FaaSt: A Latency-aware Serverless Scheme for Edge–Cloud Environments

Kien Nguyen*, Hiep Dao†, Tung Nguyen†, Frank Loh*, Nguyen Huu Thanh†, Tobias Hoßfeld*
*University of Würzburg, Institute of Computer Science, Würzburg, Germany
†Hanoi University of Science and Technology, Hanoi, Vietnam
Corresponding email: kien.nguyen@uni-wuerzburg.de

***Abstract*—Serverless computing offers an effective way to improve resource utilization and can significantly leverage edge–cloud environments. In this context, serverless request response times are one of the main focuses for optimization. However, the end-to-end routing path from a request to a function, which considerably contributes to the overall delay, is often overlooked. This oversight raises concerns about the actual performance of serverless in edge–cloud deployments, as serverless architectures are initially designed for cloud computing where node-to-node delay is negligible. To address this concern, our paper examines the current serverless networking design in a realistic edge–cloud setup. Our findings reveal significant performance degradation when serverless is applied in such environments due to key mismatches in the networking architecture. To mitigate these issues, we propose a novel latency-aware scheme and a load balancing mechanism for edge–cloud environments, called FaaSt. This proposal is implemented using open sources and standards, resulting in a significant reduction in response times.**

***Index Terms*—serverless computing, latency, edge–cloud**

I. Introduction

Recently, serverless computing has gained popularity in the field of edge–cloud computing, a paradigm that combines proximity to data sources at the edge with abundant resources of the cloud. In this scenario, serverless computing enhances the utilization of scarce edge resources with the introduction of request-driven computing, which ensures that resources are used only when needed while providing unified and automated resource management for heterogeneous environments. This synergy is well-suited for the dynamic workload characteristics of Internet of Things (IoT) ecosystems, increasing the performance, scalability, and efficiency of event-driven IoT models such as Industrial IoT, and connected vehicles [1].

One of the keys to realize serverless computing in the edge–cloud paradigm is to optimize end-to-end latency. This can be approached by either enhancing the operation of serverless functions, such as reducing cold start times [2] or from a broader perspective, such as managing request routing in a multi-cluster system [3]. However, serverless computing was initially developed and predominantly used in cloud data centers, where resources are abundant and latency between peers is negligible. Integrating serverless into the heterogeneous and limited-capability edge–cloud may raise concerns about platform compatibility. In this context, the internal networking design of current serverless architectures, optimized for cloud environments, may not perform well in distributed setups like the edge–cloud. Consequently, the actual latency experienced by users or machines may significantly differ from theoretical models. To the best of our knowledge, there is no existing literature that addresses this issue comprehensively. To fill this gap, we tackle the following three research questions (RQ):

RQ1: What is the real response time of serverless functions when being hosted on a distributed system such as an edge–cloud?

RQ2: What are the influencing factors that contribute to the response time?

RQ3: How can we improve the response time of serverless function in the edge–cloud?

To answer the research questions, we develop a real edge–cloud testbed for serverless deployments using well-known open-source tools and open APIs. Our contributions are three-fold. First, we investigate the response time of serverless deployments in the edge–cloud environment by conducting measurements on a real testbed. Second, we identify the influential factors in the architectural design that contribute to latency. Third, based on the identified weaknesses, we propose a novel latency-aware scheme for serverless computing in edge–cloud environments. This scheme considers latency-contributing factors such as network delay and sojourn time to forward requests on the least delay path. The proposed scheme is implemented using open sources and standards, enabling other researchers to reproduce and further optimize the solution not only within the edge–cloud context but also in distributed systems such as multiple datacenters (multi-DCs) or multi-access edge computing (MEC). The source code can be found in our Github repository [4].

The remainder is structured as follows. Section II summarizes the important background and highlights the current research in the area. Subsequently, Section III introduces the testbed, the evaluated test cases, and measurement results. To mitigate the identified problems, Section IV describes the proposed scheme along with a load-balancing algorithm. Finally, Section V concludes the works and opens questions for future research.

II. Backgrounds and Related Works

This section provides essential knowledge about serverless in general, the networking design of current open-source serverless platforms, and related works.

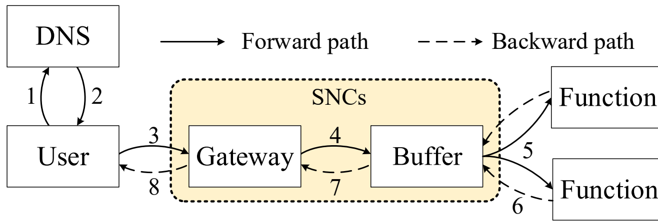


Fig. 1: **Logical perspective:** The forwarding path from user to function and backward in the popular serverless platforms. The numbers indicate the sequential steps for the request.

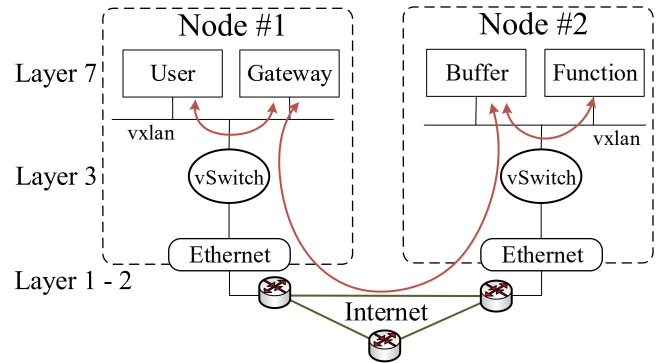


Fig. 2: **Network stack perspective:** The forwarding path, visualized as red arrows, from the user to function. The SNCs, introduced by the popular serverless platforms, belong to layer 7 in the OSI model. Virtual switches of layer 3 are created and managed by K8s.

