ScaleWave: Breaking Through Resource Bottlenecks to Scale Up Serverless Computing at the Edge

Summit Shrestha*
College of Computing
Georgia Institute of Technology
Atlanta, USA
summitshrestha@gatech.edu

Zheng Song†
CIS Department
University of Michigan at Dearborn
Dearborn, USA
zhesong@umich.edu

Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg, USA
tilevich@cs.vt.edu

Zhengquan Li
CIS Department
University of Michigan at Dearborn
Dearborn, USA
zqli@umich.edu

Christine Julien
Department of Computer Science
Virginia Tech
Blacksburg, USA
christinejulien@vt.edu

Probir Roy
CIS Department
University of Michigan at Dearborn
Dearborn, USA
probirr@umich.edu

***Abstract*—As demand grows, serverless computing systems must scale to meet increasing throughput requirements. Cloud computing easily achieves scalability by allocating abundant and elastic resources. In contrast, edge computing pre-deploys scarce and inelastic resources on site. However, edge applications often need to scale dramatically to handle bursty demand. Because they typically serve fewer users with highly variable workloads, our study shows that peak usage may require up to 8× more resources to be pre-deployed across edge nodes than in a centralized cloud. We observe that a typical serverless request can often be satisfied by different implementations, with dissimilar resource consumption profiles. When an edge application fails to scale up, the culprit is often a single bottleneck resource being fully consumed, with other resources readily available. Motivated by this observation, we introduce SCALEWAVE, a middleware for seamlessly scaling up with different implementations to fully utilize all available resources to achieve scalable serverless computing at the edge. Supporting multiple implementations, however, introduces new challenges for conventional auto-scaler designs. Reactive strategies tend to yield suboptimal performance, while proactive methods struggle with compatibility. SCALEWAVE overcomes these limitations by proactively distributing traffic across implementations, while leveraging existing auto-scalers to manage instance scaling reactively for each one. Our evaluations using real workload traces on a heterogeneous cluster of edge devices demonstrate that our design allows edge applications to serve 2x more requests with minimal latency and achieve 50% more successful requests during workload bursts — showcasing substantial gains in scalability and resource efficiency.**

***Index Terms*—Scalability, Edge Computing, Serverless**

I. INTRODUCTION

Serverless computing is a paradigm in which applications are deployed as on-demand functions, with the resource provider managing all underlying infrastructure [1]. This model frees developers from deploying their implementations and managing resources, allowing them to concentrate on application logic. It is particularly well suited for edge environments, where bringing computation closer to end users reduces latency and supports efficient processing of large data streams—critical for latency-sensitive and data-intensive applications such as real-time analytics, autonomous systems, and interactive services [2]. However, serverless platforms at the edge largely reuse systems originally designed for the cloud [3], which raises challenges when applied to resource-constrained and heterogeneous edge environments.

A defining feature of serverless platforms is **auto-scaling**, the automatic adjustment of function instances to match fluctuating workloads [4]. Existing mechanisms, however, were developed primarily for the cloud and follow a "**single-implementation**-based approach": by reacting to request rates or predicting workloads in advance [5], [6], the auto-scaler decides when to create or remove instances of the same function implementation provided by the service owner. This model is effective in cloud settings, where abundant and homogeneous resources make scaling straightforward [7]. At the edge, in contrast, resources are both limited and heterogeneous, meaning that cloud-specific scaling strategies often fail to deliver the required performance for latency-sensitive and data-intensive applications.

Our preliminary study (Section II) confirms these limitations in edge environments, using real-world mobility traces and edge server deployments. We observed that workload fluctuations at edge nodes are far more volatile than in the cloud, driven by the smaller user base and user mobility. Under such conditions, scaling with a single implementation often depletes the particular resource it relies on, creating bottlenecks that prevent further instance replication. For instance, as shown in Fig. 1(a), a face recognition function running solely on edge CPUs quickly saturates processing capacity and cannot scale further under a cloud-based auto-scaler, even though

*The majority of this work was done when the author was a Master's student at the University of Michigan at Dearborn.

†Corresponding Author

other resources such as memory, GPU, and bandwidth remain underutilized—ultimately resulting in poor throughput. Consequently, handling peak workloads in this way may require up to eight times more total computing resources across all edge nodes than in the cloud, while still yielding low average utilization. These findings underscore the need for scaling strategies tailored specifically to serverless edge environments.

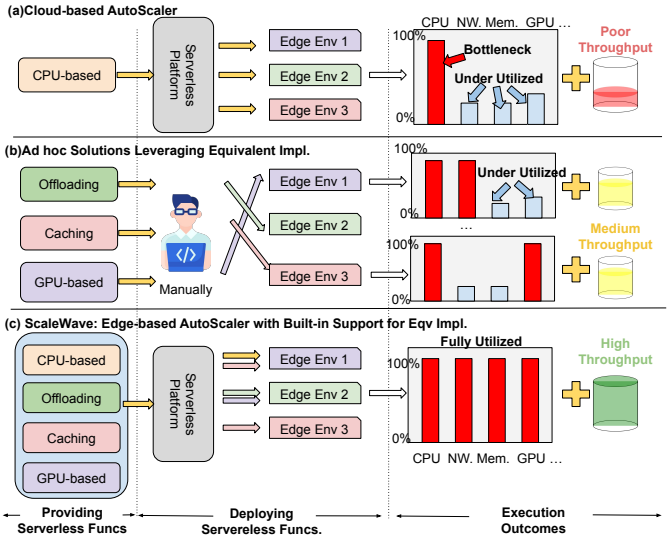


Fig. 1: ScaleWave vs. Existing Edge Scaling Approaches

Existing edge and mobile computing solutions often address resource bottlenecks by leveraging **equivalent implementations**—functionally identical variants with different resource consumption profiles. When one resource type becomes a bottleneck, the system can switch to another implementation to exploit underutilized resources. For example, offloading computation to the cloud [8] increases throughput by utilizing network bandwidth; caching matching results [9] reduces CPU demand by relying on memory; and GPU-based feature extraction [10] shifts workload away from CPUs, as demonstrated in Fig. 1(b). However, these solutions require developers to implement ad hoc logic to monitor specific resource types at an edge environment and adaptively scale, which conflicts with the serverless principle that developers should focus on functionality while the platform manages deployment. As a result, despite many attempts to use equivalent implementations to mitigate edge and mobile resource bottlenecks, existing approaches remain fragmented and cannot systematically scale across all available resources in edge environments. Consequently, although they increase utilization of certain resources, others remain underutilized, resulting in moderate improvements in overall throughput.

This paper addresses this gap by introducing ScaleWave (Scale With What You Have), an edge-native auto-scaler with built-in support for equivalent implementations. As illustrated in Fig. 1(c), ScaleWave automatically adapts to heterogeneous edge resources and manages instances of different implementations to fully utilize these resources, ensuring high throughput. In line with the serverless principle, this process requires developers only to provide the implementations, while ScaleWave handles profiling and dynamic management under fluctuating workloads and constrained resources. The core idea behind our approach is simple yet effective: it controls how request traffic is distributed across implementations, while leaving the underlying auto-scaler to adjust the number of instances for each one. This lightweight design enables ScaleWave to be integrated as an add-on to existing serverless frameworks, providing plug-and-play support for equivalent implementations and ensuring effective scaling, as demonstrated by our testbed evaluation.

