Scylla: Scheduling Multiple Latency-Sensitive Applications in the Edge-Cloud Continuum

Yinan Cao
ICTEAM
UCLouvain
Louvain-la-Neuve, Belgium
yinan.cao@uclouvain.be

Etienne Rivière
ICTEAM
UCLouvain
Louvain-la-Neuve, Belgium
etienne.riviere@uclouvain.be

Ramin Sadre
ICTEAM
UCLouvain
Louvain-la-Neuve, Belgium
ramin.sadre@uclouvain.be

***Abstract*—Cloud-hosted applications can experience significant latency, particularly for users located far from data centers. Edge computing mitigates these issues by distributing processing across geographically distributed locations. However, the limited resources at edge sites must be carefully allocated across applications to avoid contention that would outweigh the benefits of lower network latency. We present Scylla, a multi-application, latency-aware scheduler for edge/cloud environments. Scylla accurately predicts the relationship between resource allocation and tail latency using a queuing model faithful to the load distribution mechanisms used in Kubernetes. It then computes a global scheduling plan for multiple applications that ensures the respect of tail latency objectives, while minimizing the volume or cost of necessary edge resources. We evaluate Scylla on real-world infrastructure using Kubernetes and several latency-sensitive applications, including edge AI workloads. Our results show that Scylla effectively predicts tail latencies and makes scheduling decisions that utilize just the right amount of resources with low computational overhead.**

***Index Terms*—Edge-Cloud Continuum, QoS-aware Scheduling, Tail Latency, Kubernetes, Combinatorial Optimization**

I. Introduction

Edge computing extends cloud offerings based on large data centers with geographically distributed resources. The proximity of edge sites is a vital asset for enabling latency-sensitive applications, such as augmented and virtual reality, IoT measurement and actuation, and more generally, the emerging class of Edge AI applications that require fast and predictable inference times [1].

The deployment of applications across the continuum of cloud and edge sites requires careful planning. While it may seem natural to deploy application instances at edge sites close to their users, the limited resources at these sites may lead to increased response times due to contention, a phenomenon known as the *performance inversion* problem [2]. Furthermore, latency-sensitive applications generally define their Service-Level Objectives (SLO) as the worst-case service time they can bear, i.e., as their *tail latency*, expressed as a maximum for a high percentile of service-time distribution (e.g., the maximum acceptable latency for the 99^{th} percentile of this distribution, or P99 for short). During peak demand periods, resource contention between applications can lead to degraded performance, commonly referred to as the *noisy neighbor problem*, and the violation of these tail latency SLOs. While applications under contention may decide to spawn new instances at more edge sites, doing so individually can lead to global issues, such as cascading contention and SLO violation cycles, ultimately resulting in poor global resource utilization.

Existing solutions for scheduling latency-sensitive applications often fail to consider the predictability of tail latencies and the effects of contention, or do not account for the distributed nature of the edge-cloud continuum. Latency-aware schedulers were proposed for cloud-only applications [3]–[5], including for multi-component applications [6], [7], but without consideration of resource limitations at a *collection* of sites and different network latencies between these sites. Works targeting the edge-cloud continuum focus on a single application [8]–[11], and generally target *mean* service times rather than tail latency [9]–[11].

Contribution. We argue that a *global* scheduling approach is necessary to allow the deployment of multiple applications in the edge-cloud continuum. In contrast to approaches that schedule applications independently [8]–[11], a global approach enables achieving resource allocations that allow every application to meet its SLO, avoid cascading contention, and optimize resource utilization.

We demonstrate the interest of global edge-cloud scheduling with Scylla, a scheduler that holistically considers multiple applications and multiple edge and cloud sites. Scylla combines accurate models of applications' tail latencies, including queuing time under concurrent requests, with knowledge of network latencies. It generates global placement plans that ensure the tail latency SLOs are met for all applications. Scylla supports global optimization objectives that extend beyond the individual view of applications, allowing for system-wide optimization of resource utilization and costs.

Scylla builds upon the following contributions:

1) We propose a model that predicts both the mean and P99 latency of users' requests to applications. This model builds upon queuing theory, accounting for network latencies and request queuing times under contention.
2) We define the global scheduling problem of multiple

979-8-3195-1703-6/26/$31.00 ©2026 IEEE

applications in the edge-cloud continuum as an optimization problem. This enables the respect of applications' tail latencies while setting global optimization objectives, such as minimizing the number of sites used, overall resource consumption, or deployment costs.