A. Serverless Computing Networking Design

Serverless computing has gained popularity since the introduction of Amazon's Lambda [5]. It helps to reduce resource usage by activating functions only when requests arrive, which is known as scale-from-zero, thus minimizing costs for operators and users. Besides commercial products from Amazon or Microsoft, well-known open-source serverless platforms, such as Knative [6], OpenFaaS [7], and OpenWhisk [8], are widely used. These platforms typically run serverless functions in containerized environments managed by Kubernetes (K8s) to inherit the K8s' carrier-grade qualities, such as high availability, scalability, and fault tolerance. However, K8s was initially designed for cloud environments with reliable and low-latency networks. As a result, it may face challenges in heterogeneous systems like edge–cloud environments [9], potentially impacting these serverless platforms.

Our analysis reveals that these popular open-source serverless platforms share a similar application-layer networking design. A request from an end-user or device must traverse several networking components, known as Serverless Networking Components (SNCs), which typically consist of two parts: the Gateway and the Buffer [10]–[12]. These components are responsible for request redirection, load balancing, and enabling the scale-from/to-zero capability, a fundamental aspect of serverless computing. Fig. 1 describes the details of the forwarding path as follows.

Initially (paths 1 and 2), typical serverless HTTP/HTTPS requests require a DNS query to resolve the gateway IP address. CoreDNS, a K8s function, handles this resolution. To minimize latency from repeated DNS queries, the DNS-IP information can be cached locally. However, because IP addresses may change, a DNS query is periodically required, usually every 30 s, to obtain the updated address. When requests reach the Gateway (path 3), they are identified as serverless requests and classified by the type of service they require. The Gateway also functions as a bridge between the internal cluster and the external Internet. Multiple Gateways can exist within the cluster, acting as entry points for requests. Once identified and classified, requests are routed from the Gateway to Buffer (path 4). The Buffer functions as a request queue, temporarily storing requests while waiting for a function to spawn. This mechanism is the key to realizing the scale-from/to-zero mechanism of serverless [13]. Once functions are ready, the Buffer forwards queued requests to each function following a load-balancing strategy (path 5).

From a logical perspective, the Gateway and the Buffer can be combined into a single unified block, referred to as SNCs throughout this paper. However, leading open-source serverless platforms keep these components separate to take advantage of existing solutions rather than developing new ones from scratch [10]. For instance, Knative uses Envoy Ingress as Gateway and developed its own Buffer called Activator [11]. Similarly, OpenWhisk uses Nginx as the Gateway and Kafka as the Buffer [10], while OpenFaaS employs NATS Streaming as its Buffer [12]. Despite different approaches in implementation, SNCs of these popular platforms are components that belong to layer 7 in the OSI model, as described in Fig 2. Their introduction adds an additional path, on top of the traditional K8s' networking, that requests must traverse before reaching the functions. Consequently, the placement of SNCs can significantly impact end-to-end latency, particularly in edge-cloud environments.

B. Related Works

The integration of serverless computing into the edge–cloud domain has become a timely and relevant topic [3]. Key optimized metrics in this context are either resource efficiency [14] or requests' response time. To tackle requests' high response time in heterogeneous networks, K8s offers region-aware routing through Topology Aware Routing [15], while Istio, a more granular service-oriented networking layer for K8s microservices, provides similar functionality [16]. Both systems use a simple approach: forwarding requests to the local region first, and only to others when the local region is overloaded. This rigid mechanism lacks real-time tracking of latency variations, making it unable to adapt to real-time latency changes. Cicconetti et al. [17] discuss non-optimal request routing and propose a novel architecture with a cost optimization algorithm for traffic flows. Chen et al. [18] introduce a system for edge-cloud environments that uses

TABLE I: Testbed instruments. All nodes are VMs created from a server with the following configuration: Dell PowerEdge T440 (CPU Intel Xeon Silver 4210R 2.40GHz - 10 cores 20 threads, RAM 48GB DDR4).

Nodes	Software
Master: 6 core 12GB	Knative v1.12 [6], Kubernetes v1.26 [24]
Server: 6 core 12GB	Locust v2.29 [25], Wireshark v4.0 [26]
Edge: 5 core 8GB	Prometheus v59.0 [27], Linux Tc [28]

a probability-based algorithm to decide function types and locations based on cold start latency and resource availability. High response times in serverless-based microservices deployed on edge-cloud environments are addressed in [19] by routing traffic smartly between services at the application layer. Similar to our approach, Mittal et al. [20] modify Knative for a heterogeneous system, considering application response time. However, the authors do not account for the network delay between the edge and the cloud, which is addressed in this paper. Russo et al. [21] propose Serverledge, a decentralized FaaS platform that improves the performance of applications in distributed systems. However, as a simple prototype using Docker containers, it lacks robust mechanisms (e.g., high availability and scalability) and is not applicable in large deployment scenarios compared with platforms built on K8s [22].