The contributions of this paper are four-fold:

1) Identifying Scalability Challenge at the Edge: Through an empirical study using real-world traces, we demonstrate how a combination of limited in-situ resources and bursty workload requests causes the scalability challenge in serverless computing at the edge.
2) Edge-native Auto-scaler Design Supporting Equivalent Implementations: With optimality and compatibility as core design principles, we introduce ScaleWave, a lightweight layer atop existing single-implementation auto-scalers that enable scaling across multiple equivalent implementations of the same service (each utilizing different edge resources). ScaleWave redistributes request traffic across multiple implementations while relying on the underlying mechanisms to reactively adjust instances. To the best of our knowledge, ScaleWave is the first serverless-native system that provides orchestration support for leveraging equivalent implementations to mitigate resource bottlenecks at the edge.
3) Demand-aware Knapsack Formulation and Approximation Algorithm: To maintain high QoS under fluctuating workloads and limited resources, traffic distribution is formulated as an unbounded, multi-dimensional, demand-aware knapsack problem. ScaleWave employs an efficient approximation algorithm to solve this problem, enabling practical traffic allocation that sustains performance while respecting resource constraints.
4) Prototype System Realization: We implement ScaleWave based on Knative [11] by repurposing existing mechanisms (revisions and traffic splitting) to support scaling across equivalent implementations without modifying the underlying platform, and we open-source an initial version on GitHub to motivate future work.

Using real workload traces, we show that ScaleWave improves throughput/normalized performance and reduces tail latency compared to standalone Knative baselines. The remainder of this paper is organized as follows: Section II motivates the need for supporting equivalent implementations in serverless edge computing; Section III presents the design and implementation of ScaleWave; Section IV evaluates ScaleWave; and Section V concludes the paper.

II. Motivation and Related Work

In this section, we first present our empirical study, which highlights the workload fluctuations in edge environments and

max	mean	std	min	25%	50%	75%	85%	95%	99%	99.9%	mean/max
1564.05	561.651953	533.0	907.5	1796.5	1983.25	2066.45	2152.35	2464.09	2603.897	2610.0	0.599253

TABLE I: Request Distribution for Cloud

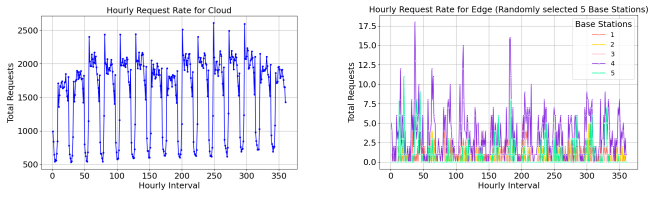
Base Station	max	mean	std	min	25%	50%	75%	85%	95%	99%	99.9%	mean/max
1	6	0.983333	1.096843	0	0	1	2	2	3	4.41	5.641	0.163889
2	5	0.405556	0.842448	0	0	0	0	1	2	3.41	4.641	0.081111
3	3	0.077778	0.324583	0	0	0	0	0	1	1.41	2.641	0.025926
4	18	3.447222	3.019434	0	1	3	5	6	8	13.82	17.282	0.191512
5	11	0.863889	1.710471	0	0	0	1	2	5	8	10.282	0.078535
...	...	...	...	...	...	...	...	...	...	...	...	...

TABLE II: Request Distribution for Edge (5 Base Stations)

shows why traditional scalability solutions designed for the cloud are insufficient at the edge. We then review how equivalent implementations have been used to overcome resource bottlenecks in edge computing along with their limitations, and then examine existing autoscaling solutions and their lack of support for scaling across equivalent implementations.

A. Workload for Cloud and Edge

The differing features between cloud and edge computing lead to their varying workloads. Edge computing brings computation closer to end users, leading to a widespread distribution of edge nodes, each serving users within a limited geographical coverage to meet latency requirements. Users' mobility patterns across different edge nodes lead to highly dynamic edge workloads. To better understand the workload differences between cloud and edge environments, we conducted a case study based on a real edge connection trace using the Shanghai Telecom dataset [12]. The trace captures workload volatility in small-population edge sites and we use it as a conservative upper bound on volatility rather than an exact deployment plan. This dataset includes Internet access records through 3,233 base stations (BS) from 9,481 mobile phones for a duration of six months. We preprocessed the data to remove null values for latitude and longitude, resulting in 563914 records from 6262 unique users. To map latitude and longitude data to base stations, we grouped them together and selected unique groups, resulting in 2769 base stations. We assume that each base station here denotes an edge node.



(a) Cloud Requests Pattern (b) Edge Requests Pattern (5 BS)

Fig. 2: Cloud Workload vs Edge Workload

For cloud workloads, we aggregated workloads from all the base stations, as shown in Fig. 2a. Based on the request distribution, we computed the average and maximum request values, which were 1564 and 2610, respectively. To obtain a value representing the variation in between the two, we further calculated their ratio (mean/max), which was found to be **0.59**. Table I shows the overall distribution of the cloud requests.

For edge workloads, we also divided the dataset into hourly intervals and filtered it by unique base stations. We then aggregated the hourly requests across all base stations, yielding a total of 996,840 hourly request entries. Fig. 2b shows the hourly requests for 5 base stations to illustrate edge workload properties. For each of the 2769 base stations, we calculated the average and maximum request values and their (mean/max) variation ratio. Table II presents the hourly request distributions for five randomly selected base stations. The average (mean/max) variation ratio across all base stations was **0.07**, providing a comparable metric against the cloud workloads.

This study found that the mean–max ratio of requests is much higher in the cloud (0.59) than at the edge (0.07). In other words, cloud workloads exhibit relatively stable demand, while edge workloads experience far more severe bursts. Assuming a linear relationship between requests and resources, sustaining these bursts would require nearly eight times more redundant resource deployment at the edge ($\frac{0.59}{0.07} \sim 8$x). Such redundancy is particularly problematic because current serverless platforms tightly couple a function to a fixed type of resource so throughput collapses once that resource becomes a bottleneck. In cloud environments, abundant resources allow auto-scalers to simply replicate instances of a single implementation, but at the resource-scarce edge, this strategy wastes capacity and incurs high cost without effectively absorbing workload spikes. These observations demonstrate why traditional "single-implementation" based scalability solutions designed for the cloud are insufficient for edge environments.

B. Ad-hoc Solutions for Scaling with Equivalent Impls.

In resource-constrained edge environments, bottlenecks can be alleviated by providing *functionally equivalent implementations* of a service that use different resource mixes while producing the same result. For instance, a neural network may execute on CPU, GPU, DSP, or NPU depending on availability [13]. Android's Neural Networks API (NNAPI) makes this process transparent: workloads automatically use the most efficient accelerator when available, but fall back to CPU if others are busy or absent [14].

Beyond hardware accelerators, equivalent implementations also arise in *dynamic offloading*, where a task can either run locally on a device or be offloaded to an edge or cloud server. Both options yield identical outputs, but the choice depends on context: offloading can save energy and reduce latency when bandwidth is plentiful, while local execution avoids network

delay when connectivity is poor [15]. A third example is *caching*, which offers an alternate path between recomputing a result or retrieving it from storage. Systems can dynamically choose between the two, trading CPU cycles for memory depending on load and urgency [16].