3) We identify the NP-hardness of this global scheduling problem and propose efficient heuristics that provide near-optimal solutions in practical scenarios at a significantly reduced computational overhead (up to 500x).
4) We integrate our scheduler into a multi-site Kubernetes [12] installation, enabling the holistic scheduling of multiple applications.

We evaluate Scylla on a real testbed with emulated representative network latencies. We use four latency-sensitive applications and three different global optimization objectives. Our results demonstrate the ability of Scylla models to accurately predict application tail service latency when deployed on Kubernetes and subjected to requests from multiple independent clients. The placement plans computed by Scylla ensure compliance with P99 latency SLOs for all applications, even during high concurrency, while using only the necessary resources. In contrast, the state-of-the-art research scheduler INVAR [9] and the Kubernetes HPA built-in auto-scaler fail to meet these objectives, resulting in tail latency SLOs violations of up to 171% under the same conditions.

Outline. Section II overviews related work. Section III details our problem statement. Section IV describes our scheduling algorithm. Section V extensively evaluates the performance of Scylla in a Kubernetes testbed, compares it to INVAR [9], Kubernetes' HPA, and local-first heuristic, and discusses our approach's advantages and limitations. Section VI concludes the paper and discusses future work.

II. Related Work

Several approaches have been explored for scheduling applications across the edge-cloud continuum. Simple approaches rely on *reactive* auto-scaling techniques, e.g., based on thresholds on average CPU utilization [13], [14]; Kubernetes' HPA mechanism is based on this principle and commonly used in production [15]. More advanced approaches employ a *predictive* approach based on analytical models that forecast the performance of applications and their response to resource allocation [2], [5], [10], [11], [16]–[18]. This forecasting may use machine learning, requiring sufficient historical telemetry data [3], [4], [7], or simpler analytical models. In this latter class, INVAR [9] explicitly targets edge-cloud settings and aims to avoid the performance inversion problem [2]. Its model uses queuing theory to predict the mean end-to-end latency for an application and determine the number of required instances at edge sites, along with mapping users to these sites, using the deployment budget as the optimization objective. LASSY [8] extends over INVAR by considering tail latency (P99), but remains limited to scheduling a single application.

Cloud-focused schedulers [3]–[7] employ complex but often *horizontal* models, i.e., without awareness of the distributed nature of the infrastructure and of users' locations; the closest among them to our setting, FIRM [3] and Sora [5], target SLO-aware resource management but assume a single datacenter and do not model heterogeneous network latencies across distributed sites. Schedulers targeting an edge-cloud setting focus on a single application at a time, forfeiting the ability to globally optimize SLO-compliance and resource utilization [2], [8]–[11], [16], [17]. Moreover, most of these approaches are only validated through simulations: for instance, the INVAR [9] queuing model has been shown to be ill-matched to the load balancing and request dispatching mechanisms used in modern cloud stacks such as Kubernetes [8]. To the best of our knowledge, no existing system simultaneously accounts for an edge-cloud distributed setting, realistic latency models validated against production deployment systems such as Kubernetes, and a global optimization approach across multiple applications.

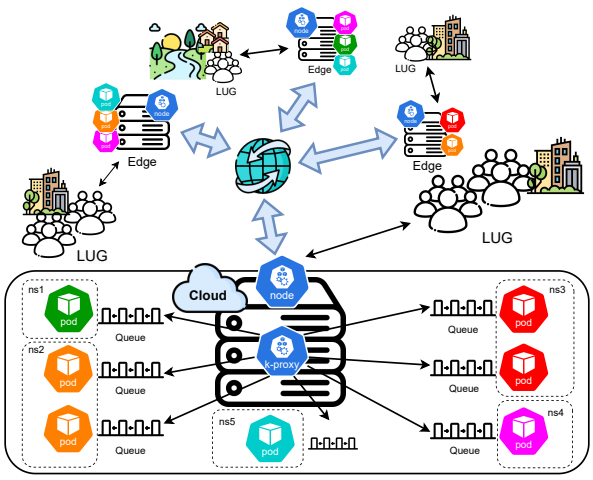


Fig. 1: A typical edge-cloud continuum infrastructure.

III. Problem Statement and Scope

We now describe the targeted edge-cloud continuum architecture and detail the challenge of globally scheduling multiple applications that we address in this paper.