To the best of our knowledge, current networking-related research focuses on optimizing end-to-end delay via either function placement or layer 2 routing. The latter is a general networking issue rather than one specific to serverless computing. Existing works do not address latency within the serverless platform itself, particularly the latency caused by traffic passing through its components. In one of our previous studies [23], we observed an interesting result: the response time of serverless deployments on edge devices was higher than that of remote deployments, regardless of processing power—contrary to expectations. This observation led us to the research questions of this paper. Our focus is on the internal networking design of serverless platforms, particularly on how the positioning of layer 7 SNCs (Fig. 2) affects request response times in edge-cloud environments.

III. Serverless Response Time Investigation

In this section, we examine the real response times of serverless requests. This involves conducting a detailed investigation into the internal architecture of an open-source serverless. While the testbed focuses specifically on the edge-cloud environment, the investigated results can be interpreted for other distributed environments (MEC, multi-DCs), where high latency between peers exists.

A. Testbed Description

The testbed for this work consists of four VMs configured as a serverless cluster using Knative on top of K8s. The cluster uses Calico [29] for networking. In this K8s-based setup, the VMs are divided into Worker and Master roles, serving as the data plane and control plane, respectively, as shown in Fig. 3. The Worker nodes consist of three nodes with different computing capacities: two high-capacity VMs representing cloud nodes and one lower-capacity VM representing the edge device. The latency between the two cloud nodes is stable and low ($<1\,\text{ms}$) emulating a data center environment. In contrast, the connection between the edge device and the cloud nodes is manipulated by Linux Traffic Control (TC) [28] to emulate the edge–cloud network. The latency between the edge and cloud nodes is set to three values: $50 \pm 10\text{ms}$, $70 \pm 10\text{ms}$, and $100{\pm}20\text{ms}$ with the latter number indicates the jitter following a normal distribution [30]. These configurations reflect a typical short-to-long path between users and data centers [31]. Details about the testbed instruments are presented in Table I.

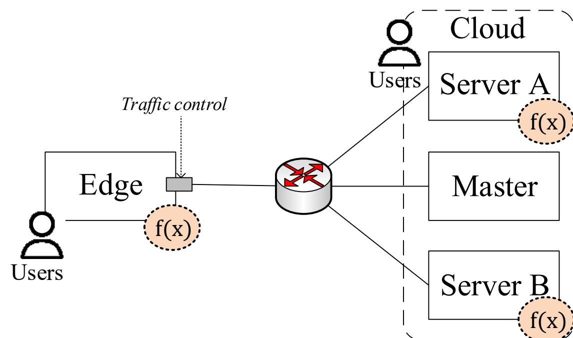


Fig. 3: Testbed diagram.

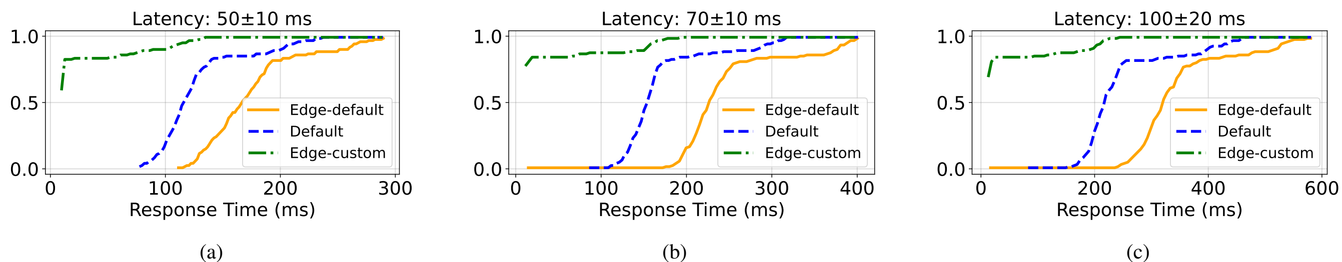
In Fig. 3, *Users* represent the origin of requests, which can come from a real user, a machine/device, or even another function within the cluster. The function, denoted as $f(x)$ can reside on any computing node across the platform. Once the data is processed, the result is sent back to the request origin. Thus, the user serves as the Ingress and Egress (In/Egress) point for the request. This testbed emulates a unified serverless platform, seamlessly transparent from edge to cloud, allowing applications to effectively utilize resource sharing between the two computing tiers.

B. Testing Scenarios

As discussed in Section II-B, the lack of research on the influence of platform networking design raises concerns about the response time perceived at In/Egress points. To address this, we conducted a series of test cases, ranging from the default setup to experiments involving the reconfiguration of SNCs. In most cases, the positions of the Ingress/Egress points, SNCs, and functions are reconfigured to examine the latency experienced from Ingress to Egress and to identify the root causes of latency issues. A summary of each component's physical location in the testbed (on the edge device, cloud server, or both) according to each test case is shown in Table II. The primary measurement focus is the request's response time.

TABLE II: Position of each component in the path from the end user (In/Egress) to function according to test cases.

Components \ Cases	Default	Edge–default	Edge–custom	Edge–duplicated	Cloud–duplicated	Proposal
In/Egress	edge	edge	edge	edge	cloud	both
SNCs	cloud	cloud	edge	both	both	both
Function	cloud	edge	edge	both	both	both



(a) (b) (c)

Fig. 4: Comparison of latencies between the default SNCs location (Edge-default and Default) and the edge-placed SNCs case (Edge-custom) under varying edge–cloud latencies as CDFs.