Together, these strategies exploit multiple interchangeable execution paths to balance performance, energy, and resource use. Yet existing solutions implement ad hoc logic to monitor specific resource types and scale adaptively. Despite their effectiveness, such mechanisms are generally supported only in an *ad hoc manner* (e.g., [9], [13]–[15]), typically designed for narrow application domains. As a result, although many equivalent implementation–based approaches have been proposed to mitigate edge and mobile resource bottlenecks, they remain fragmented because each focuses on only certain types of resources (e.g., CPU, GPU, memory, or network) and lacks a unifying mechanism to coordinate them. Without systematic orchestration and a general scaling mechanism, these solutions cannot scale across the heterogeneous mix of resources available in edge environments, leaving many resources underutilized and bottlenecks unresolved. Next, we review existing autoscaling solutions to understand their support in scaling across equivalent implementations.

C. Existing Auto-scaling Solutions

Existing studies on auto-scalers can be broadly categorized as reactive [6], [17]–[19] or proactive [5], [20]–[24], focusing on maximizing utilization through optimal allocation [25], [26], parallelism and multiplexing [27], [28], and profile- or contention-aware placements [29], [30]. Yet, they mainly target homogeneous cloud resources and overlook the constraints of heterogeneous, resource-limited edge environments.

Autoscaling at the edge faces tighter resource budgets and stricter latency constraints than the cloud, making function-based serverless deployments a popular choice. Prior autoscaling efforts at the edge primarily adjust capacity of a single implementation by changing replica count and sometimes per-instance resources and concurrency. Knative hybrid autoscaler [31] jointly tunes replicas, per-instance CPU/memory, and concurrency using profiling and traffic prediction, but it still scales a single service implementation saturated by CPU or memory. KneeScale [18] targets budget-constrained edge deployments by scaling to a knee point of diminishing returns to choose an efficient replica count, again assuming one implementation per function which depends on a single type of resource. RIA [32] optimizes ROI (QoS per cost) using a SHAP-based DQN to select scaling actions, but likewise adapts capacity within a single service implementation over time. To our knowledge, no existing auto-scaler supports scaling across multiple implementations of the same function, and most approaches require platform modifications.

Prior work on offloading, caching, or accelerator fallback relies on ad-hoc, application-specific logic for which serverless platforms lack orchestration support. Our approach with SCALEWAVE provides a unified, lightweight layer that profiles each implementation, observes resource pressure at runtime, and redistributes traffic so existing autoscalers scale replicas across heterogeneous resources without modification. This lets the system continue scaling even when one resource (CPU, uplink) saturates by shifting traffic to implementations relying on underutilized resources (GPU, memory), expanding scaling capability beyond optimizing a single bottleneck absent in today's serverless platforms.

III. SCALEWAVE: DESIGN AND IMPLEMENTATION

Motivated by these observations, we present SCALEWAVE, a lightweight auto-scaler for edge-based serverless computing. SCALEWAVE systemically supports equivalent implementations to scale bursty workloads under constrained, heterogeneous resources. This section first outlines the challenges and corresponding design choices, then details the system components, and finally describes our reference implementation.

A. Challenges and Solution Overview

Auto-scaling with equivalent implementation requires fundamentally reconsidering underlying assumptions and novel system designs. Compared to scaling a single implementation, multiple implementations possess distinct resource consumption patterns and Quality of Service (QoS) outcomes. One intuitive approach is to adhere to existing reactive auto-scaler designs and dynamically decide which implementation to spawn as requests increase. However, this approach cannot guarantee global optimality, as achieving optimal scaling sometimes requires proactively terminating instances of certain implementations to free resources for others. Alternatively, controlling both the number of instances and the allocation of requests proactively for each implementation can achieve global optimality but introduces compatibility issues. This approach complicates the interaction between the auto-scaler and other modules of the serverless framework, such as the concurrency controller and lifecycle manager.

Our key innovation for scaling across equivalent function implementations is **scaling by traffic distribution**, a mechanism that proactively directs incoming requests across equivalent implementations and thereby triggers the underlying single-implementation serverless autoscalers to scale instances up/down according to their native policies. By explicitly controlling the distribution weights among implementations while relying on the platform's implicit scaling actions, this approach provides a lightweight but effective abstraction that extends existing serverless platforms to support equivalent implementations, enhancing both generalizability and compatibility.

Fig. 3 presents an overview of our solution. SCALEWAVE is introduced as an additional layer (highlighted in green) that provides high-level abstractions to enable multiple equivalent implementations of application services on top of the existing serverless framework (marked in blue). Internally, SCALEWAVE treats each equivalent implementation as a stand-alone serverless function, but it logically groups them as alternative implementations of the same service and exposes a unified access interface at the top layer. For each implementation, the underlying serverless auto-scaler still manages the creation

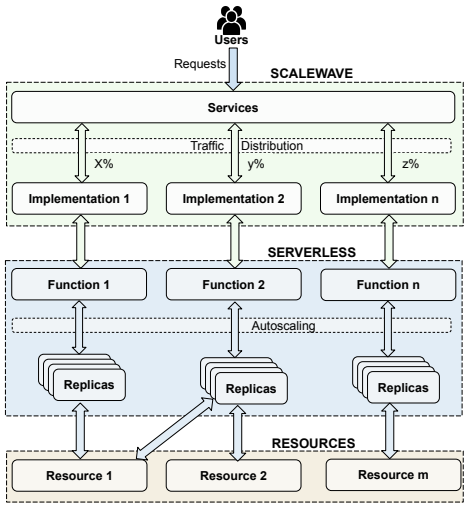


Fig. 3: Solution Overview of SCALEWAVE

and removal of replicas in response to incoming requests. At runtime, SCALEWAVE directs requests to the most suitable implementation based on current resource availability and performance benefits, thereby achieving scaling across equivalent implementations while leveraging the built-in autoscaling mechanisms of existing serverless platforms.

B. Traffic Distribution Workflow

SCALEWAVE consists of three components: an **Equivalent Service Manager**, an **Optimizer**, and an **Observer**. The Equivalent Service Manager is composed of a configuration handler, an equivalence handler, and a route handler, and it is responsible for interfacing with both end users and the underlying serverless platform to deliver services. The Observer and Optimizer operate together to analyze runtime statistics and compute the optimal traffic distribution.