A. The Edge-cloud Continuum

The edge-cloud continuum model that we consider in this work, illustrated by Figure 1, aligns with the model used by related work [2], [8], [9], [13], [19]. Cloud sites are large data centers equipped with massive computing resources, whereas edge sites are small data centers or server stacks with more limited resources. Geographically, cloud sites are located far from densely populated areas where most users reside, whereas edge sites are positioned closer to these users. As a result, cloud sites offer abundant computing resources but incur significantly higher network latency for most users than edge sites [20].

Most users are concentrated in densely populated regions, such as towns or cities. Users who share similar network latencies to various cloud and edge sites can, therefore, be identified as a group [21]. We refer to such a group of users

as a *Local User Group*, or LUG for short, and treat it as a single entity, following the approach of prior studies [8], [9], [22], [23] and industrial practice [24], [25].

We focus on services that involve a synchronous request-response loop between the user and the back-end, involving computations of varying complexity, from lightweight real-time services to more complex edge AI applications. Such services can be effectively deployed as stateless, monolithic applications across different sites, e.g., as containerized instances. Statelessness here means that any instance can handle any request, a design widely regarded as efficient and modern [2], [3], [5]. Each instance of a given application consumes a fixed, equal amount of computing resources, whereas instances of different applications have different resource requirements. To accommodate varying workloads, the application can scale out or in by adding or removing instances on a site or across sites. However, the resources allocated to a specific instance remain fixed; we do not vertically scale individual instances up or down by modifying their assigned resources.

We consider that each cloud and edge site uses Kubernetes [12], a widely used open-source orchestration platform for containerized applications. In Kubernetes, each physical or virtual machine is referred to as a *node*, and the application instances running on them are called *pods*. To expose a group of pods as a single network endpoint, Kubernetes uses a *service*, which abstracts and load-balances traffic across the same pods. A service proxy, *kube-proxy* [26], routes incoming user requests to individual pods using a random dispatch method. If the selected pod is busy, the request is queued at that instance, as illustrated in Figure 1.

B. Challenge

We consider a multi-application setting. Each application may receive requests from different LUGs and aims to serve these requests with a target Service Level Objective (SLO). This SLO relates to the end-to-end *tail latency*, i.e., from the moment the client in a given LUG sends the request until the response is received. It is defined as a maximal acceptable value for a high percentile of the latency distribution. We use the 99$^{\text{th}}$ percentile, or P99, in the remainder of this paper.

The problem at hand is to decide and instruct Kubernetes instances at all cloud and edge sites to bootstrap new application instances, and to decide on the mapping between LUGs and sites (i.e., all requests from a LUG for a given application are directed to a specific site, as commonly used by hyperscalers to enable local caching [24], [25]). The primary goal is to determine a placement and mapping that enables meeting SLOs for all applications. A secondary goal is that this placement aims to be minimal with respect to a global optimization objective. Examples of such global objectives include minimizing the number of instances deployed overall (and thus reducing fragmentation), minimizing the number of sites supporting instances (and thus allowing unused sites to be shut down), or minimizing overall deployment costs when each site is associated with specific billing criteria.

Name	Description
$st \in ST$	Cloud and edge sites
$app \in APP$	Applications
$nslots_{st}$	Slots of computing resource available at site st
$lug \in LUG$	Local user group(s)
$l_{lug,st}$	Round trip netw. latency between user group lug and site st
λ_{lug}	Request rate of user group lug
$lugapp_{lug}$	Application used by LUG lug
σ_{app}	Slots required by each instance of application app
μ_{app}	Service rate of an application instance
c_{st}	Cost per slot at site st
eel_{app}^{max}	SLO: Maximum P99 latency for application app
$ninsts_{st,app}$	Instances of app deployed on site st
$lugsite_{lug}$	Site that user group lug is directed to
$L_{st,app}$	Set of user groups sending requests for app to site st
$w_{st,app}$	Queueing delay at site st for app

TABLE I: Model parameters (top) and variables (bottom).

Deploying an application instance on the "*closest*" edge site to a given LUG yields the lowest networking latency, but this is not always feasible or desirable. As multiple applications compete for edge resources, a single site may not suffice to serve all the closest LUGs. In the worst case, contention can result in end-to-end latency worse than that of a cloud deployment [2]. Furthermore, this may require deploying instances across multiple sites for the same application, resulting in increased costs and resource fragmentation (i.e., allocated resources that are not fully utilized). The choice of a placement and mapping to a LUG is the result of a tradeoff between networking and queuing latencies on the one hand, and the volume and cost of allocated resources on the other hand. Intuitively, deploying an application instance close to a LUG results in lower network latency and allows for more *slack* in queuing requests while still respecting the SLO. In contrast, deploying such an instance at a more distant site may require more resources to achieve lower maximum queuing delays and guarantee the same SLO. Global scheduling enables determining trade-offs between latency and resource usage across all applications while meeting their SLOs and minimizing a global optimization objective.

