An Empirical Study on Edge-to-Cloud Continuum for Smart Applications: Performance, Design Patterns, and Key Factors

Norman Chen*§, Adel N. Toosi*§, Bahman Javadi†, Daghash Alqahtani*,
Mohammad S. Aslanpour*, Minxian Xu‡§
*Department of Software Systems and Cybersecurity, Monash University, Clayton, Australia
{norman.chen, adel.n.toosi, daghash.alqahtani, mohammad.aslanpour}@monash.edu
†School of Computer, Data and Mathematical Sciences, Western Sydney University,
b.javadi@westernsydney.edu.au
‡Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
mx.xu@siat.ac.cn

***Abstract*—The rapid evolution of cloud-native technologies has facilitated seamless application deployment and execution across the entire edge-to-cloud continuum. This continuum offers a myriad of benefits, including reduced latency, optimized bandwidth utilization, enhanced data privacy, improved reliability, scalability, and flexibility. However, realizing a coherent edge-to-cloud continuum poses challenges especially in resource management, due to the heterogeneous and dynamic nature of computing resources such as resource scheduling and load balancing. This paper focuses on the Container-as-a-Service model enabling independent execution of functions/microservices anywhere on the continuum. We propose an architectural design for constructing a practical edge-to-cloud infrastructure and conduct comprehensive performance evaluations using a real edge-to-cloud testbed. Through an empirical study, we aim to identify key factors impacting application performance and resource management within the continuum, with a specific focus on AI-based IoT applications. Our experiments explore various design patterns including load balancing techniques, scheduling algorithms, invocation methods, gateway and data source location, and factors such as bandwidth and delay, providing practical insights for practitioners and researchers alike.**

***Index Terms*—continuum, edge, cloud, kubernetes, performance, load balancing, edge-to-cloud**

I. Introduction

The rapid advancement of cloud native technologies, including containerization [1], microservices [2], declarative infrastructure-as-code [3], and Function-as-a-Service [4], has paved the way for seamless deployment and execution of applications across the entire spectrum of the edge-to-cloud continuum or simply *compute continuum* [5]. This new vision entails the seamless movement of software components across different levels of computational hierarchy, representing the next stage of integration between IoT and cloud technologies [6]. It offers numerous benefits, such as reduced latency, bandwidth optimization, enhanced data privacy, improved reliability and resilience, enhanced scalability and flexibility, among many others.

The realization of a coherent edge-to-cloud continuum for efficient application deployment, however, presents several challenges, with resource management emerging as a pivotal concern. With the diverse array of computing resources spanning from edge to cloud, encompassing devices like smartphones, IoT sensors, edge servers, and cloud data centers, as well as networks such as cellular, wireless, and wired connections, it becomes imperative to minimize operational complexities, streamline infrastructure management, and simplify the deployment of distributed applications within this continuum of computing. In other words, to fully unlock the potential offered by the continuum and harness the complete capabilities of this distributed architecture, we require effective and efficient management and orchestration of resources throughout the continuum. This paper aims to identify the key factors that impact the performance of applications when various resource management techniques are employed in a heterogeneous and dynamic continuum environment. We specifically focus on the serverless model because the stateless property of functions enables independent execution anywhere on the continuum, optimizing for seamless scalability, which is important for the efficient deployment of applications on the continuum.

We specifically focus on Artificial Intelligence (AI)-based Internet of Things (IoT) applications, commonly referred to as smart applications. Typical examples of applications within the compute continuum are video analytics and image processing. A system overview and a typical data pipeline of an AI-based IoT application deployed in a serverless model across the cloud-to-edge continuum are illustrated in Fig. 1. An IoT device, such as a camera or a robot, generates data in response to an event. This data often requires processing, such as a computer vision task like object detection. The device forwards its requests to the load balancer, which then distributes

§Corresponding authors.

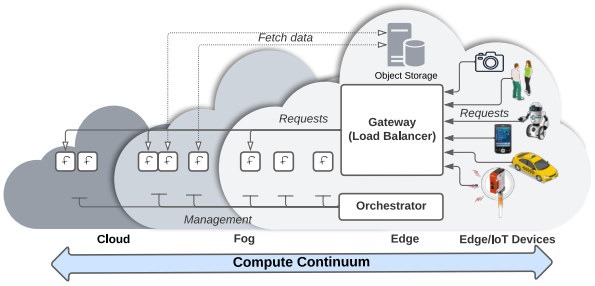


Fig. 1. Overall system overview and data pipeline.

them across heterogeneous computing layers, ranging from edge to fog and cloud. In our example, the load balancer at a gateway forwards the request to a compute node where one of the relevant function's replicas is deployed. The function then performs AI inference and returns the response to the IoT device/end-user for subsequent actuation/decision-making.

Several architectural decisions can potentially influence the performance of this pipeline. The IoT device can either send the generated data along with the request to the load balancer or first store the data in an object storage, attaching a reference to the object in the request sent to the load balancer. The impact of these design patterns, particularly in terms of overhead and practicality for data-intensive AI-based IoT applications, is unknown. The gateway itself must be deployed on one of the computing layers, forwarding the received request to one of the replicas of the functions hosted on a computing node. The optimal placement of the gateway also remains a question. Additionally, the algorithm that the load balancer should employ for load balancing logic is uncertain. The options for this algorithm can vary and can include various algorithms such as round-robin, random, or least connection being widely used in practice. The distinctive bandwidth between the load balancer and the computing layers is likely to significantly influence the load balancer's performance. Conversely, the function replicas are distributed across the computing layers based on the policies of a function scheduler. The policy can lead to significantly varied placements of the replicas, yet there remains limited understanding of the performance implications of these scheduling policies within this context.

This paper aims to address these questions through an empirical study. In this context, our goal is not to propose a new approach or method to address compute continuum challenges. Rather, our focus is on examining the key factors that impact the performance of smart applications deployed within a practical compute continuum. We utilize widely adopted tools and techniques, aiming to offer insights into best practices and identify areas for further research. Thus, we make the following key contributions:

- We propose an architectural design to construct a practical edge-to-cloud infrastructure that spans over resources ranging from small devices, such as single-board computers, at the edge to powerful virtual machines in the cloud. This design utilizes off-the-shelf and commonly used tools like Kubernetes, Tailscale, ZeroTier, HAProxy, and others.
- We empirically conduct comprehensive performance evaluation experiments using a standard *YOLOv3* object detection[1] service deployed on a real edge-to-cloud testbed. We explore the impact of various design patterns and factors, reporting major findings and providing practical insights useful for practitioners and researchers. Specifically, we explore the effects of the location of centralized components, such as the *gateway* and *data source location*.
- We also evaluate various *load balancing* techniques, including *Round Robin*, *Least Connection*, *Weighted Round Robin*, *Random*, and a *sequencing* algorithm [7]. The research also delves into different function scheduling algorithms, such as *Kubernetes' default* and *throughput-aware*. The impact of various function invocation methods, such as *call-by-value* and *call-by-reference*, is also examined. Furthermore, we scrutinize the influence of major constraints, such as *delay* and *bandwidth* on the overall performance.