Fig. 4 illustrates our overall system workflow in a distributed edge environment, where the components highlighted in green represent the physical and logical components of SCALEWAVE, while the others represent components of the underlying serverless platforms. The overall system workflow is divided into two stages: offline and online. During the

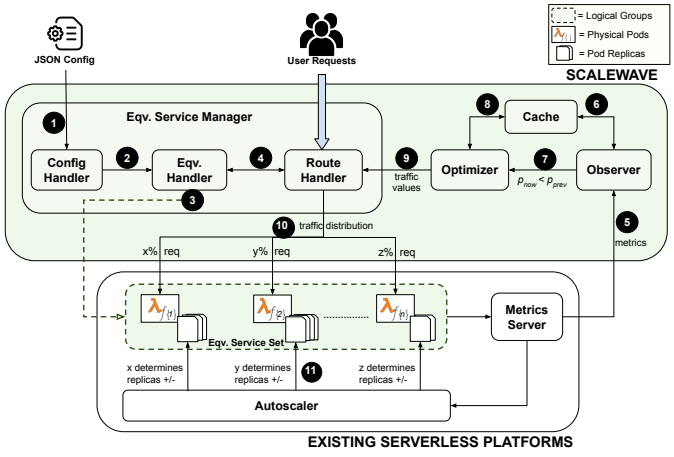


Fig. 4: Traffic Distribution Workflow

offline stage, the Config Handler reads the developer-supplied configurations containing 1-to-M service-to-implementation mappings❶. The Config Handler then parses these configurations and invokes the Eqv. Handler with a create or update call ❷. The Eqv. Handler leverages the underlying serverless platform's APIs to create each implementation as a separate serverless function, logically grouping them as an Eqv. Service Set❸. For each equivalent implementation (underlying serverless function), the Route Handler initially assigns a unique, addressable named route with equal traffic distribution weights and communicates this to the Eqv. Handler for necessary bookkeeping ❹. This marks the completion of the offline phase and triggers the start of the online phase.

The online phase executes continuously in the background as an independent daemon process at periodic intervals. It begins with the Observer filtering the necessary "requests" and "resources" metrics related to the Eqv. Service Set (metrics for all underlying serverless functions) and relevant autoscaler metrics, and then computing the performance ❺. The result is saved to the cache ❻ for future reference ❽. If the current performance is lower than the previous performance, it triggers the Optimizer Component❼. The Optimizer then solves the performance optimization problem to obtain the new replica counts, calculates the new traffic distribution values accordingly, and triggers the Route Handler with those values❾. The Route Handler, using these updated weights, reconfigures traffic distribution for each named route associated with each equivalent implementation (underlying serverless function)❿. Under the hood, the Autoscaler, which triggers the creation and deletion of replicas for each serverless function based on incoming request counts, is indirectly driven by ScaleWave, as ScaleWave determines the number of requests sent to each underlying serverless function⓫.

C. Traffic Calculation Algorithm

1) Demand-based Dynamic Knapsack Problem: An intuitive approach to maximizing the overall utility provided by all implementations under limited resources is to formulate the performance optimization problem as a classical knapsack problem, where equivalent implementations are prioritized based on their performance-to-resource ratio.

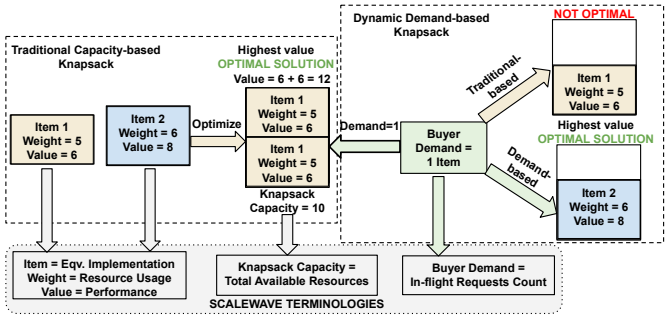


Fig. 5: Dynamic Demand-based Knapsack

However, when an edge system receives few requests, solving the traditional knapsack problem may yield suboptimal performance. This is because the solution may favor

implementations with a higher performance-to-resource ratio, rather than those that deliver better performance per request. As illustrated in Fig. 5, consider a knapsack with capacity 10 (resource constraint) and two items: Item1 and Item2, with weights of 5 and 6, and values (i.e., per-request performance) of 6 and 8, respectively. The traditional solution selects Item1 twice (5×2=10) for a total value of 12 (6×2), which corresponds to creating two replicas of implementation 1 for two requests. However, if there is only one request, assigning it to implementation 2 yields a higher performance (8 vs. 6). This example shows that maximizing knapsack profit alone does not guarantee optimal performance when demand is fixed.

To address this limitation, we introduce the concepts of buyer and demand into the knapsack problem, resulting in a new formulation we call the Demand-based Dynamic Knapsack. In this model, a buyer at time t requests a specific number of items d, which may vary over time. The goal is to select the most profitable d items within the resource constraint. In the previous example, Item 2 would now be chosen, as it yields better performance under limited demand. In our system, demand corresponds to incoming serverless function requests, and item selection determines which equivalent implementation's replicas to create.

2) System Modeling and Problem Formulation: We model the edge system as a black box with finite amounts of resources across multiple categories. Let $\mathcal{R} = \{r = 1, 2, \ldots, R\}$ denote the set of resource categories, and let $\{A_r \mid \forall r \in \mathcal{R}\}$ represent the amount of each resource currently available for allocating new replicas. For a given service, we define $\mathcal{E} = \{e = 1, 2, \ldots, E\}$ as the set of its equivalent implementations. As mentioned earlier, we adopt $p = t/l$ as the target performance metric, where t is the throughput (i.e., the number of successful requests per time unit), and l is the end-to-end latency, including cold-start, queuing, and processing delays. This metric p increases with higher throughput and lower latency. As shown in Fig. 4, SCALEWAVE uses this observed ratio $p = t/l$, avoiding prediction and maintaining accuracy under fluctuating workloads.

For each equivalent implementation e, let n_e denote the current number of replicas and x_e the desired number of additional replicas. At runtime, we measure the throughput t_e and end-to-end latency l_e of e, and compute the total performance as $p_e = t_e/l_e$. The performance per replica is then given by $p_e^n = p_e/n_e$. Accordingly, our optimization objective is to maximize the total performance contributed by the new replicas, formulated as: $\max \sum_{e=1}^{E} p_e^n \times x_e$

Next, we define the constraints: 1) The primary constraint ensures that the total resource consumption by all new replicas does not exceed the available capacity. Let a_e^r denote the average amount of resource r consumed by each replica of e to achieve performance p_e, we have $\sum^{\forall e \in \mathcal{E}} a_e^r \times x_e \leq A_r, \forall r \in \mathcal{R}$; 2) Another constraint arises from service demand, i.e., the number of additional replicas to be created should not exceed the demand, implied by the currently queued and in-flight requests. Let c_e denote the concurrency target of e, representing the maximum number of concurrent requests that each replica can handle before requiring scaling, typically specified by the developer. Let q_e denote the number of in-flight or queued requests for e, which can be measured at runtime. We have: $x_e \leq \max\left(\left\lceil \frac{\sum_{e=1}^{E} q_e}{c_e} - \sum_{e=1}^{E} n_e \right\rceil, 0\right) + 1, \forall e \in \mathcal{E}$.