IV. Scheduling with Scylla

We now formalize the scheduling problem introduced in Section III and show how suitable scheduling fulfilling the SLOs of all applications and minimizing operational costs can be obtained. The symbols used in the following definitions are summarized in Table I.

A. System and scheduling model

In our model, an edge-cloud infrastructure consists of multiple cloud and edge sites $ST = \{st_1, \ldots, st_{|ST|}\}$. Each site hosts servers that provide a certain amount of computation resources. We abstract from the individual machines and express the available resources at site $st \in ST$ as a number $nslots_{st} \in \mathbb{N}$ of uniform computation "slots".

We extend our previous work [8] and consider the set of monolithic applications $APP = \{app_1, \ldots, app_{|APP|}\}$,

for which instances can be deployed at the different sites. Each instance of an application app requires an application-specific number σ_{app} of computation slots. One of the tasks of the scheduler will be to determine the number of instances $ninsts_{st,app} \in \mathbb{N} \cup 0$ of app that should be deployed on site st, subject to the resource constraints

$$\sum_{app \in APP} ninsts_{st,app} \cdot \sigma_{app} \leq nslots_{st}, \text{ for all } st \in ST. \quad (1)$$

The instances of an application app serve user-issued requests. We assume that each request requires a constant service time $\frac{1}{\mu_{app}}$ and, therefore, an instance can serve requests at a rate of μ_{app}. As we will see in Section V, this model provides excellent results for a wide range of applications, including those employing machine learning models, where the service time is primarily determined by a constant inference time.

As motivated in Section III-A, a local user group $lug \in LUG = \{lug_1, \ldots, lug_{|LUG|}\}$ stands for a group of clients in the real world who are geographically close to each other and have similar network latencies to the different sites. The round trip network latency between a group lug and site st is given by $l_{lug,st}$. We assume that a group lug sends requests to exactly one application $lugapp_{lug} \in APP$ and one site $lugsite_{lug} \in ST$ (without loss of generality, as the same group of physical users can be represented by multiple LUGs sending requests to different sites and applications). We assume that the number of requests sent by group lug per time unit follows a Poisson distribution with rate $\lambda_{lug} \in \mathbb{R}_{>0}$. This follows from the idea that LUGs represent large numbers of independently acting real-world clients.

Scheduling the set of applications APP means determining (a) how many resource slots $ninsts_{st,app} \cdot \sigma_{app}$ to allocate for each application on each site, and (b) which LUG should be served by which site, i.e., $lugsite_{lug}$, such that the application SLOs are fulfilled. As explained in Section III-B, the SLO of application app specifies that the probability that a request experiences an end-to-end-latency of less than eel_{app}^{max} must be greater than or equal to 0.99.

Following the request dispatching mechanism of Kubernetes (see Section III-A), requests for an application app from users to a site st are randomly and evenly dispatched to the $ninsts_{st,app}$ instances running at that site. Under the assumptions stated above, each of the instances can be treated as an independent $M|D|1$ queue with service rate μ_{app} and arrival rate $\lambda_{st,app}^{inst}$ given by

$$\lambda_{st,app}^{inst} = \frac{\sum_{lug \in L_{st,app}} \lambda_{lug}}{ninsts_{st,app}}, \quad (2)$$

where $L_{st,app}$ is the set of LUGs sending requests to application app on site st:

$$L_{st,app} = \{lug \in LUG \mid lugsite_{lug} = st \wedge lugapp_{lug} = app\}. \quad (3)$$

Since we assume that all instances of an application receive the same number of computation resources, all requests processed by a site for an application experience the same queueing delay distribution. The end-to-end latency of a request sent by lug for application $a := lugapp_{lug}$ to site $s := lugsite_{lug}$ is composed of the network latency $l_{lug,s}$, the queueing delay $w_{s,a}$ at the instance serving the request, and the constant service time $\frac{1}{\mu_a}$. Accounting for application's SLO yields the following set of constraints:

$$P(l_{lug,s} + w_{s,a} + \frac{1}{\mu_a} \leq eel_a^{max}) \geq 0.99$$
$$\text{with } a := lugapp_{lug}, s := lugsite_{lug}, \text{ for all } lug \in LUG. \quad (4)$$