II. Proposed System Architecture

Various architectural proposals have emerged within the realm of the compute continuum, reflecting the diverse perspectives held by various companies and researchers regarding its implementation [8]–[10]. This paper introduces a generic architecture specifically tailored for the compute continuum as shown in Figure 2. Our proposed architecture seamlessly integrates distributed computing resources spanning a spectrum of devices, including IoT devices, edge and fog servers, cloud servers, and other resources existing within the continuum at the *infrastructure* layer. Central to our architecture is a *management and orchestration (MANO)* layer tasked with composing the compute continuum for the deployment of applications. Positioned at the top layer, we find the *application* layer, comprising software components with their data flow designed according to the serverless programming model.

Our focus is on the MANO layer. A central orchestrator oversees the management of containers, incorporating a scheduler for placement and an auto-scaler for dynamic scaling. Additionally, the orchestrator hosts a gateway equipped with a load balancer, responsible for optimizing resource utilization and requests distribution. We advocate for the utilization of two overlay networks to address both data and control aspects, thereby ensuring robust connectivity among diverse system components. Furthermore, our architecture features a storage system facilitating the storage of data sources and states for functions anywhere within the continuum.

We also use containerized environments to deploy our AI-based IoT applications in a serverless manner in our proposed architecture. Given that we focus on container orchestrations and tools such as Kubernetes which provides a flexible platform for deploying, managing, and scaling serverless applications, we refrain from leveraging specific serverless frame-

[1]https://pjreddie.com/darknet/yolo/

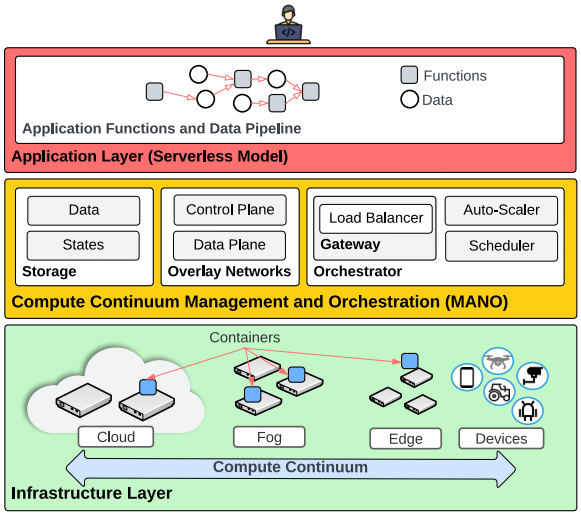


Fig. 2. Proposed System Architecture.

works like Knative, OpenFaaS or OpenWhisk. In addition, certain features of serverless such as Function-as-a-Service (FaaS), Pay-as-You-Go Pricing and Event-Driven Execution, are not crucial to our investigation. Therefore, we find it sufficient to limit our scope to Containers-as-a-Service (CaaS) management. Thus, throughout the remainder of the paper, we intentionally omit explicit references to serverless and its FaaS model, and instead focus on CaaS, even though the discussion remains applicable to both serverless and FaaS.

A. Orchestrator

We propose containerization as the key component for the efficient deployment of applications in the continuum due to its portability, resource efficiency, isolation, and flexibility benefits. Effectively integrating and managing containers at scale across diverse resources within the continuum necessitates the implementation of container orchestration technologies. We advocate for the utilization of orchestration platforms like *Kubernetes* to simplify container management processes in our proposed architecture. The platform establishes a cluster across computing resources for running applications and coordinates microservices and functions deployments seamlessly. It provides various functionalities, with notable scheduling and autoscaling services. The orchestrator is also responsible for hosting a gateway that sits between the IoT devices and the backend services and containers. The gateway receives requests from the edge and IoT devices and forwards them to the available functions/containers. It can be placed anywhere along the edge-to-cloud continuum.

B. Overlay Networks

To establish connectivity among all nodes in our cluster, we propose two distinct peer-to-peer meshed overlay networks. The initial overlay network is configured for *control*, while the second overlay network is responsible for transferring application *data* such as HTTP or MQTT packets from edge devices to the containers and among the containers. Separating the control and data planes in our proposed architecture is essential for enhancing scalability, performance, and security. The control plane transfers management messages for the orchestrator, while the data plane efficiently forwards user and application data. This ensures that control and management data packets are delivered to their destinations regardless of the traffic conditions of the data network. In other words, this design allows for optimized network operations while avoiding operation disruptions due to congestion or disconnection in the data plane. Additionally, with two separate networks, the available bandwidth on each network can be separately controlled. This enables more in-depth studies to be carried out on the effect of bandwidth limitations.

C. Storage

In the compute continuum, managing storage efficiently is crucial for the seamless operation of stateless functions. The storage platform should offer features like object storage, enabling developers to store and manage data effectively while abstracting away underlying infrastructure complexities. Additionally, it would allow developers to trigger functions in response to storage events or access stored data directly from their serverless applications. Moreover, function states, which encapsulate the context and execution state of serverless functions, can also be stored and managed within these distributed storage systems.

III. System Implementation and Testbed

To implement our proposed architecture for the compute continuum and prepare a testbed for evaluation, we assemble a setup as outlined below.

A. Infrastructure

We include various types of nodes from different layers of the continuum, as follows:

- **Edge$_{\text{ARM}}$**: Bare-metal Raspberry Pi 4 Model B 4GB located at the local edge network in the laboratory.
- **Edge$_{\text{VM}}$**: Virtual Machines (VM) on the edge network, running Ubuntu 20.04 LTS and hosted on a local machine using Proxmox Virtual Environment.[2]
- **Fog**: VMs hosted on a nearby cloud infrastructure at the university (Nectar Cloud[3]), running Ubuntu 20.04 LTS.
- **Cloud**: VMs in the cloud, running Ubuntu 22.04 LTS, hosted on Amazon Web Services (AWS) in the US East (N. Virginia) region.

We use varying numbers of each node type in our testbed cluster based on the amount of required resources for the purposes of conducting this research. Table I shows further details of the each of the node types within the testbed.

B. Compute Continuum MANO

Orchestrator: We use *Docker* as the container runtime in our setup. A single Kubernetes cluster is built over the nodes in the infrastructure layer using *K3s*[4], a lightweight distribution

[2]https://proxmox.com
[3]https://ardc.edu.au/services/ardc-nectar-research-cloud/
[4]https://k3s.io/

TABLE I
TYPES OF NODES WITHIN THE TESTBED

Node Type	Edge_{ARM}	Edge_{VM}	Fog	Cloud
Location	Edge	Edge	Fog	Cloud
Is VM?	No	Yes	Yes	Yes
CPU Model	Broadcom Cortex A72	Intel Core i5-4590	AMD EPYC 7002	Intel Xeon E5-2686 v4
CPU Arch.	ARM64	x86	x86	x86
CPU Cores	4 CPUs	2 vCPUs	2 vCPUs	2 vCPUs
CPU Freq	1.5 GHz	3.3 GHz	2.3 GHz	2.3 GHz
Memory	4 GB	4 GB	4 GB	4 GB

of Kubernetes. We use *K3s* as it is highly suitable for IoT and edge computing, and can be easily installed on low-resourced devices such as Raspberry Pi 4 Model B.