Putting everything together, we formulate the performance optimization problem as below:

$$\begin{aligned} &\mathcal{X} = \arg\max \sum_{e=1}^{E} p_e^n \times x_e \\ &s.t.: \sum^{\forall e \in \mathcal{E}} a_e^r \times x_e \leq A_r, \forall r \in \mathcal{R} \\ &s.t.: \sum_{e=1}^{E} x_e \leq \max\left(\left\lceil \frac{\sum_{e=1}^{E} q_e}{c_e} - \sum_{e=1}^{E} n_e \right\rceil, 0\right) + 1, \forall e \in \mathcal{E} \end{aligned} \tag{1}$$

By incorporating dynamic demand as a constraint, Equation 1 becomes a multi-dimensional unbounded knapsack problem—an NP-hard problem that has been extensively studied. To ensure efficient solving, we adopt a metaheuristic-based genetic programming approach. The additional replica counts obtained from solving the optimization problem in Eq. 1 are used to recompute the new traffic distribution weights. The basic idea is to distribute the traffic to each implementation based on their replica counts (i.e., $x_e + n_e$) and the throughput of each replica (i.e., t_e^n). Let TD_e denote the updated **T**raffic **D**istribution weight for each equivalent implementation e, computed as:

$$TD_e = \left(\frac{t_e^n \times (x_e + n_e)}{\sum^{\forall e \in \mathcal{E}} t_e^n \times (x_e + n_e)} \right) \times 100\%, \forall e \in \mathcal{E} \tag{2}$$

D. Developer Support for Equivalence

SCALEWAVE provides a language-agnostic, declarative model for defining equivalent implementations using JSON, where developers quickly create implementations by specifying a `"name"` and container `"image"` for each. Functions map to multiple implementations (1-M), each tied to a single container image (1-1), as shown in Fig. 6. The system integrates with the serverless platform to manage their lifecycle and supports special-purpose resources (e.g., GPUs) by allowing developers to include a `"resource"` key with the appropriate label. Each resource is represented by a static tag (e.g., `"gpu"`) applied to cluster nodes, enabling the system to add a `"NodeSelector"` constraint so that implementations deploy only on matching devices.

E. Reference Implementation

We implemented SCALEWAVE and released its initial version as open source on GitHub[1]. SCALEWAVE is implemented as an add-on component that integrates seamlessly with the existing Knative infrastructure, requiring no modifications to the underlying platform. The implementation, written entirely in Python, comprises approximately 1,800 lines of code across the runtime system and the developer support module.

[1]https://github.com/898djh/ScaleWave.git

```
{"services": [{
      "name": "face-recognition",
      "equivalent_implementations": [{
           "name": "edge-cpu-based",
           "image": "docker.io/<user>/<img>",
           "env": {"key1": "value", "key2": ".."}
        },
        {
           "name": "edge-gpu-based",
           "image": "docker.io/<user>/<img>",
           "resource": "gpu"
        }]
    },
    .....
]}
```

Fig. 6: JSON-based Equivalence Definition

In particular, our reference implementation builds on Knative's built-in features—*revisions*, which represent snapshots of application code and configuration [33], and *traffic splitting*, which distributes incoming requests across multiple revisions [34]. While these features are typically used for deployment strategies such as blue-green deployments [35] and canary releases [36], we repurpose them to enable scaling across equivalent implementations. On top of Knative's static traffic splitting, we build a dynamic runtime that monitors workload and resource usage at runtime and proactively adjusts traffic allocation among revisions.

IV. Evaluation and Result Analysis

By deploying the ScaleWave system in a real-world testbed, we conducted a comprehensive evaluation of our proposed approach and compared it with baseline methods. Specifically, our evaluation was driven by the following evaluation questions:

EQ1: Utilizing the same resources, what is ScaleWave's impact on overall function performance?

Finding: ScaleWave improves the normalized function performance by almost 42%, successfully serving 2x more requests with minimal latency (under 50 seconds for over 6000 requests per hour) using the same resources compared to baseline approaches (see Sec. IV-B for more details).

EQ2: What is ScaleWave's impact on overall resource utilization?

Finding: ScaleWave improves overall resource utilization by over 20% with minimal resource wastage compared to baseline approaches (see Sec. IV-C for more details).

A. Experimental Setup

We setup a cluster of edge devices representing an edge node and implemented our motivating "face recognition" function and their equivalent implementations as container images for our test suite. The "face recognition" function was implemented using an open-source Python library [37]. We deployed our test function to the cluster and executed it following an inter-arrival pattern obtained from a real-world edge connection trace.

Baseline Approaches: To compare with single-implementation-based auto-scaling, we use the following functions in conjunction with Knative's built-in auto-scaler—namely, the concurrency-based Knative Pod Autoscaler (KPA)—as baseline approaches.

- *Offloading-only + default KPA (Offload):* The offloading-only approach uploads the image directly to a cloud server.
- *Local-only + default KPA (Local):* Here, it first finds the face region in a given image. Then, it extracts the features of the face region into a float array of size 32. This encoding is compared with a set of given images, and the name of the person is returned based on the closest match in the feature database.
- *Local-Offloading + default KPA (Local-Offload):* This approach extracts the encoding at the network edge and uploads only the serialized float array to the cloud.

Testbed Configuration: We set up a cluster of heterogeneous edge devices: a stronger x86 node, a Pi-class device, and a Jetson GPU node (specifications in Table III) for our deployment testbed. This mix mirrors common edge clusters where devices with different resource capabilities jointly serve requests. The cluster included one manager node and three worker nodes, differing in architecture and resources offerings. We ranked these nodes based on a combination of their basic resource configurations (CPU, Memory, Disk, and Network) and selected the one offering the highest configuration as our manager node, while the others served as worker nodes. Additionally, we hosted an instance of the application on a cloud server using Knative Serving for our "Offloading-only" baseline.

Devices	Lenovo Thinkpad	HP ProDesk	Raspberry Pi 4B	Jetson Nano
Role	Manager	Worker	Worker	Worker
Architecture	Intel i7 x86	Intel i5 x86	ARMv8	ARMv7
OS	Ubuntu 22.04	Ubuntu 20.04	Raspbian	Ubuntu 18.04
CPU (cores)	12	6	4	4
Memory (GB)	16	8	8	4
Disk (GB)	512	256	64	128
Disk Type	SSD	HDD	MicroSD	MicroSD
GPU (cores)	x	x	x	128

TABLE III: Testbed Specifications

Application Function and Equivalent Implementations: For the "face recognition" function, we developed three equivalent implementations relying on different resources, summarized in Table IV. These implementations were provided as API endpoints developed using the Flask framework and built as container images with Docker's multi-architecture build.

Eqv Method	**Deployment**	**Resources**
1	Offloading-only	Network
2	Local-only	CPU, GPU
3	Local-Offloading	CPU, Network

TABLE IV: Equivalent Implementations for Face Recognition

Workload Generation: As a real edge workload trace is not readily available, we still used the Shanghai Telecom dataset from our empirical study to generate an hour-long edge workload trace. We randomly selected a month-long connection trace from one base station and sampled it to create an hour-long workload trace, ensuring enough data points to showcase bursty scenarios. Finally, we used the

sampled one-hour connection trace dataset and calculated the differences between connection start times to obtain a set of inter-arrival times (in seconds) for each unique user. We played this previously generated workload trace for each baseline approach and our approach at varying concurrency levels, collecting results for each run. To vary the data sent during requests, we defined a static list of images with different sizes in our script and randomly selected one for each request.

Task Allocation: Task allocation maps each request to an equivalent implementation and node. It proceeds via (1) resource-level filtering (e.g., GPU replicas only on GPU-labeled nodes) and (2) node-level heuristic selection to best match implementation-to-resource needs. Once allocated, SCALEWAVE redistributes traffic accordingly.