We assume that the size of the requests and responses is small compared to the network bandwidth, i.e., the network latency does not depend on the individual requests and responses. Furthermore, similarly to previous work [9], we assume that network latencies between users and sites in the continuum are relatively stable. This allows us to approximate $l_{lug,s}$ by the average network latency $\bar{l}_{lug,s}$ and to write the constraints (4) as

$$P(w_{s,a} \leq eel_a^{max} - \bar{l}_{lug,s} - \frac{1}{\mu_a}) \geq 0.99$$
$$\text{for all } lug \in LUG. \quad (5)$$

The probability $P(w_{s,a} \leq \cdot)$ in constraints (5) is the queueing delay distribution of the $M|D|1$ queue. It is given by [27]

$$P(w_{s,a} \leq t) = (1 - \frac{\lambda_{s,a}^{inst}}{\mu_a}) \cdot \sum_{i=0}^{\lfloor t \cdot \mu_a \rfloor} e^{-\lambda_{s,a}^{inst}(i/\mu_a - t)} \frac{(i/\mu_a - t)^i}{i!} (\lambda_{s,a}^{inst})^i. \quad (6)$$

B. Scheduling as optimization problem

In general, the scheduling problem presented above will have multiple solutions. An optimization objective is therefore necessary to select the best solution. Since edge-cloud providers often charge fees based on allocated resources, one such objective could be to minimize these fees, i.e.,

$$\min \sum_{st \in ST, app \in APP} c_{st} \cdot ninsts_{st,app} \cdot \sigma_{app}, \quad (7)$$

subject to the constraints (1) and (5), where c_{st} is the cost of a slot at st. The solution of this mixed-integer programming problem is the optimal instance allocation $ninsts_{st,app}$ for all sites st and applications app, as well as the user mapping $lugsite_{lug}$ for all user groups lug.

Alternative objectives can be easily formulated. For example, the infrastructure operator could want to minimize the number of used sites, i.e., $\min |\{st \in ST \mid \sum_{app \in APP} ninsts_{st,app} > 0\}|$, as the unused sites can then be shut down to save energy.

Some of the model parameters, namely the request rates λ_{lug} of the user groups, the average network latencies $\bar{l}_{lug,st}$, and the service rates μ_{app}, have to be determined before scheduling can take place. We consider the techniques used

to establish these parameters to be outside the scope of this paper, given that they have been discussed at length in existing literature. These include direct and indirect techniques for estimating network latencies [28] and methods for determining arrival and service rates based on previous observations [13].

C. Layered Permutation Search heuristic

The above optimization problem, as a generalized variant of the *multiple knapsack problem* extended with latency requirements, is commonly known as NP-hard.

Global optimization. Modern solvers employ advanced techniques, such as branch-and-bound and cutting planes, to solve such problems efficiently for small-scale scenarios. Scylla specifically uses the Gurobi Optimizer [29]. We denote the solver algorithm that computes the optimal solution for our scheduling problem as G_{opt}. Unfortunately, for larger-scale scenarios consisting of numerous sites, applications, and local user groups, the exponential growth of computation time for G_{opt} is prohibitive.

Full permutation search. One way to reduce the computation time is to use a greedy algorithm H_{Full} with full permutation search: We could first schedule application app_1 by running G_{opt} with just app_1 and its users, then schedule app_2 on the remaining resources, and so on until all applications are placed. However, such an approach is sensitive to the order in which applications are considered. Such full permutation search of all $|APP|!$ application orderings would, thus, be necessary to avoid suboptimal solutions. While G_{opt} with a single application in each permutation can significantly reduce computation time, a full enumeration of all $|APP|!$ permutations exhaustively explores the sequential decision space to approach the global optimum, it is computationally intractable for large $|APP|$.