The control plane node of the cluster is responsible for orchestrating all containers across the entire cluster, and it can be located anywhere in the continuum. In our setup, we position the K3s control plane at the fog. We note that control plane location has minimal or no impact on the performance and outcomes of our experiments as load is only generated once all worker pods are ready. Therefore, we will not evaluate the impact of the control plane location in Section IV.

Worker nodes are those that execute function pods/replicas and process the workload/requests submitted by clients (IoT devices or end users). A worker node runs scheduled pods for the function under test, and may host multiple of such pods.

We utilize *HAProxy Kubernetes Ingress Controller* as our preferred gateway and load balancer mechanism due to its flexibility in creating custom load-balancing strategies and ease of setup within a Kubernetes cluster. Our proposed architecture enables the *HAProxy* gateway to be deployed on any node along the edge-to-cloud continuum.

All client requests are directed to the gateway, which, in turn, routes them to a specific function or pod based on the employed load balancing algorithm. In our setup, clients issuing function requests are not deployed within the Kubernetes cluster. We assume that these clients consist of IoT devices or end users, always situated at the edge.

Object Storage: We require object storage systems to maintain states or data, such as images, videos, logs, or any other content utilized by our stateless functions. Given that the application used for experimentation is an image processing application which does not have any intermediate state and only requires a data source, we opt to solely use a web server to host and serve image files as the data source. The significance of the location of image files or data source in our experiments is discussed in Section III-C. We employ *lighttpd*,[5] a web server optimized for high-performance environments. The web server nodes operate independently and are not integrated into the Kubernetes cluster, thus unable to host any pods. Nevertheless, they remain connected to the same overlay networks that link all actual nodes in the cluster. Further details about our networking approach between nodes are provided in the following.

[5]https://www.lighttpd.net/

Overlay Networks: The control overlay network is configured through *Tailscale*,[6] a VPN software leveraging *Wireguard*,[7] for control messages. The second overlay network utilizes *ZeroTier* for the data plane, employing its proprietary protocol.[8] The control plane transfers management and control messages for K3s, while the data plane efficiently forwards user data such as images or video frames for the application.

With two separate networks, the available bandwidth on each network can be individually controlled. We employ the *Wondershaper*[9] tool on all worker and web server nodes to impose maximum bandwidth speed limits on the *ZeroTier* overlay network for testing purposes. Internally, *Wondershaper* utilizes the Linux *tc* (traffic control)[10] utility to configure the Linux kernel's packet scheduler, thereby restricting network bandwidth for the chosen network interface. It is important to note that we do not restrict network bandwidth on the device generating load/requests, nor on gateway nodes, to measure the maximum possible throughput within the system under the specified limits.

Table II displays the latencies (in milliseconds) among various types of nodes for the *Tailscale* and *ZeroTier* overlay networks. The latency remains consistently comparable across both overlay networks, as they utilize the same underlying physical network infrastructure.

TABLE II
AVERAGE ROUND-TRIP LATENCY BETWEEN EDGE, FOG AND CLOUD NODES IN MILLISECONDS FOR THE (*TailScale*, *ZeroTier*) OVERLAY NETWORKS MEASURED USING THE *ping* UTILITY

	Edge_{ARM}	**Edge_{VM}**	**Fog**	**Cloud**
Edge_{ARM}	(1.5, 2.0)	(2.2, 2.2)	(33.9, 33.1)	(245.4, 253.3)
Edge_{VM}	(1.3, 1.1)	(1.8, 1.6)	(30.0, 32.5)	(255.6, 248.7)
Fog	(31.2, 35.0)	(33.6, 33.1)	(1.1, 1.8)	(209.0, 208.9)
Cloud	(247.9, 246.0)	(237.8, 238.5)	(208.8, 210.2)	(1.3, 1.7)

C. The Benchmark Application

In recent years, there has been a growing trend of deploying AI and ML models directly on edge devices [11] [12]. Real-time video analytics at the edge is one of the killer applications for the edge-to-cloud continuum [13] and has frequently been used as a benchmark application for performance analysis of resource management techniques at the edge [14].

In this paper, we leverage the *You Only Look Once v3* (*YOLOv3*) object detection model [15] [16] as the main application for our performance analysis. *YOLOv3* possesses several compelling features that make it an excellent choice for our research objectives. It is known to be both *bandwidth-intensive* and *compute-intensive*, resembling the nature of a diverse array of applications that are well-suited for computation in the edge-to-cloud continuum. The *YOLOv3* object detection model can first be pre-trained on a computationally powerful server, and then be seamlessly deployed on various resources across the continuum. We specifically use the *YOLOv3-tiny* model

[6]https://tailscale.com/
[7]https://www.wireguard.com/
[8]https://www.zerotier.com/
[9]https://github.com/magnific0/wondershaper
[10]https://man7.org/linux/man-pages/man8/tc.8.html

TABLE III
AVERAGE CPU TIME TAKEN FOR ONE FUNCTION INVOCATION AS REPORTED BY OUR *YOLOv3* FUNCTION

Node Type	Average CPU Time Per Invocation (ms)
$Edge_{ARM}$	4798.560
$Edge_{VM}$	891.916
Fog	585.281
Cloud	500.636

that was trained on the *Common Objects in Context* (COCO) dataset[11] which, in exchange for lower accuracy/confidence, performs satisfactorily enough on our $Edge_{ARM}$ nodes (i.e. Raspberry Pi 4 Model B 4GB), which are the least powerful nodes in our testbed cluster.

We implement *YOLOv3* object detection as a Python web service application. This is done by utilising *OpenCV* in *Python*, in which we import the *YOLOv3-tiny* model. This allows *YOLOv3* to be invocable as a function call. We then use *Flask*,[12] a lightweight *Python* web framework, to expose a HTTP endpoint where clients can invoke the *YOLOv3* object detection as a web service by sending HTTP requests containing either 1) an image payload or 2) an image URL. We further discuss these two methods of invoking our *YOLOv3* function in Section III-C1. Upon completion of object detection, our *YOLOv3* function then returns an HTTP response that contains the detection results, as well as response headers containing various metrics and information, such as Function invocation start time, Image fetch/load time (I/O time), actual image processing time (CPU time), total elapsed time (sum of I/O time and CPU time), Hostname and IP address of function worker. The resulting *Python* application is then containerized as a multi-platform (x86 and ARM64) *Docker* image and we deploy the same image seamlessly and consistently across all worker nodes in the testbed cluster as a Kubernetes deployment.

Throughout our experiments, we utilize a sample image [17] for processing with *YOLOv3*. The image is compressed to approximately 243.5KB in size and contains 10 car objects. *YOLOv3* infers that there are 13 cars in the image with an average confidence of 0.7524. Under zero load across the cluster, the average CPU time taken for each function invocation/request (excluding all latencies incurred by any I/O such as network transfers) is shown in Table III.