Concurrency Targets (c): Concurrency targets, set by developers, define the number of concurrent requests a replica can handle before additional replicas are provisioned. To evaluate the robustness and adaptability of both the baseline approaches and our proposed method, we conducted experiments under three different concurrency settings ($c_e = 70, 35, 18$), corresponding to high, medium, and low concurrency levels, respectively. These settings simulate a range of deployment scenarios—from replicas designed to handle large volumes of traffic with minimal scaling (high concurrency) to those that are more sensitive to load changes and require more frequent scaling (low concurrency). By varying c, we assess how each method performs under different resource-utilization pressures and scaling behaviors, providing a comprehensive understanding of their effectiveness across diverse operational conditions.

B. Throughput and Performance

For latency-sensitive edge systems, evaluating performance based solely on throughput can result in high latency. Therefore, both throughput and latency must be considered. These are competing measures, i.e., an edge system will have a high performance if it can service more requests with minimal latency. Thus, we measure the normalized performance as the ratio of throughput to average latency.

Throughput: Throughput refers to the total number of requests an edge system can successfully serve over a time period. To measure and compare the throughput of our approach against other baselines, we ran the hour-long workload trace for each approach, with a default timeout of 5 minutes (300 seconds) and varying concurrency values ($c_e = 70, 35, 18$). After the run, we summed the number of requests returning a 2xx status code (e.g. 200 STATUS OK, denoting successful HTTP response) to obtain the successful request count.

Latency: Latency refers to the total time taken by the system to serve user requests from the time they are issued, encompassing the time to achieve a certain throughput. Various factors contribute to end-to-end (E2E) latency, including processing time, queuing delay, cold-start latency, and network latency. During function executions, we measured E2E latency for each successful request by calculating the time difference between request initiation and response received.

Normalized Performance: Normalized Performance refers to the system's performance in terms of their throughput and average latency. We calculate normalized performance as the ratio of throughput to average latency, with higher throughput and lower latency indicating better performance. Based on these two values collected during function executions, we aggregated total throughput and computed the average latency for each baseline across three concurrency values.

Fig. 7 compares the number of successful requests, their average latency values, and normalized performance over one hour, with a total of 8846 requests issued for each baseline using three different concurrency values. For c=70, SCALEWAVE achieved the highest number of successful requests, serving almost 3000 more requests than the offloading-only, 2000 requests more than the local-only, and almost 1000 requests more than the local-offloading baseline approaches. In terms of latency, it reduced E2E latency by nearly 2x that of the offloading-only, by 1.2x compared to local-offloading, and increased by 1.2x compared to the local-only, offering a better balance in latency values. For lower concurrency values (c=35 and c=18), which impose more replica creation, our approach maintained a similar throughput to the offloading-only and achieved higher throughput than local-only and the local-offloading approaches. This is caused due to the cloud's unlimited resources supporting more replicas thus handling more offloaded requests, unlike local-only and local-offloading, which fail to serve as many requests. However, this advantage of the offloading-only approach comes with higher latency due to network delay, while our approach consistently keeps latency under 50 seconds for serving over 6000 requests in all cases.

An interesting observation in latency values for c=35 and c=18 is that SCALEWAVE consistently has the lowest latency among all baselines. This reduction is due to SCALEWAVE's unique design for supporting equivalent implementations, which decouples function from specific resources and simultaneously utilizes other non-competing resources, while other baselines are tightly coupled to the same set of resources. This effect is more pronounced with lower concurrency (c=18), as more replicas must be created quickly to handle high incoming requests, soon throttling resources and resulting in higher cold-start and overall latency, thus increasing E2E latency. Moreover, among all baselines and for each concurrency value, SCALEWAVE offers the highest normalized performance, improving by up to 1.5x as target concurrency decreases.

Fig. 8 compares the tail average latency (p85/p90/p95) at three concurrency targets (c=70,35,18). Local is always saturated at 300s; Local-Offload is 300s at c=70 and remains $\sim$170–300s at lower c values. Offload helps only at lighter loads (c=18, p85$\approx$50–60s) but collapses to 300s by p95 for c$\geq$35 and at all percentiles for c=70. ScaleWave consistently achieves lower tail latencies, which degrades smoothly with percentile, representing $\approx$50–85% lower tails than the other baselines.

Summary: SCALEWAVE outperforms local-only and local-offloading baselines by serving more requests with the same

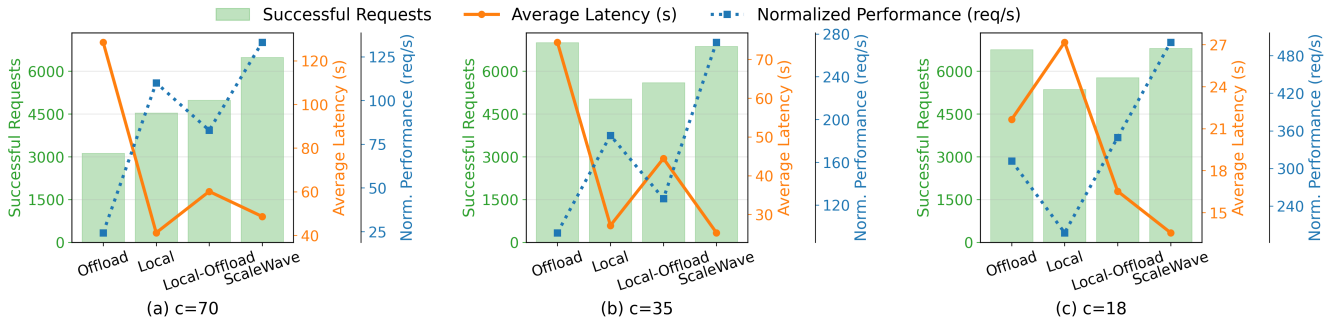


Fig. 7: Total Successful Requests VS. Average Latency VS. Performance over an hour interval for different concurrency values

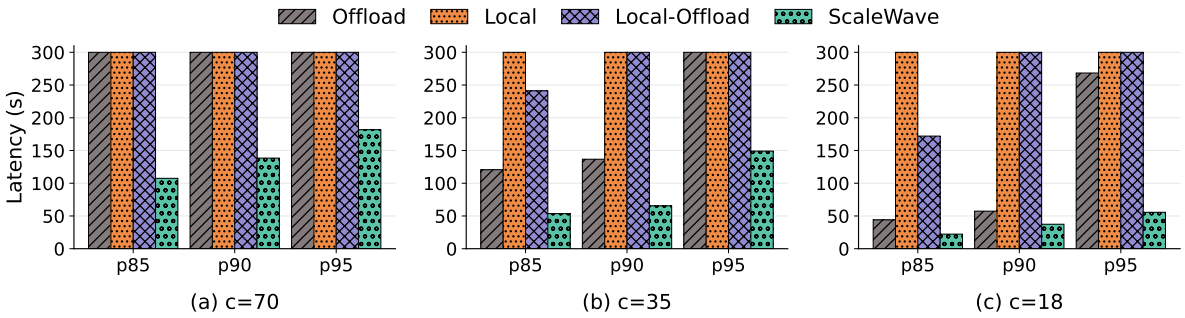


Fig. 8: Tail Latency Comparison for different concurrency values

resources. It achieves higher throughput at high concurrency and similar throughput at low concurrency compared to the offloading-only baseline. SCALEWAVE also maintains higher successful rates and lower latency than offloading-only and local-offloading baselines, with only slightly higher latency than edge at high concurrency. Overall, SCALEWAVE improves the normalized performance by up to 42%, serving up to 2x more requests with minimal latency (under 50 seconds), as compared with single-implementation based auto-scaling.

