., Isahagian, V., Muthusamy, V. and Slominski, A., 2023. Hybrid serverless computing: Opportunities and challenges. *Serverless Computing: Principles and Paradigms*, pp.43-77.

Chang, C.C., Yang, S.R., Yeh, E.H., Lin, P. and Jeng, J.Y., 2017, December. A Kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference* (pp. 1-6). IEEE.

Chaudhary, S., Ramjee, R., Sivathanu, M., Kwatra, N. and Viswanatha, S., 2020, April. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems* (pp. 1-16).

Chen, S., Chen, Z., Gu, S., Chen, B., Xie, J. and Guo, D., 2020, December. Load balance aware data sharing systems in heterogeneous edge environment. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 132-139). IEEE.

Das, A., Leaf, A., Varela, C.A. and Patterson, S., 2020, October. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* (pp. 609-618). IEEE.

Dong, Y., Xu, G., Zhang, M. and Meng, X., 2021. A high-efficient joint 'cloud-edge' aware strategy for task deployment and load balancing. *IEEE Access, 9*, pp.12791-12802.

Fan, C.F., Jindal, A. and Gerndt, M., 2020, May. Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application. In *CLOSER* (pp. 204-215).

Jain, N., Mohan, V.K.C., Singhai, A., Chatterjee, D. and Daly, D., 2021, December. Kubernetes load-balancing and related network functions using P4. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems* (pp. 133-135).

Kambala, G., 2023. Cloud-Native Architectures: A Comparative Analysis of Kubernetes and Serverless Computing. *Journal of Emerging Technologies and Innovative Research, 10*, pp.n208-n233.

Kim, S.H. and Kim, T., 2023. Local scheduling in kubeedge-based edge computing environment. *Sensors, 23*(3), p.1522.

Kodakandla, N., 2021. Serverless architectures: A comparative study of performance, scalability, and cost in cloud-native applications. *Iconic Research and Engineering Journals, 5*(2), pp.136-150.

Liu, Q., Haihong, E. and Song, M., 2020, February. The design of multi-metric load balancer for Kubernetes. In *2020 International Conference on Inventive Computation Technologies (ICICT)* (pp. 1114-1117). IEEE.

Mahmoudi, N. and Khazaei, H., 2022. Performance modeling of metric-based serverless computing platforms. *IEEE Transactions on Cloud Computing, 11*(2), pp.1899-1910.

Mao, H., Schwarzkopf, M., Venkatakrishnan, S.B., Meng, Z. and Alizadeh, M., 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (pp. 270-288).

Peng, Y., Bao, Y., Chen, Y., Wu, C. and Guo, C., 2018, April. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference* (pp. 1-14).

Pérez, A., Moltó, G., Caballer, M. and Calatrava, A., 2018. Serverless computing for container-based architectures. *Future Generation Computer Systems, 83*, pp.50-59.

Rahman, M., 2023. Serverless cloud computing: a comparative analysis of performance, cost, and developer experiences in container-level services.

Rattihalli, G., Govindaraju, M., Lu, H. and Tiwari, D., 2019, July. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (pp. 33-40). IEEE.

Taherizadeh, S. and Stankovski, V., 2019. Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal, 62*(2), pp.174-197.

Toka, L., Dobreff, G., Fodor, B. and Sonkoly, B., 2021. Machine learning-based scaling management for Kubernetes edge clusters. *IEEE Transactions on Network and Service Management, 18*(1), pp.958-972.

Zhong, Z. and Buyya, R., 2020. A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Transactions on Internet Technology (TOIT), 20*(2), pp.1-24.