1) Application Function Invocation: We utilize two distinct design patterns (models) for passing arguments to YOLO functions that resemble the traditional programming language paradigms known as "*Call-by-Reference*" and "*Call-by-Value*". In the *Call-by-Value* model, the image itself serves as the payload of the HTTP request sent to the function. For example, a network camera periodically captures an image and sends it directly via HTTP request to the gateway for object detection and processing. Conversely, in the *Call-by-Reference* model, the request does not directly carry the image; rather, it includes the URL pointing to the image source. Subsequently, the function utilizes this URL to retrieve the image for processing. For example, multiple co-located network cameras, each equipped with an adjacent motion sensor, constantly record their surroundings. When motion is detected by a sensor, it can send a request with the event timestamp and the associated data source URL. The function/worker pod can then fetch the recording or image directly from the URL. Figure 3 illustrates the pipeline of request invocation and processing using our containerised *YOLOv3* function for these two models.

Throughout our experiments, we consistently employed synchronous request handling, irrespective of the call model. Unlike systems with a central job queue where the next job is picked up by an available worker, our *HAProxy* load balancer dynamically allocates each incoming HTTP request to an available worker in real-time. Hence, each worker can be seen as an independent parallel queue. We discuss load balancing methods in further detail in Section IV-A2.

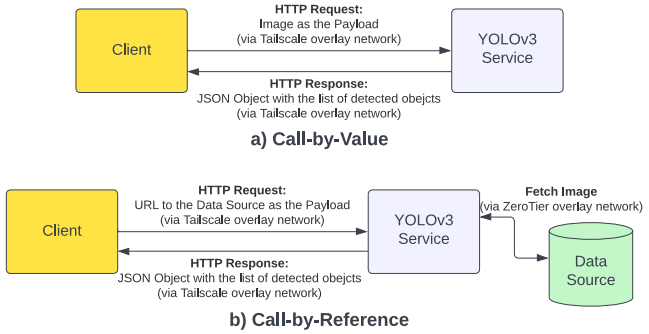


Fig. 3. Invocation pipeline for *YOLOv3* function.

The benefit of incorporating both design patterns of invoking the *YOLOv3* function is twofold. Firstly, it provides us with the flexibility to examine the influence of data source location on load balancing algorithms. Secondly, it encompasses a broader range of application design patterns.

IV. PERFORMANCE EVALUATION

A. Experimental Setup

Before discussing our experiments, we provide some consideration to horizontal scaling, load balancing, and function scheduling policies.

1) Horizontal scaling considerations: To ensure more predictable horizontal scaling behaviour across a heterogeneous cluster, we set each pod instance to be allocated exactly 300 millicores of CPU and exactly 350MiB of memory. As shown previously in Table III, despite equal CPU and Memory allocation across pods, the performance of pods varies based on the performance of the physical CPU on their respective nodes. Given that all our worker nodes have 4GB of Memory, a maximum of 6 pods is deployed at any given time on a given worker node, reserving half of the Memory of the host for the operating system and other applications.

To mitigate potential interference and randomness caused by autoscaling mechanisms, we choose to disable autoscaling in Kubernetes. Instead, we implement a linear increase in the

[11]https://cocodataset.org/
[12]https://flask.palletsprojects.com/

number of function pods within the system, aligning it with the number of threads concurrently generating requests at the load generator. This approach assumes that the number of pods in the system always equals the number of concurrent threads at the load generator. By disabling autoscaling and manually adjusting the pod count based on the generated load, we achieve a more controlled understanding of the load balancing effects on system performance.

2) Load balancing considerations: We use five widely used load balancing algorithms to distribute requests to the function/pods in the cluster:

Random: It is a load balancing technique that distributes incoming requests uniformly random across a set of resources.

Round Robin: In round-robin load balancing, the requests are sequentially allocated to each available worker pod in a circular or round-robin fashion, ensuring each worker pod receives an equal number of requests over time.

Least Connections: To account for the heterogeneous performance of resources, the least connection load balancing algorithm is used to distribute incoming requests based on the current number of active connections each worker pod is handling. The goal is to proportionally distribute the workload among worker pods by directing new requests to the worker pod with the fewest active connections at any given time.

Weighted Round Robin: Worker pods are assigned weight or priority values based on their capacity or performance metrics, like throughput. These weights determine the proportion of requests directed to each pod: higher weights mean more traffic, while lower weights mean less.

Billiard Sequencing Algorithm: Minimizing load balancing overhead for latency-sensitive applications, we explore using billiard sequences [18] to efficiently dispatch requests among function workers in the cluster. We implement deterministic billiard sequencing, which takes into account both the profiled worker performance and the prior dispatch sequence similar to [7]. The use of billiard sequences typically results in reduced queuing time for function requests compared to other deterministic load balancing algorithms like Round Robin and Weighted Round Robin [7].

Out of the five load balancing algorithms discussed, *HAProxy* supports the Random, Round Robin, Weighted Round Robin and Least Connections out of the box. For the Billiard algorithm, we create a custom *HAProxy* configuration file which loads a *Lua* script that implements the Billiard algorithm and sets other configurations and parameters in *HAProxy*. For algorithms that require routing probabilities such as Weighted Round Robin and Billiard, we calculate the routing probabilities for each pod based on the performance of each worker node with the following equation:

$$P_p = (W_n \div M_n) \div \max_{\forall n}(W_n) \cdot 256,$$

where P_p is the routing probability for pod p, W_n represents the weighting of the node calculated based on the number of successful *YOLOv3* function invocations a worker node completes across all of its pods within a window of 60 seconds (Table IV), and M_n is the maximum number of pod replicas.

TABLE IV
PODS' RELATIVE PERFORMANCE ON EACH WORKER BASED ON PROFILING

Node Type	Max Pod Count	Avg. of Successful Invocations Per Node	Avg. of Successful Invocations Per Pod
$\text{Edge}_{\text{ARM1}}$	6	37.667	6.278
$\text{Edge}_{\text{ARM2}}$	6	37.333	6.222
$\text{Edge}_{\text{ARM3}}$	6	33.333	5.556
Edge_{VM1}	6	172.333	28.722
Edge_{VM2}	6	183.333	30.556
Fog_1	6	271.333	45.222
Fog_2	6	257.000	42.833
Cloud_1	6	169.000	28.167
SUM	48	1161.332	193.555

The range weighting values that *HAProxy* accepts for each target is between 0 - 256, inclusive.

3) Function Scheduling Policies: Since resources are heterogeneous in terms of computational power and latency, placement policy will significantly affect the overall performance of the system. To observe the performance of load balancing algorithms, we consider two different placement strategies for the function pods.

Default Kubernetes Scheduler Placement: The Kubernetes default pod placement policy, or scheduler, meticulously evaluates numerous constraints and resource metrics per node to optimize resource utilization across the cluster. Although this placement policy generally results in balanced pod placements across the cluster, it is not deterministic with each experiment run. Therefore, for consistent results, we simulate it deterministically using a predefined static ordering to determine the node on which the next pod will be deployed.

To establish this ordering, we start with a benchmark deployment of zero pods, gradually increasing the number of replicas in the deployment by one per round. In each round, the new replica's node is added to the ordering, assigning it a score equivalent to the current round number. For a benchmark comprising n rounds, each node S may appear in the resulting ordering between 0 and n times ($0 \leq x \leq n$). In our benchmarks, each node appears exactly six times, as all nodes have an equal amount of memory. We conduct the benchmark five times in total, averaging the scores for each node occurrence, S_x. These averaged values of S_x are then sorted in ascending order. In our actual experiments, we deploy three pods per iteration based on this sorted ordering. The final pod scheduling order achieved through this method is presented in Table V.