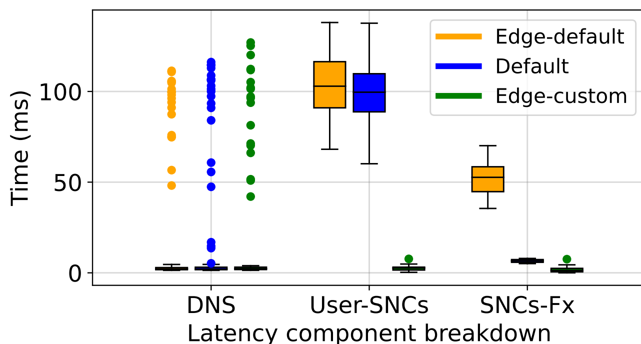


Fig. 5: Latency breakdown between the default SNCs location (Edge-default and Default) and the edge-placed SNCs case (Edge-custom).

In all test cases, the serverless function is a simple web application that returns an HTTP message when triggered, requiring only $3\,\mathrm{ms}$ to $5\,\mathrm{ms}$ for processing on both edge and cloud devices. This ensures that latency caused by the application processing time is minimized, allowing us to focus on the network-induced latency.

C. Networking Design Problem of Serverless in Edge–Cloud

First, we conduct a simple test scenario where a single user, emulated by Locust traffic generator [25], from the edge sends requests every $2\,\mathrm{s}$ for $5\,\mathrm{min}$. We perform two test cases over the native Knative platform: Default and Edge-default. The suffix *Default* indicates that the SNCs remain unmodified and distinguishes this scenario from the *custom* system introduced later. In the Default case, the serverless platform autonomously determines the function's location, which normally places functions at the powerful node, which in this case is one of the cloud nodes. Whereas in the Edge-default case, the function is manually set to reside at the edge device. Theoretically, the Edge-default case is expected to yield lower latency than the cloud case since it avoids the propagation delay.

1) Latency of Serverless in Edge–Cloud: Fig. 4 shows the Cumulative Distribution Function (CDF) of the response times under different emulated latencies. For now, we focus only on Default (blue dotted line) and Edge-default (yellow solid line), the customized case Edge-custom is introduced later. It is observed that higher network latency induces a higher average response time while the trend remains unchanged. Despite the function's proximity, the Edge-default always exhibits higher average latency compared to Default, which places the function in the cloud. Specifically in the case of Fig. 4–a, nearly 80% of all requests in the Default case are responded back within $150\,\mathrm{ms}$, which is $200\,\mathrm{ms}$ for Edge-default. Additionally, about 20% of requests in both configurations experience an additional $100\,\mathrm{ms}$ latency, indicating an intermittent factor causing high latency. The imbalance in processing time between the two tiers has been eliminated by using a simple function, suggesting that there is a network-related factor in the Edge-default configuration that causes the increased latency.

Looking into the components contributing to the total response time, Fig. 5 breaks down the latency for each segment of the route from Ingress to the serverless function. As the trend remains in different network latencies, we report only the 50 ± 10ms case. We focus on three key segments that contribute to the total network delay: the DNS query, the path from the user to the SNCs (User-SNCs), and the path from the SNCs to the function (SNCs-Fx). The figure shows that while the edge and cloud cases exhibit similar behavior in DNS latency and the User-SNCs path, the SNCs-Fx path

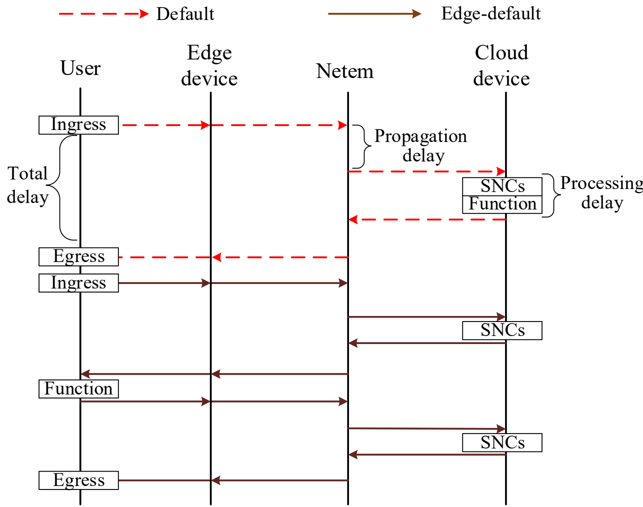


Fig. 6: Trace of routing path in two cases, Default and Edge–default. Occasionally, both cases require one more round trip to the DNS server situated on the cloud side to retrieve the updated IP addresses.

for the Edge-default case is significantly longer than that for the cloud. This likely causes the high latency observed in the Edge-default case. Additionally, although most DNS queries are resolved in less than 1 ms, there are many outliers ranging from 50 ms to 100 ms, leading to a number of requests experiencing high latency, as mentioned previously. This is caused by the periodic DNS query from the User to the DNS server on the cloud side to renew the destined IP address.

The results lead us to answer the first research question RQ1: *The real response time of requests placed in a serverless-enabled edge–cloud suffers from unexpectedly high latency, with edge latency surpassing cloud latency even without accounting for any processing time.*

2) The Routing Path of Serverless: To understand the problem, we trace the physical route from In/Egress to the function in the Default and the Edge-default case using Wireshark. In our testbed, the DNS server is located at the Master node in the cloud, requiring the Ingress to traverse a long distance to obtain the address. Once the DNS is resolved, the subsequent physical route for each case is illustrated in Fig. 6.