C. Resource Utilization Efficiency

We measured and aggregated resource usage data every second from all three nodes. The `psutil` [38] library was used to measure general resource utilization, while the `tegrastats` [39] command line utility measured GPU utilization on the Jetson Nano. We used average normalized resource utilization to obtain a single metric representing overall resource usage across different baselines. To obtain this metric, we first calculated the average utilization for each resource category. Then, we concatenated these averaged values across different baselines and identified the maximum value for each resource category. We computed the ratio of each average utilization to the maximum utilization to get normalized average usage. Finally, we calculated the mean of these normalized values.

Fig. 9 shows that using SCALEWAVE consistently achieves the highest average normalized utilization across different concurrency values, peaking at 78.34% for c=70. It showed a 42.87% improvement in average normalized resource utilization compared to offloading-only, 26.6% compared to local-only, and 22.26% compared to local-offloading. The offloading-only approach exhibits the lowest utilization, the local-only approach shows variable resource efficiency, and the

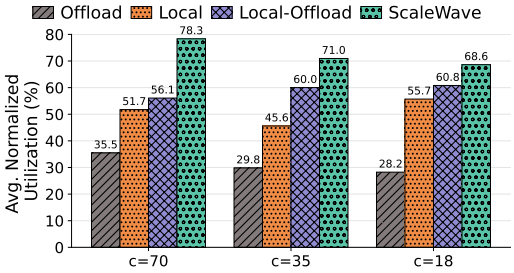


Fig. 9: Resource Utilization VS. Concurrency Values

local-offloading approach maintains high utilization but falls short of the SCALEWAVE approach. Overall, SCALEWAVE consistently achieves better resource utilization due to its access to various resource categories.

Moreover, we compared the maximum resource utilization when using SCALEWAVE with the other baseline approaches, as shown in Table V, varying concurrency values. We observed that SCALEWAVE consistently accesses resources not utilized by other baselines. For instance, at c=70, the offloading-only approach underutilizes edge CPU (76.13%) and memory (43.53%) but has the highest network upload bandwidth usage. The local-only and local-offloading approaches better utilize CPU and memory, with local-only CPU at 99.5% and memory at 81.67%, and local-offloading exceeding 90% for both. However, both underutilize network upload bandwidth compared to the offloading-only. None of the baselines use GPU resources due to implementation limitations, despite availability. SCALEWAVE shows a better balance, efficiently using CPU, memory, network upload bandwidth, and GPU resources, showcasing more efficient edge resource utilization.

Summary: SCALEWAVE achieves the highest average normal-

Conc.	Eqv. Impl.	CPU Usage (%)	Memory Usage (%)	Network Upload (bytes/s)	GPU Usage (%)
c = 70	Offload	76.13	43.53	6502413.00	0
	Local	99.50	81.66	2171108.66	0
	Local-Offload	98.10	92.23	830653.00	0
	ScaleWave	96.66	80.96	4371618.33	33.00
c = 35	Offload	87.80	34.70	5512313.67	0
	Local	100.00	91.60	682594.00	0
	Local-Offload	100.00	99.33	1126069.33	0
	ScaleWave	99.90	84.07	5714055.00	33.00
c = 18	Offload	95.87	43.43	6119026.00	0
	Local	100.00	99.07	1484092.67	0
	Local-Offload	100.00	99.07	1456626.67	0
	ScaleWave	99.93	92.83	6279394.00	33.00

TABLE V: Maximum Resource Utilization

ized resource utilization across different concurrency values, peaking at 78.34% for c=70. It efficiently utilizes all available edge resources in different dimensions, which other baselines underutilize. ScaleWave's gradual traffic adjustment mechanism ensures consistent performance, improving overall resource utilization.

D. ScaleWave *System Overheads*

Across our experiments, overhead from metrics collection, optimization, and routing updates is negligible relative to the orchestration frequency or per-request execution time (typically several seconds), enabled by isolating components in separate processes and maintaining atomic bookkeeping even under frequent reconfiguration. The Observer follows Knative's stable/panic monitoring windows (15s sampling in stable; 6s in panic when request volume rises 1.5x over the previous window), balancing responsiveness and overhead, and is configurable. ScaleWave further limits control-plane churn by updating traffic only when performance degrades and by using gradual rollouts to prevent oscillation.

Components	Latency	Overhead@Stable	Overhead@Panic
Observer	63ms	N/A	N/A
Optimizer	2ms	0.01%	0.03%
Routing Updates	50ms	0.3%	0.83%

TABLE VI: Latency Overhead

Table VI summarizes the measured latency overheads: "Overhead@Stable" and "Overhead@Panic" quantify on-critical-path latency overhead from ScaleWave 's control actions during stable and panic modes, respectively. These overhead values are calculated by dividing the component incured latency by the respective observer window (i.e., $overhead = component\_latency \div window\_size$). Because the Observer runs as a separate background process (off the orchestration critical path), we report its per-cycle latency (63 ms) and mark Overhead@Stable/Panic as N/A. In contrast, the Optimizer and Routing Updates execute on the orchestration/control-plane path when triggered, adding 2 ms and 50 ms latency respectively, with negligible overhead under both modes (0.01–0.03% for Optimizer and 0.3–0.83% for Routing Updates), thus enabling timely scaling within the observer window sizes and across resource-constrained micro-edge clusters with minimal resource footprint of 0.01% CPU and $\approx$ 4MB memory usage as presented in Fig. 10.

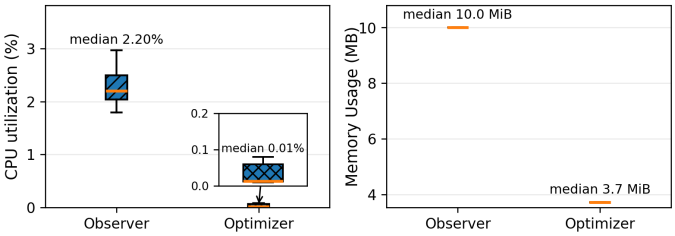


Fig. 10: Resource Overhead

E. *Threats to Validity:*

Although our study provides an end-to-end evaluation of ScaleWave, it has a few limitations.

(i) Application scope: We evaluate ScaleWave using a single motivating service (face recognition) with three equivalent implementations, which isolates the core mechanism of scaling across equivalents but may not capture trade-offs in other applications or implementation families.

(ii) Hardware and network environment: Results are obtained on one heterogeneous micro-edge cluster (Table III); different device mixes, resource ratios (CPU/GPU/memory), and LAN/WiFi/cellular conditions may change the relative benefits and reconfiguration frequency.

(iii) Workload realism: Due to limited availability of real edge workload traces, we use a synthetic request trace derived from the Shanghai Telecom dataset by sampling a base-station connection trace to obtain inter-arrival behavior and replaying it at different concurrency levels. While this captures volatility typical of small-population edge sites, it may not reflect diurnal effects, multi-service interference, or workload composition shifts in production.