Throughput-aware Greedy Scheduling: In this Scheduling, functions are strategically positioned on nodes based on their throughput performance. The process starts by choosing nodes with the highest throughput, assigning as many pods/replicas as possible to each node using profiling data. When a node reaches its capacity and can no longer accommodate additional functions, the next node with the best throughput is selected, and the process continues (Table V).

B. Workload Generation

Workload generation is performed using the *Grafana k6*[13] utility as it is able to log events and metrics at a highly granular

[13]https://k6.io/

level out of the box, and scripts for custom load generation and logging behaviour can be easily written and run with it.

The experiment comprises multiple load generation iterations. As previously mentioned, in each iteration, we increment the total number of pods across the system by 3, up to a maximum of 6 pods deployed on any particular node, according to the current scheduling policy at any given time. This incremental addition of 3 new pods per iteration aims to reduce the overall number of iterations required per experiment. Note that, the number of concurrent threads at the load generator equals the number of pods in the system. Once all pods are ready, the load starts to be generated for a window of 30 seconds. After this window ends, no new requests will be generated, but any requests that are still in-flight will be allowed to complete.

TABLE V
DETERMINISTIC POD PLACEMENT SEQUENCE FOR DEFAULT KUBERNETES AND THROUGHPUT-AWARE PLACEMENT POLICIES

Iter	# of Pods	Kubernetes			Throughput-aware		
		Pod 1	Pod 2	Pod 3	Pod 1	Pod 2	Pod 3
1	3	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}2}$	Fog_1	Fog_1	Fog_1
2	6	$\text{Edge}_{\text{VM}1}$	Cloud #1	Fog #1	Fog_1	Fog_1	Fog_1
3	9	$\text{Edge}_{\text{VM}2}$	Fog_2	$\text{Edge}_{\text{ARM}3}$	Fog_2	Fog_2	Fog_2
4	12	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}1}$	Fog_2	Fog_2	Fog_2	Fog_2
5	15	Cloud_1	Fog_1	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}2}$
6	18	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}2}$
7	21	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{VM}1}$
8	24	Fog_2	Cloud_1	Fog_1	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{VM}1}$
9	27	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}3}$	Cloud_1	Cloud_1	Cloud_1
10	30	Fog_1	Cloud_1	$\text{Edge}_{\text{VM}1}$	Cloud_1	Cloud_1	Cloud_1
11	33	$\text{Edge}_{\text{VM}2}$	Fog_2	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}1}$
12	36	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}1}$	Fog_1	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{ARM}1}$
13	39	$\text{Edge}_{\text{VM}2}$	Value 4	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}2}$
14	42	Fog_2	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}2}$	$\text{Edge}_{\text{ARM}2}$
15	45	$\text{Edge}_{\text{ARM}1}$	$\text{Edge}_{\text{VM}2}$	$\text{Edge}_{\text{VM}1}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}3}$
16	48	Fog_1	Fog_2	Cloud_1	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}3}$	$\text{Edge}_{\text{ARM}3}$

C. Experimental Results

In this section, we report major findings and experimental results evaluating the impact of various factors. We specifically explore the effects of load balancing algorithms, call methods, location of data source, available bandwidth, and gateway location. The metrics we focus on are throughput and its inverse correlation, response time. We define performance in terms of maximizing the average throughput, and the cumulative probability distribution (CDF) of the response time. Each experiment is repeated 5 times, and the average and standard error are reported in the results.

1) Effects of Load Balancing: In this section, we evaluate the impact of various load balancing algorithms (as discussed in Section IV-A2) on the performance of the system. We assume the gateway is positioned at the edge and employ *call-by-value* as the default setting, sending the image as the payload of the request. We report results for the algorithms under Kubernetes' default and throughput-aware placement policies. No bandwidth restrictions are imposed, allowing for the utilization of the maximum unrestricted Internet bandwidth available at the edge for our testbed, which is $\sim$100 Mbps for downloads and $\sim$17 Mbps for uploads.

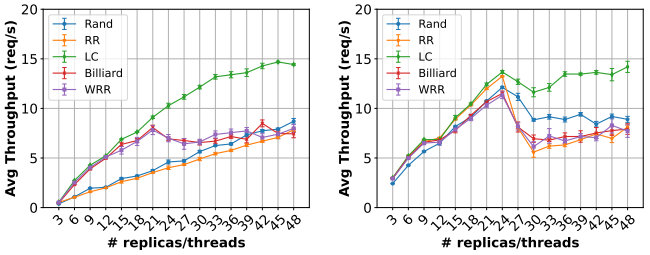
Default Kubernetes placement policy: Figure 4(a) demonstrates that the Least Connections algorithm outperforms all other algorithms, while Round Robin performs the worst due to pods being spread across heterogeneous nodes in the cluster, resulting in a noticeable difference of around 3000-4000 milliseconds slower response time compared to Random at around 60% of served requests (Figure 5(a)). Additionally, we observe that load balancing algorithms requiring routing probabilities to be specified, such as Billiard and Weighted Round Robin, initially perform comparably to Least Connections. However, their performance decreases as more worker pods are introduced to the system, eventually reaching near-equivalent performance levels as Round Robin and Random.

Throughput-aware placement policy: When the throughput-aware placement policy is utilized, we find in Figure 4(a) that the Least Connections algorithm still performs the best throughout all iterations of the experiments. Notably, when all worker pods are located at either the fog and/or edge, we observe that the Round Robin algorithm ranks second in performance after Least Connections. Interestingly, the performance then sharply declines to become the worst overall once worker pods located in the cloud are introduced in the 9^{th} and 10^{th} iterations (27 and 30 active replicas, respectively), although the average performance of cloud worker pods are approximately similar to that of fog worker pods (as per Table IV).

This is due to the fact that worker pods at the cloud node take on average approximately 750 milliseconds ($\sim$250ms latency, $\sim$500ms CPU time) per function invocation compared to worker pods on the Edge_{VM} nodes which take approximately 892 milliseconds ($\sim$2ms latency, $\sim$890ms CPU time), and those at the fog nodes which take approximately 615 milliseconds ($\sim$30ms latency, $\sim$585ms CPU time). This shows that latency plays a non-neglible part in the drop in performance in the 9^{th} and 10^{th} iterations, where algorithms that send a higher proportion of requests to cloud worker pods experience a higher drop in performance. Response times across the board are better with throughput-aware placement policy compared to default Kubernetes placement policy as it avoids sending requests to slower nodes as much as possible throughout the entire lifetime of the experiments.