In the Default configuration, user requests are routed first to SNCs, which are located in the cloud and, therefore, primarily contribute to network delay. The path from the SNCs to the function is negligible since they are co-located in the cloud. After processing, the result returns to the Egress via the same route. On the other hand, the Edge-default, instead of routing directly to the local device, requests must traverse the lengthy edge–cloud path to reach the SNCs, then loop back to the edge to access the function. This unnecessary loop doubles the latency in the Edge-default, even though it should theoretically have a lower latency than the Default case.

This issue arises from the non-latency-aware design of serverless platforms that do not place SNCs, which are responsible for network routing, in optimal positions. Since most serverless platforms are built on top of K8s, the SNCs are treated as normal K8s deployments. Thus, their positions are determined by the K8s scheduler, which typically favors nodes with high computing power. While this design may not impact connections within a local network, such as in a DC, it creates problems in heterogeneous systems such as edge–cloud. In these environments, SNCs often end up on the powerful cloud side, causing the looping issue when functions are placed at the edge.

3) Optimal Position for Serverless Networking Components: We address the identified problem by relocating the SNCs from the cloud to the edge device. This is achieved by customizing the SNCs manifest file. With the SNCs now positioned on the same side as the function, the edge–cloud path latency should be eliminated (except the DNS query). This configuration is called Edge-custom and is compared to the previous cases, Edge-default and Default. For this evaluation, the requests are generated in the same manner as before, and the results are shown by a green dotted line in Fig. 4.

As observed, the latency of Edge-custom is significantly lower in all cases, with 75% of all requests below 20 ms. The results reflect the true edge scenario when requests are processed locally without traversing through the external network since the network loop has been eliminated. The Edge-custom also shows signs of impact as 25% of the requests' latency increases along the backbone latency, which is explained by looking at the breakdown latency in Fig. 5. While Edge-custom has almost eliminated the User-SNCs and SNCs-Fx latency, the DNS mechanism still requires periodic queries to the DNS server residing on the cloud side. Thus, around one-fourth of the requests suffer from higher latency than the rest.

Placing SNCs at the edge can solve network looping when the function is also at the edge. However, in real scenarios like microservices, requests often come from the cloud side, not just edge users. If the network gateway is at the edge, while the In/Egress and functions are in the cloud, the loop reappears, causing high latency. Thus, placing SNCs at the edge alone does not fully resolve the issue.

D. The Load Balancing Problem in Edge–Cloud

As a solution, multiple SNCs design, which places SNCs at each computing tier, is implemented. Evaluation is conducted again with two In/Egress points on both sides uniformly generating requests. By providing each computing tier with its own SNCs, the expectation is that the looping problem will be eliminated, as requests will be routed to their local SNCs and functions. The measurement results, shown in Fig. 7, compare the Edge-duplicated and Cloud-duplicated cases based on the response times perceived by the Egress at the edge and cloud, respectively. These are compared to the local processing case (Edge-custom) and the looping case (Edge-default) introduced earlier.

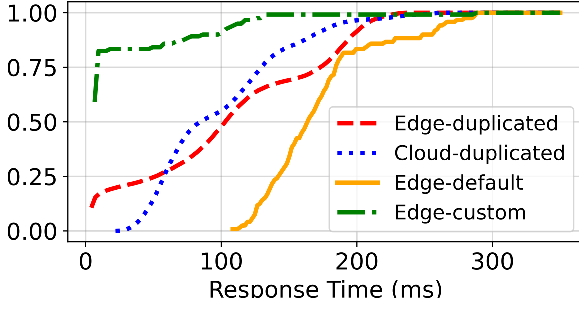


Fig. 7: Comparison between the single SNCs (Edge-default, Edge-custom) case versus the multiple SNCs cases (duplicated) where SNCs are placed at both computing tiers.

Contrary to expectations that Edge-duplicated and Cloud-duplicated would perform similarly to Edge-custom, their performance, while better than the looping case, does not match the local processing case. This discrepancy arises because, even though SNCs are distributed across each tier, traffic is not routed internally within its region but is evenly distributed among functions placed in both the edge and the cloud, which is done by the default serverless Load Balancing (LB) mechanism. In distributed system environments like edge–cloud, the LB is unaware of local functions and may forward requests to distant functions. For example, a user request originating from an edge device may be routed to the local gateway but then load-balanced to a function located in the cloud. In such cases, the advantage of proximity to the data source in the edge–cloud model is not utilized.

From the results, the second research question, RQ2, is answered as follows: *In the scope of our investigation, there are two main issues contributing to the high response time of serverless deployments in an edge–cloud environment. First, the placement of SNCs is not optimized for latency. Current FaaS platforms lack mechanisms to strategically place SNCs that requests from all nodes in distributed environments can benefit. Second, the LB mechanism for traffic between SNCs and the function is also problematic for an edge–cloud setup, as it prioritizes computing power over request response time.*

IV. A Latency-aware Scheme for Edge–Cloud Serverless Computing

This section introduces FaaSt, a latency-aware scheme for serverless deployments in edge–cloud environments that is implemented on a real testbed using open-source tools and APIs. Evaluated results, in comparison with the default serverless platform, are presented at the end.