(iv) Generalizability: Although our trace is base-station-derived, ScaleWave is deployment-agnostic with its profiling, resource load monitoring, and traffic redistribution mechanisms apply to any heterogeneous, resource-limited micro-edge setting (e.g., buildings, campuses, enterprise clusters). However, in resource-abundant environments (e.g., regional data centers with ample bandwidth and compute), the bottlenecks ScaleWave targets occur less frequently, and the gains are therefore expected to be smaller.

V. Conclusion

This paper introduced ScaleWave, a scalable serverless edge computing framework tailored for resource-constrained environments. By leveraging equivalent implementations—predefined variants of a function that fulfill the same semantics but utilize different resources—ScaleWave decouples application logic from specific resource bottlenecks. Our empirical evaluations demonstrate ScaleWave improves normalized performance by up to 42% and achieves 2× higher throughput under bursty edge workloads, all with consistent and efficient resource utilization.

VI. Acknowledgement

This research is supported by NSF through the grant #2104337.

REFERENCES

[1] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[2] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proceedings of the 30th international symposium on high-performance parallel and distributed computing*, 2021, pp. 239–251.

[3] P. Raith, S. Nastic, and S. Dustdar, "Serverless edge computing—where we are and what lies ahead," *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, 2023.

[4] M. Tari, M. Ghobaei-Arani, J. Pouramini, and M. Ghorbian, "Auto-scaling mechanisms in serverless computing: A comprehensive review," *Computer Science Review*, vol. 53, p. 100650, 2024.

[5] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," in *14th IEEE/ACM International Conference on Utility and Cloud Computing*, 2021, pp. 1–8.

[6] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *The Computer Journal*, vol. 62, no. 2, pp. 174–197, 2019.

[7] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: survey, trends and future directions," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 399–432, 2019.

[8] A. Koubaa, A. Ammar, A. Kanhouch, and Y. AlHabashi, "Cloud versus edge deployment strategies of real-time face recognition inference," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 1, pp. 143–160, 2021.

[9] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *2017 IEEE ICDCS*. IEEE, 2017, pp. 276–286.

[10] Y. Yamato, T. Demizu, H. Noguchi, and M. Kataoka, "Automatic gpu offloading technology for open iot environment," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2669–2678, 2018.

[11] Knative, "Overview - knative," 2024. [Online]. Available: https://knative.dev/docs/serving/

[12] S. Wang, Y. Zhao, J. Xu, J. Yuan, and C.-H. Hsu, "Edge server placement in mobile edge computing," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 160–168, 2019.

[13] Oak Ridge National Laboratory, "Iris: From edge to exascale," https://iris-programming.github.io/, 2025, accessed: 2025-09-14.

[14] A. Pawar, "Nnapi explained: The ultimate 2025 guide to android's ai acceleration," *softAai Blog (Medium)*, 2025. [Online]. Available: https://medium.com

[15] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *MobiSys'10*, 2010, pp. 49–62.

[16] X. Liu *et al.*, "Collaborative task offloading and service caching strategy for mobile edge computing," *Sensors*, vol. 22, no. 22, p. 8618, 2022.

[17] M. N. Fekri, K. Grolinger, and S. Mir, "Distributed load forecasting using smart meter data: Federated learning with recurrent neural networks," *International Journal of Electrical Power & Energy Systems*, vol. 137, p. 107669, 2022.

[18] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 180–189.

[19] S. Wang, X. Liu, M. Gao, M. Chen, K. L. Yung, and S. Jiang, "Multi-objective auto-scaling scheduling for micro-service workflows in hybrid clouds," *Enterprise Information Systems*, vol. 17, no. 7, p. 2069478, 2023.

[20] K. Ray and A. Banerjee, "Horizontal auto-scaling for multi-access edge computing using safe reinforcement learning," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 6, pp. 1–33, 2021.

[21] T. P. da Silva, A. R. Neto, T. V. Batista, F. C. Delicato, P. F. Pires, and F. Lopes, "Online machine learning for auto-scaling in the edge computing," *Pervasive and Mobile Computing*, vol. 87, p. 101722, 2022.

[22] X. Ma, K. Zong, and A. Rezaeipanah, "Auto-scaling and computation offloading in edge/cloud computing: a fuzzy q-learning-based approach," *Wireless Networks*, vol. 30, no. 2, pp. 637–648, 2024.

[23] J. Dogani and F. Khunjush, "Proactive auto-scaling technique for web applications in container-based edge computing using federated learning model," *Journal of Parallel and Distributed Computing*, vol. 187, p. 104837, 2024.

[24] K. Cheng, S. Zhang, C. Tu, X. Shi, Z. Yin, S. Lu, Y. Liang, and Q. Gu, "Proscale: Proactive autoscaling for microservice with time-varying workload at the edge," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1294–1312, 2023.

[25] H. Ko and S. Pack, "Function-aware resource management framework for serverless edge computing," *IEEE Internet of Things Journal*, vol. 10, no. 2, pp. 1310–1319, 2022.

[26] A. Raza, N. Akhtar, V. Isahagian, I. Matta, and L. Huang, "Configuration and placement of serverless applications using statistical learning," *IEEE Transactions on Network and Service Management*, 2023.

[27] Z. Wu, Y. Deng, Y. Zhou, J. Li, and S. Pang, "Faasbatch: Enhancing the efficiency of serverless computing by batching and expanding functions," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 372–382.

[28] Z. Wu, Y. Deng, Y. Zhou, J. Li, S. Pang, and X. Qin, "Faasbatch: Boosting serverless efficiency with in-container parallelism and resource multiplexing," *IEEE Transactions on Computers*, 2024.

[29] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 78–93.

[30] K. Luo, T. Ouyang, Z. Zhou, and X. Chen, "Beeflow: Behavior tree-based serverless workflow modeling and scheduling for resource-constrained edge clusters," *Journal of Systems Architecture*, vol. 143, p. 102968, 2023.

[31] M.-N. Tran and Y. Kim, "Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing," *Future Generation Computer Systems*, vol. 152, pp. 304–316, 2024.

[32] H. Li, H. Liu, A. Chen, X. Ma, Q. Liu, and J. Du, "Ria: Return on investment auto-scaler for serverless edge functions," in *ICPP*, 2024, pp. 772–781.

[33] Knative, "Revisions - knative," 2024. [Online]. Available: https://knative.dev/docs/concepts/serving-resources/revisions/

[34] ——, "Traffic splitting - knative," 2024. [Online]. Available: https://knative.dev/docs/getting-started/first-traffic-split/

[35] B. Yang, A. Sailer, S. Jain, A. E. Tomala-Reyes, M. Singh, and A. Ramnath, "Service discovery based blue-green deployment technique in cloud native environments," in *2018 IEEE international conference on services computing (SCC)*. IEEE, 2018, pp. 185–192.

[36] D. Ernst, A. Becker, and S. Tai, "Rapid canary assessment through proxying and two-stage load balancing," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2019, pp. 116–122.

[37] Ageitgey, "Github - ageitgey/face_recognition: The world's simplest facial recognition api for python and the command line," 2024. [Online]. Available: https://github.com/ageitgey/face_recognition

[38] "psutil documentation — psutil 6.0.0 documentation," 2024. [Online]. Available: https://psutil.readthedocs.io/en/latest/

[39] NVIDIA, "tegrastats utility," 2024. [Online]. Available: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html