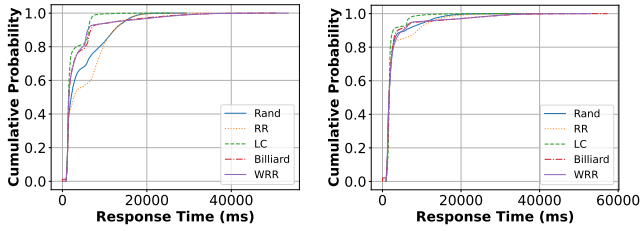
Key Insights: In heterogeneous environments such as compute continuum where worker pods are distributed across diverse nodes in the cluster, the Least Connections load balancing algorithm demonstrates best performance. By dynamically directing traffic to nodes with the fewest active connections, it effectively optimizes resource utilization and minimizes response times. When worker pods are more homogeneous, Round Robin remains a good alternative algorithm to use. Otherwise, Billiard and Weighted Round Robin are viable alternatives, but only when the degree of heterogeneity of worker pods across the cluster is relatively low.

2) Effects of Call Method: In this section, we keep the experiment settings as in the previous section. However, we now switch the call method from *call-by-value* to *call-by-reference*. Unless otherwise specified, from this section onwards, we report results solely for the Least Connections algorithm as it outperforms all other load balancing algorithms.



(a) (b)

Fig. 4. Average throughput of load balancing algorithms, including Random (Rand), Round Robin (RR), Least Connection (LC), Billiard, Weighted Round Robin (WRR) with a) Kubernetes and b) Throughput-aware policies, both using call-by-value method, and gateway and payload at the edge.



(a) (b)

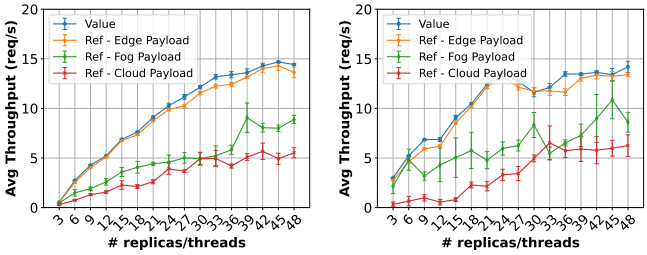
Fig. 5. Response time CDF of load balancing algorithms, including Random (Rand), Round Robin (RR), Least Connection (LC), Billiard, Weighted Round Robin (WRR) with a) Kubernetes and b) Throughput-aware policies, both using call-by-value method, and gateway and payload at the edge.

Although *call-by-reference* has its advantages as described in Section III-C1, it incurs additional latency due to the need to make additional network requests to fetch the payload. This increases the proportion of I/O wait time spent by each worker pod, and network/bandwidth congestion at the data source.

Default Kubernetes placement policy: As Figure 6(a) shows, after changing the call method to *call-by-reference*, when the data source still remains at the edge, the performance remains relatively similar to that of *pass-by-value*, with slight differences attributed to additional network requests. However, when the data source (payload) is moved to the fog and cloud, the impact becomes more significant. In the next section, we delve into this impact with greater detail.

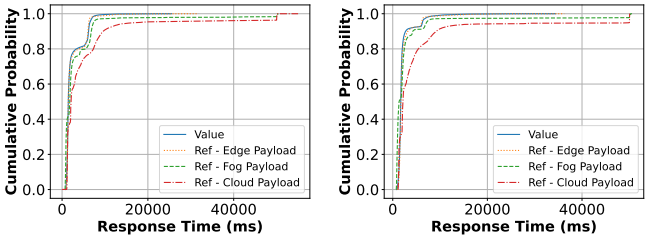
Throughput-aware placement policy: When the throughput-aware placement policy is used, as shown in Figure 6(b), we witness a similar observation, with a slight increase in variability of the performance.

Key Insights: The utilization of both *call-by-value* and *call-by-reference* methods results in relatively similar performance when the data source and the trigger point for the requests remain at the same location. However, if there is a larger distance between the two, the performance drops substantially as network delay plays a significant role. Although the use of *call-by-reference* addresses some key limitations of *call-by-value*, such as overloading the gateway by large requests, the additional network calls incurred add extra delay when using *call-by-reference*. *Call-by-reference* remains a viable option when *call-by-value* is infeasible. Note that regardless of the call method, a throughput-aware placement policy provides better overall response time compared to the Kubernetes default policy by placing pods on nodes with higher throughput



(a) (b)

Fig. 6. Average Throughput of the Least Connection load balancing algorithm for *call-by-value* (Value) and *call-by-reference* (Ref) methods with a) Kubernetes and b) Throughput-aware policies, and gateway and payload (data source) at the Edge, Fog, and Cloud.



(a) (b)

Fig. 7. Response time CDF of the Least Connection load balancing algorithm for *call-by-value* (Value) and *call-by-reference* (Ref) methods with a) Kubernetes and b) Throughput-aware policies, and gateway and payload (Data Source) at the Edge, Fog, and Cloud.

during the earlier stages, as shown in Figure 7.

3) Effects of Data Source Location: We now investigate the impact of the data source location towards the overall performance. Here, different amounts of latency can be added to the response time due to the location of the data source. We focus on the *call-by-reference* method because it permits the movement of the data source. For the *call-by-value* method, the payload data is always on the same device generating the load and is sent with the request, thus eliminating any networking latency in fetching the payload data, and only disk I/O is involved.

Default Kubernetes placement policy: In Figure 6(a), we observe that while the payload resides in the Fog, the performance is slightly better compared to the Cloud. Though the performance gap is evident, it is not as significant as the difference in delay. While cloud latency is at approximately 8 times higher than that of the fog, the disparity in throughput never exceeds two orders of magnitude. This is due to there being a total of 5 nodes at the edge, compared to 2 at the fog and only 1 at the cloud. In addition, due to the limited download bandwidth at the edge, contention among the pods for fetching the remote data exacerbates the situation, resulting in a decrease in overall throughput.

Throughput-aware placement policy: Overall, similar observations apply to the Throughput-aware placement policy in Figure 6(b). More substantial amount of variability in performance can be seen, which in the fog payload scenario starts occurring after more non-fog pods are introduced, and starts in the cloud payload scenario after more edge pods are added to the system.

Key Insights: Bandwidth plays a larger role compared to

latency in affecting performance, especially for data-intensive applications such as image processing. The co-location of as many infrastructure components and as much data as possible in the same segment of the continuum (in this case, at the edge) will generally result in better performance outcomes due to better local link speeds between them (intra-segment), compared to inter-segment link speeds between the edge and fog, fog and cloud, and edge and cloud.

4) Effects of Bandwidth Restrictions: In the previous section, we discovered that available bandwidth can significantly impact performance, even more than the additional delay introduced by the location of the data source for each request. Therefore, to gauge the impact of available bandwidth more precisely, we replicate the experiments using a setup similar to the previous section, but with imposed bandwidth limits. Bandwidth limits are consistently applied simultaneously across all worker nodes and the web server node. Note that the available bandwidth capacity contributes to the overall latency (network wait time) when fetching a payload over the network. Additional delays occur due to contention when any link in the network between two devices becomes saturated.