A. Ideas and Implementation

The proposed scheme is illustrated in Fig. 8. This figure represents a simplified topology of a system with geographically distributed computing devices, where latency between peers varies. Requests can originate from end users or other functions located anywhere in the topology. These requests can be served by functions initiated on any computing device.

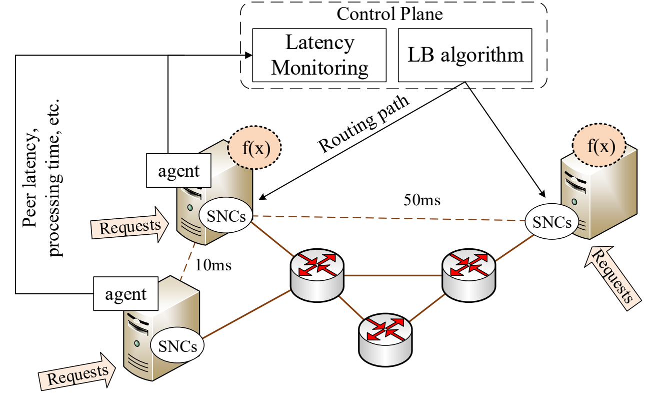


Fig. 8: The proposed FaaSt scheme for serverless deployments in edge–cloud.

Algorithm 1 FaaSt Load Balancing Algorithm

Input: latency metrics
Output: forwarding policies

1 **while** *every t seconds* **do**
2   Collect network delay matrix D, where d_{ij} is the delay from node i to j
    Collect average processing time P of functions placed at node i
    Calculate weighted matrix W from D and P
    Broadcast the weighted matrix to all SNCs
3 Forward requests from node i to node j following w_{ij}
4 Forward requests to function in node j following round-robin

To achieve the latency objective for a unified serverless system in an edge–cloud environment, FaaSt is designed with SNCs distributed across every device or within each local network cluster. This is done by deploying the Gateway-Activator pair using *DaemonSet* provided by K8s. The system is then configured to forward requests from one device or local cluster to its local SNCs, avoiding the unnecessary latency caused by long paths between Ingress and SNCs. To achieve this, we set the *internalTrafficPolicy* parameter in the *kourier* service, which is responsible for managing HTTP traffic routing between SNCs, to *Local* and modify the *kourier*'s source code to guide the Gateway to forward requests to the Activator on the same node. However, as demonstrated in Section III–D, this mechanism only addresses a part of the problem.

To address the load balancing issue, a latency-aware LB mechanism for SNCs is proposed. The objective of this LB mechanism is to minimize the response time of requests by considering two key factors: network delay between nodes and the sojourn time that requests spend on each device. The LB mechanism may prioritize forwarding requests to local functions when it determines that the network delay to a distant node would significantly exceed the low processing time at the local node. Conversely, when local node resources

are insufficient or the request queue is full, resulting in high processing times, the algorithm may forward requests to distant nodes if they offer a lower overall response time despite the network delay.

To retrieve knowledge of the current network delay and processing time, FaaSt employs a monitoring agent installed on every node to collect these metrics, as illustrated in Fig. 8. Network delay is measured as round trip time (RTT) using ping, while processing delay is monitored using Prometheus [27] and Knative API. These metrics are periodically collected by a Golang-based controller in the control plane via the APC API. The load balancing algorithm is implemented by replacing the default *randomChoice2Policy()* function in the Knative serving source code, which selects two random functions and forwards requests to the one with the fewest connections, with our custom *FaaStPolicy()*. Periodically, the LB queries latency information from the Controller to update its forwarding policies. Details about the proposed FaaSt LB mechanism are presented in Algorithm 1, with the assumption that functions are already available in the cluster.

After every predefined period of time (t seconds), FaaSt updates the average latency experienced by current requests based on the information collected from the agents. The average RTT among devices, measured by the agents and stored at the metric server as matrix D.

$$D = \begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ d_{i1} & d_{i2} & \cdots & d_{ij} \end{pmatrix}$$

$$d_{ij} \geq 0\,(\text{ms}), \quad i, j \in \{1, \ldots, N\}$$

With N as the number of computing nodes, d_{ij} represents the network delay of requests originating from computing node i to reach the functions residing in computing node j. D is a hollow symmetric matrix with the diagonal line (e.g. d_{11}, d_{22}) equals zero since network delay does not exist for the device itself. Furthermore, $d_{ij} = d_{ji}$ when $i = j$ as they are RTTs of the same pair. Then, the average processing time of requests that have been processed on each node during the last t seconds is represented by matrix P.

$$P = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ p_{i1} & p_{i2} & \cdots & p_{ij} \end{pmatrix}, \; p_{ij} \geq 0(ms)$$

p_{ij} represents the average time taken for requests originating from node i to be processed on node j. Since the request processing time is independent of the originating node i, $p_{1j} = p_{2j} = \ldots = p_{ij}$ for all $j \in \{1, \ldots, N\}$. This means that the values of the elements in each column are equal. The matrix is still represented in a large size to match the dimensions of matrix D for later calculation.

The average response time for requests originating from node i and served on node j during the last t seconds is estimated by summing matrices D and P to obtain matrix R.

$$R = D + P$$

Matrix W is derived from matrix R, which is a weighted distribution matrix that guides the LB in allocating requests to the appropriate computing nodes. Its element w_{ij} is inversely proportional to the estimated response time r_{ij} multiplied by 100% following this formula.

$$w_{ij} = \frac{r_{ij}^{-1}}{\sum_{k=1}^{N} r_{ik}^{-1}} \times 100(\%), \;\; \forall i, j \in \{1, \ldots, N\}$$

w_{ij} represents the percentage likelihood that requests originating from node i will be distributed to node j. In this context, nodes that are far from i (high d_{ij}) and overloaded (high p_{ij}) will have a low w_{ij}, resulting in a reduced probability that requests from i will be assigned to those nodes.

The weighted matrix W is then broadcasted to all SNCs in the cluster. Upon receiving requests, each node i will follow the probability rule of its own w_{ij} for all $j \in \{1, \ldots, N\}$ to forward them to one of the N nodes. If the destination node hosts multiple functions, the specific function will be selected in a round-robin manner.

The proposed FaaSt scheme is available in our GitHub repository [4], where researchers can find installation instructions for reproduction and further optimization or customization.

B. Evaluation

In the evaluation, we compare our proposed scheme with the default setup of Knative named our Baseline in the following. Requests are randomly generated from both computing sides with ascending levels of concurrent users, from 5 to 50 in steps of 5. Each user is sending requests at the rate of 10 requests per second. In all cases, approximately 200 000 requests are generated. This approach tests the behavior of the proposed scheme under low and high load conditions. It is important to note that the serverless system may automatically scale functions based on the load. However, the focus of this evaluation is not on the autoscaling mechanism, so both cases use the native autoscaler mechanism provided by K8s. The scenario is also evaluated under increasing latency, similar to Fig. 4: 50 ms, 70 ms, and 100 ms.

General Results: Fig. 9 shows the latency comparison between our proposal and the default Knative setup as CDF. We report solely the lowest and highest load/latency scenarios as the results from middle cases illustrate the transition phase between low and high loads. During low load, as seen in Fig. 9–a and Fig. 9–b, the proposed FaaSt significantly outperforms the Baseline, reducing peak latency by multiple folds. It also shows that increasing network delay severely impacts the Baseline, whereas it shows almost no impact on FaaSt, which is aware of the high edge–cloud latency and decides to route requests to the edge device. When the load increases, the gap between the two cases narrows. FaaSt shows a flatter curve, while the Baseline shows less variation. This can be explained by the fact that under high load, as the processing time at the edge node increases, FaaSt must

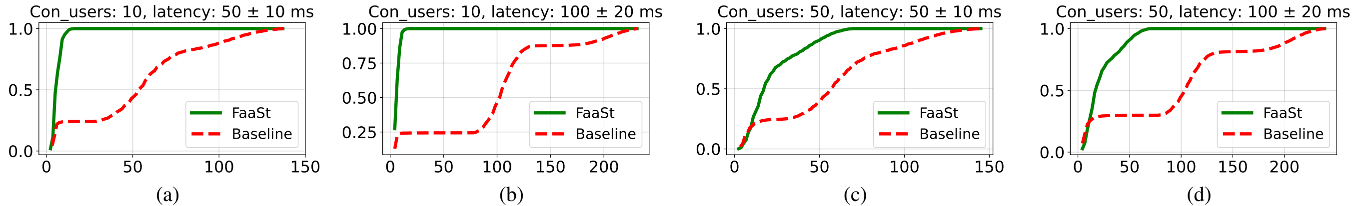


Fig. 9: Latency comparison between FaaSt and the default Knative system as CDF under varying number of concurrent users (Con_users) and network delay. The x-axis indicates the request response time in milliseconds.

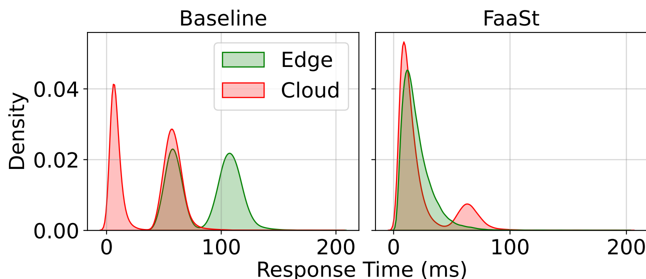


Fig. 10: Density distribution of requests' response time processed by each computing tier in the **Baseline** (left) and **FaaSt** (right). The request completion rate within the measurement period is 75% and 90%, respectively.

redirect a portion of requests to the cloud, thus increasing the average response time.

The reason FaaSt outperforms the Baseline is illustrated in Fig. 10, which show the density distribution of requests served by the Baseline and FaaSt systems according to the scenario in Fig. 9–c. The Baseline system, which prioritizes computing power, processes more requests in the cloud than in the edge. A high density of requests processed in the cloud within the latency range of $50\,\mathrm{ms}$ to $100\,\mathrm{ms}$ indicates that many requests originating from the edge must traverse to the cloud. In the same range, a high density of edge-processed requests suggest they are routed to different side SNCs or functions. The worst-case scenario, with latencies ranging from $100\,\mathrm{ms}$ to $150\,\mathrm{ms}$, points out that these requests have traversed in a loop.

In contrast, FaaSt shows that the majority of requests are distributed in the low latency range ($<50\,\mathrm{ms}$), resulting from local routing and processing. An exception is observed in the response time range of $50\,\mathrm{ms}$ to $100\,\mathrm{ms}$ for requests processed in the cloud, indicating that some edge-originating requests have been shifted to the cloud due to a high processing time at local devices under high load. Finally, due to faster response, FaaSt can accept and process 15% more requests than Knative during the measurement period.