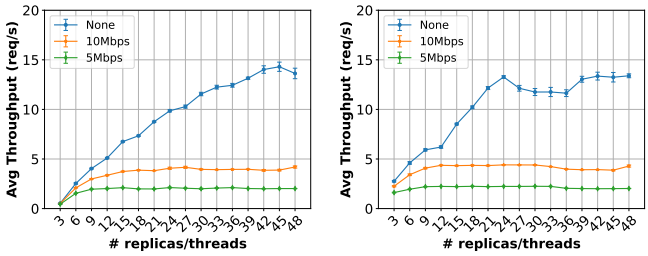
Default Kubernetes placement policy: In Figure 8(a), it is evident that as the bandwidth limit tightens, the system's performance noticeably reduces. This is due to the reduction in data transfer rate caused by lower bandwidth, particularly impacting the transmission of larger data objects (images) from the source to the pod.

Throughput-aware placement policy: A similar trend can be observed for the Throughput-aware placement policy. However, the performance advantage that Throughput-aware policy has over Kubernetes placement policy at the earlier iterations is negated when available bandwidth is sufficiently low.

Key Insights: Bandwidth is shown to be the most critical factor influencing system performance. Specifically, since the edge segment of the network may have lower bandwidth compared to the rest of the continuum, bandwidth capacity must be regarded as the most influential factor affecting system performance.

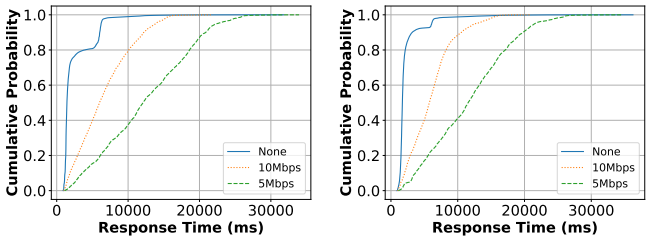
5) Effects of Gateway Location: In this section, we now explore the effects of gateway location. As a gateway serves as the central ingestion point of all requests, its performance can be affected by the location in the continuum, and can become a potential bottleneck in the network. The nature and size of requests also plays an important role, e.g., *call-by-reference* or *call-by-value*. If requests are sent through *call-by-reference* method, the gateway is not involved in the path the payload takes between the data source and the worker pod.

Default Kubernetes placement policy: As shown in Figures 10(a) and 11(a), the location of the gateway can significantly affect the overall system performance in terms of throughput, due to the additional delay introduced by remote layers such as fog and cloud. However, as expected, the impact is more pronounced in the case of *call-by-value*, as it necessitates the entire payload to traverse to the gateway before reaching the designated pods.



(a) (b)

Fig. 8. Average Throughput of the Least Connection load balancing algorithm for *various bandwidth limits* with a) Kubernetes and b) Throughput-aware policies, call-by-reference method, and payload and gateway at the edge.



(a) (b)

Fig. 9. Response time CDF of the Least Connection load balancing algorithm for *various bandwidth limits* with a) Kubernetes and b) Throughput-aware policies, call-by-reference method, and payload and gateway at the edge.

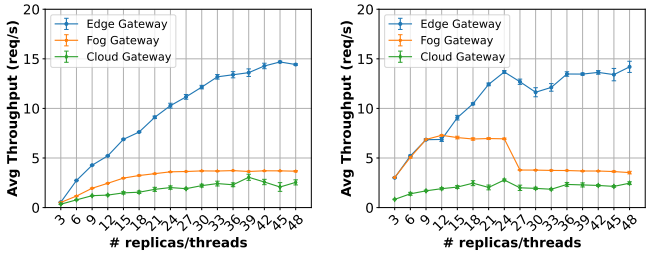
Throughput-aware placement policy: As shown in Figure 11(b), when pods are scheduled according to the Throughput-aware placement policy, the *call-by-reference* method demonstrates lower sensitivity to the gateway's location. However, in contrast to the Kubernetes placement policy, when the gateway is located in the fog compared to when it is located at the edge, the average throughput remains significantly close when the number of replicas ranges from 3 to 12, as shown in Figure 10(b). This is because the gateway and pods are all located in the fog. The system maintains a slightly higher throughput of 7.5 requests per second from 12 to 24 active replicas. The reason is that the Throughput-aware placement policy schedules pods at the Edge_{VM} and fog nodes during these iterations. This level persists until pods are introduced at the cloud node at 27 active replicas, causing the overall throughput to drop to around 4 requests per second for the rest of the experiment.

Key Insights: The location of the gateway can significantly affect the performance of the systems, especially for requests carrying a large payload. The closer the gateway is to the trigger points of the requests, the better the overall performance.

V. Related Work

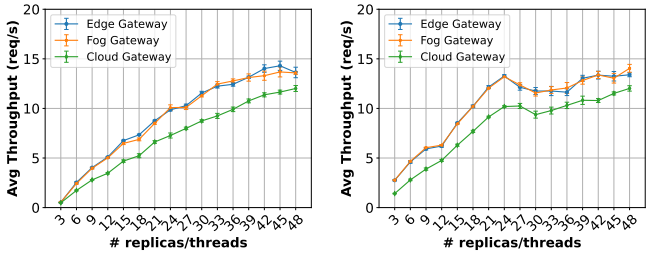
A. Resource Management in Edge-Cloud Continuum

A notable contribution is found in the work by authors [19], which introduces KaiS, a scheduling framework intricately integrated with bespoke learning algorithms designed for Kubernetes (k8s)-based edge-cloud systems. By dynamically acquiring knowledge of scheduling policies for request dispatch and service orchestration, KaiS aims to augment the long-term system throughput rate. Another significant scholarly work [20] introduces Apollo, an innovative framework for



(a) (b)

Fig. 10. Average Throughput of the Least Connection load balancing algorithm for *various gateway locations* for *call-by-value* method, with a) Kubernetes and b) Throughput-aware policies, and payload at the edge.



(a) (b)

Fig. 11. Average Throughput of the Least Connection load balancing algorithm for *various gateway locations* for *call-by-reference* method, with a) Kubernetes and b) Throughput-aware policies, and payload at the edge.

the distribution of serverless function compositions across the cloud-edge continuum.

Taking a different perspective, a study by authors [21] presents Nautilus, a runtime system meticulously crafted to deploy microservice-based user-facing services effectively in the cloud-edge continuum. Nautilus ensures Quality of Service (QoS) while minimizing computational resource requirements, incorporating components such as a communication-aware microservice mapper and a contention-aware resource manager. Additionally, a scholarly contribution [22] introduces MiCADO-Edge, an extension of the MiCADO cloud orchestration framework tailored specifically to edge and fog nodes.

Furthermore, [23] encompasses proposals for dynamic orchestration architectures spanning the Cloud-Edge continuum. These architectures prioritize application QoS by providing schedulers with input parameters commonly utilized in contemporary scheduling algorithms. Addressing cost-effective scheduling, [24] eliminates the necessity for manual placement of edge services in the cloud-to-edge computing continuum. Lastly, [25] outlines the extension of the Knative platform for scheduling complex serverless applications in the Cloud-to-Edge continuum. The objective is to overcome Knative's static scheduling limitations by imbuing the scheduler with awareness of runtime communication intensity, resource usage, and cluster network conditions.