To this end, we can answer our third and last research question RQ3: *Improving the response time of requests in a serverless-enabled edge–cloud environment can be achieved by redesigning the platform's architecture to be latency-aware, specifically by optimizing the placement of networking components and implementing a strategic LB mechanism to ensure requests are routed to the function with the lowest latency.*

Limitations: Although the proposed FaaSt scheme shows significant improvement, several limitations can be identified. First, the LB approach may not be scalable for large scenarios, as $\binom{n}{2}$ network latency measurements need to be conducted for n nodes, increasing the complexity by $O(n^2)$. Second, the measurement results from geographically distributed nodes must be sent to a centralized metric server which leads to a single point of failure and introduces high latency. Third, the processing time matrix P currently assumes homogeneous application deployments. For heterogeneous deployments, more detailed information needs to be stored in multiple matrices. Despite these acknowledged limitations, the FaaSt load balancing mechanism serves as a proof of concept and can be potentially improved in future work.

V. Conclusion

Serverless computing enhances resource provisioning and automates operations, increasing resource utilization and reducing operational effort. While integrating serverless computing into the edge–cloud environment can provide mutual benefits, current serverless architectures are not optimized for in edge-related environments. Specifically, this paper identifies two major concerns: the positioning of networking components and the load balancing strategy. These issues increase application response times, which are crucial for modern service and IoT deployments.

To address these challenges, this paper proposes a novel scheme called FaaSt, designed to be latency-aware by targeting these identified problems. The results demonstrate significant improvements in the response times, with requests efficiently allocated to positions that yield the lowest latency, resulting in a significant reduction in the total latency. The scheme is implemented using open-source tools and standards, enabling researchers to further optimize it.

Nevertheless, several questions remain, most of which are addressed in our limitations section. Additionally, the impact of queuing in the buffer, the influence of the buffer's physical positioning within the cluster on response times, and the integration of a latency-aware scheduler and auto-scaler are aspects open for future research directions.

REFERENCES

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, 2019.

[2] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019.

[3] M. S. Aslanpour *et al.*, "Serverless edge computing: vision and challenges," in *Australasian computer science week multiconference*, 2021.

[4] FaaSt - Serverless Platform for Distributed System. [Online]. Available: https://github.com/bonavadeur/faast

[5] Amazon. Serverless Computing - AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/

[6] Knative. Knative Serving Autoscaling System. [Online]. Available: https://github.com/knative/serving/blob/main/docs/scaling/SYSTEM.md

[7] OpenFaaS. OpenFaaS - Serverless Functions Made Simple. [Online]. Available: https://docs.openfaas.com/

[8] Apache. Open Source Serverless Cloud Platform. [Online]. Available: https://openwhisk.apache.org/

[9] P. Kayal, "Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE, 2020.

[10] Redhat. System overview. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/about.mdplease-form-a-line-kafka

[11] Knative. Architecture - knative. [Online]. Available: https://knative.dev/docs/serving/architecture/

[12] OpenFaaS. Openfaas stack. [Online]. Available: https://docs.openfaas.com/architecture/stack/

[13] Knative. Serverless Containers in Kubernetes Environments. [Online]. Available: https://knative.dev/docs/

[14] K. Nguyen *et al.*, "Serverless computing lifecycle model for edge cloud deployments," in *2023 IEEE International Conference on Communications Workshops*. IEEE, 2023.

[15] Kubernetes. Topology Aware Routing. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/topology-aware-routing/

[16] Istio. Locality Load Balancing. [Online]. Available: https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/

[17] C. Cicconetti, M. Conti, and A. Passarella, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Transactions on Network and Service Management*, 2020.

[18] C. Chen, M. Herrera, G. Zheng, L. Xia, Z. Ling, and J. Wang, "Cross-edge orchestration of serverless functions with probabilistic caching," *IEEE Transactions on Services Computing*.

[19] A. Marchese and O. Tomarchio, "Orchestrating serverless applications in the cloud-to-edge continuum." Association for Computing Machinery, 2023.

[20] V. Mittal *et al.*, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *ACM Symposium on Cloud Computing*, 2021.

[21] G. Russo *et al.*, "Serverledge: Decentralized function-as-a-service for the edge-cloud continuum," in *International Conference on Pervasive Computing and Communications*. IEEE, 2023.

[22] S. Sekigawa, C. Sasaki, and A. Tagami, "Toward a cloud-native telecom infrastructure: Analysis and evaluations of kubernetes networking," in *2022 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2022.

[23] K. Nguyen *et al.*, "Investigation of serverless consumption and performance in multi-access edge computing," in *International Conference on Information Networking*. IEEE, 2024.

[24] Kubernetes. Production-Grade Container Orchestration. [Online]. Available: https://kubernetes.io

[25] Locust. A load testing framework. [Online]. Available: https://locust.io/

[26] Wireshark. Go deep.

[27] Prometheus. Monitoring System & Time Series Database. [Online]. Available: https://prometheus.io/

[28] Linux. TC(8) - Linux Manual Page. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[29] Tigera. About Calico. [Online]. Available: https://docs.tigera.io/calico/latest/about/

[30] Z. Cai *et al.*, "Jitter: random jittering loss function," in *Int. Joint Conference on Neural Networks*. IEEE, 2021.

[31] B. Charyyev *et al.*, "Latency comparison of cloud datacenters and edge servers," in *Global Communications Conference*. IEEE, 2020.