B. Performance Evaluation on Serverless Edge Applcations

Palade et al. [26] evaluate four open-source serverless frameworks (Kubeless, Apache Open-Whisk, OpenFaaS, and Knative) within the context of an edge computing environment. Javed et al. [27] examine the performance of serverless platforms, including OpenFaaS, AWS Greengrass, and Apache OpenWhisk, on single-board computers at the network edge. Comparative analysis with public cloud offerings like AWS Lambda and Azure Functions is conducted. Moereover, conducting a thorough examination, [28] assesses the performance of a serverless edge computing system using well-known open-source frameworks (Kubeless, OpenFaaS, Fission, and funcX). Yet, these studies are different from our work as they only consider the edge and evaluate only the serverless frameworks. Bac et al. [29] introduce an architectural framework for deploying machine learning workloads as serverless functions in edge environments. This work differs from ours as they introduce a new framework.

Focused on processing environments with edge resources, Cilic et al. [30] scrutinize current executions, considering QoS parameters and orchestration platforms. The study assesses prevalent edge orchestration platforms for their workflow in incorporating remote devices, emphasizing the adaptability of scheduling algorithms to enhance targeted QoS attributes. This study does not consider edge-cloud computing and only evaluate the orchestration without investigating different load balancing methodologies.

Similar to our work, which introduces a structured approach, the study in [31] assesses serverless platforms in hybrid edge-cloud infrastructures. The methodology involves establishing a Kubernetes cluster on diverse devices, with a focus on evaluating OpenFaaS, OpenWhisk, and Lean OpenWhisk. This exploration provides insights into the performance and adaptability of these platforms in hybrid environments. However, they do not evaluate performance as comprehensively as our work does, owing to other aspects such as scheduling, load balancing, data source locations, etc.

VI. Conclusions and Future Work

In this paper, we presented a novel architectural design for a practical edge-to-cloud infrastructure, utilizing common tools like Kubernetes, Tailscale, ZeroTier, HAProxy, and others. Through comprehensive emperical performance evaluations on a real testbed, we examined various design patterns and factors affecting the performance of AI-based IoT application in the compute continuum. Results showed that that in heterogeneous environments, the Least Connections load balancing algorithm optimally directs traffic to nodes with fewer active connections, enhancing resource utilization and minimizing response times. The choice between call-by-value and call-by-reference methods depends on the proximity of data sources and request trigger points. Bandwidth emerged as a critical factor, with co-locating components at the edge proving beneficial due to faster local link speeds. The gateway's proximity to request trigger points significantly impacts system performance, especially for large payloads.

In future, we will focus on design and development of scheduling mechanisms that can autonomously adjust system configurations based on real-time changes in workload, network conditions, and resource availability. We also investigate approaches to enhance energy efficiency and sustainability of edge-to-cloud infrastructures.

ACKNOWLEDGEMENT

This work is supported by Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010251) and Chinese Academy of Sciences President's International Fellowship Initiative (Grant. 2023VTB0005).

REFERENCES

[1] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[2] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51, IEEE, 2016.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[4] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pp. 1063–1075, 2019.

[5] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, 2020.

[6] D. Khalyeyev, T. Bureš, and P. Hnětynka, "Towards characterization of edge-cloud continuum," in *European Conference on Software Architecture*, pp. 215–230, Springer, 2022.

[7] B. Javadi, P. Thulasiraman, and R. Buyya, "Enhancing performance of failure-prone clusters by adaptive provisioning of cloud resources," *The Journal of Supercomputing*, vol. 63, pp. 467–489, 2013.

[8] A. Morichetta, N. Spring, P. Raith, and S. Dustdar, "Intent-based management for the distributed computing continuum," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 239–249, 2023.

[9] I. Zyrianoff, A. Heideker, D. Silva, J. Kleinschmidt, J.-P. Soininen, T. Salmon Cinotti, and C. Kamienski, "Architecting and deploying iot smart applications: A performance–oriented approach," *Sensors*, vol. 20, no. 1, p. 84, 2019.

[10] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, "A unified model for the mobile-edge-cloud continuum," *ACM Trans. Internet Technol.*, vol. 19, apr 2019.

[11] L. Ciampi, C. Gennaro, F. Carrara, F. Falchi, C. Vairo, and G. Amato, "Multi-camera vehicle counting using edge-ai," *Expert Systems with Applications*, vol. 207, p. 117929, 2022.

[12] J. Du, H. Wu, M. Xu, and R. Buyya, "Computation energy efficiency maximization for noma-based and wireless-powered mobile edge computing with backscatter communication," *IEEE Transactions on Mobile Computing*, vol. 23, no. 6, pp. 6954–6970, 2024.

[13] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[14] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "Defog: Fog computing benchmarks," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, (New York, NY, USA), p. 47–58, Association for Computing Machinery, 2019.

[15] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[16] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[17] Car and Driver, "2019 10 best cars," 2018. https://www.caranddriver.com/features/a25252134/10best-cars-2019.

[18] A. Hordijk and D. van der Laan, "Periodic routing to parallel queues and billiard sequences," *Mathematical Methods of Operations Research*, vol. 59, pp. 173–192, 2004.

[19] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *IEEE INFOCOM 2021-IEEE conference on computer communications*, pp. 1–10, IEEE, 2021.

[20] F. Smirnov, C. Engelhardt, J. Mittelberger, B. Pourmohseni, and T. Fahringer, "Apollo: Towards an efficient distributed orchestration of serverless function compositions in the cloud-edge continuum," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 1–10, 2021.

[21] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[22] A. Ullah, H. Dagdeviren, R. C. Ariyattu, J. DesLauriers, T. Kiss, and J. Bowden, "Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum," *Journal of Grid Computing*, vol. 19, pp. 1–28, 2021.

[23] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, and M. Marcos, "Quality of service aware orchestration for cloud–edge continuum applications," *Sensors*, vol. 22, no. 5, p. 1755, 2022.

[24] S. Rac and M. Brorsson, "Cost-aware service placement and scheduling in the edge-cloud continuum," *ACM Transactions on Architecture and Code Optimization*, 2024.

[25] A. Marchese and O. Tomarchio, "Orchestrating serverless applications in the cloud-to-edge continuum," in *Proceedings of the 1st International Workshop on Middleware for the Computing Continuum*, pp. 12–17, 2023.

[26] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642, pp. 206–211, IEEE, 2019.

[27] H. Javed, A. N. Toosi, and M. S. Aslanpour, "Serverless platforms on the edge: a performance analysis," in *New Frontiers in Cloud Computing and Internet of Things*, pp. 165–184, Springer, 2022.

[28] Q. L. Trieu, B. Javadi, J. Basilakis, and A. N. Toosi, "Performance evaluation of serverless edge computing for machine learning applications," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pp. 139–144, IEEE, 2022.

[29] T. P. Bac, M. N. Tran, and Y. Kim, "Serverless computing approach for deploying machine learning applications in edge layer," in *2022 International Conference on Information Networking (ICOIN)*, pp. 396–401, IEEE, 2022.

[30] I. Čilić, P. Krivić, I. Podnar Žarko, and M. Kušek, "Performance evaluation of container orchestration tools in edge computing environments," *Sensors*, vol. 23, no. 8, p. 4008, 2023.

[31] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, and D. Soudris, "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*, pp. 428–438, Springer, 2021.