Layered Permutation Search. To address computational complexity, we propose the *Layered Permutation Search (LPS)* heuristic H_{LPS} (Algorithm 1). Instead of enumerating all $|APP|!$ permutations, H_{LPS} takes a depth parameter d and evaluates all ordered prefixes of d applications, while appending the remaining $|APP| - d$ applications in a random order. For each permutation, applications are scheduled sequentially via G_{opt} (lines 6–13): for each application in turn, G_{opt} determines the optimal instance count and LUG to ST assignment subject to the tail latency SLO constraint and the remaining available resource budget; if no feasible placement exists, the permutation is discarded and the next one is tried. If the resulting solution $\mathcal{S}'$ yields a lower objective value $f(\mathcal{S}')$ than the best found so far, the global solution is updated (line 15). By increasing d from 1 to $|APP|$, the search space is explored progressively: at $d = 1$, only $|APP|$ permutations are evaluated, while at $d = |APP|$, all $|APP|!$ permutations are covered. The depth d thus acts as a tunable trade-off between solution quality and computational cost.

The LPS strategy can be viewed as a generalized *Beam Search* [30] where the beam width is determined by the prefix depth d. By exhaustively exploring all permutations at a fixed depth, LPS transforms from a high-breadth heuristic into a progressive full-space search as d increases.

Algorithm 1 H_{LPS}: Layered Permutation Search Heuristic

```
1: Input: d                                          ▷ Desired depth parameter
2: S ← ∅, V ← ∞                                      ▷ Best solution and objective value
3: P ← GeneratePrefixPermutations(APP, d)
4: for π ∈ P do
5:     avail_resources ← nslot_S, S' ← ∅
6:     for app in π do
7:         G_opt(app, avail_resources)
8:         if scheduling feasible then
9:             Update S' and avail_resources
10:        else
11:            S' ← ∅; break
12:        end if
13:    end for
14:    if S' ≠ ∅ and f(S') < V then
15:        S ← S', V ← f(S')
16:    end if
17: end for
18: return S
```

V. Experimental Evaluation

We provide a detailed experimental evaluation of Scylla and answer the following research questions (RQ):

- **RQ1:** How accurate are Scylla's latency predictions and queueing model when compared to applications running over Kubernetes in a realistic deployment setting?
- **RQ2:** How does Scylla perform in complex scenarios consisting of multiple sites, applications, and LUGs?
- **RQ3:** What is the impact of different global optimization objectives, such as the minimization of used slots, open sites, or global costs?
- **RQ4:** Does H_{LPS} provide a scalable alternative to G_{opt} and H_{Full} while preserving solution optimality in practical multi-application scenarios?

We support reproducibility and provide our code, the experimental setup, and instructions to reproduce our work [31].

A. Experimental setup

We perform all our experiments on a testbed cluster and do not use simulations. We used 22 cluster nodes, each with an 18-core CPU (Intel Xeon Gold 5220) and 96 GB of RAM. We deploy Kubernetes with a dedicated node hosts the control plane, and logically isolate groups of nodes using *labels* to form cloud and edge sites as well as LUGs.

For reproducibility and flexibility, we utilize network emulation to replicate WAN latencies between LUGs and the different sites. We implement a network topology generator that utilizes network coordinates in an Euclidean plane and decides on latencies between all system actors as the distance in that plane [28]. We emulate the resulting latencies with *tc*.

LUGs and load injection. We emulate LUGs workloads using *httperf* [32], a load generator able to send requests to a remote service at a high rate while accurately following a Poisson distribution of inter-request times. This distribution is key to emulating the behavior of a group of independently

acting users. The *httperf* processes corresponding to LUGs are pinned on dedicated cores to avoid scheduling hazards.

Resource allocation. We allocate resources at cloud and edge sites at the granularity of a *slot* with 1 CPU core and 1 GB of RAM. Each application instance is deployed as a Kubernetes pod by the control plane and uses a fixed, integer number of slots. Slot capacity is homogeneous across sites.

Applications. We select four representative applications with diverse resource needs. (**a**) *Thumbnailing* accepts a `base64` string representation of an image and returns a resized image in the same format. Each instance of *Thumbnailing* is assigned one slot. (**b**) *OCR* implements text-recognition from an image provided as input. We implement it using Tesseract [33], a reference OCR engine based on an LSTM neural network. Each instance of *OCR* requires two slots. (**c**) *MatrixProduct* multiplies two fixed-size matrices and returns the result. Each instance of *MatrixProduct* is assigned two slots. (**d**) *Translator* implements a translation service. We implement it with EasyNMT [34] and the neural machine translation model Opus-MT [35], [36]. *Translator* is the most demanding application, each of its instances requires eight slots.

We choose these applications for their diversity and representativeness of latency-sensitive edge-cloud workloads. *Thumbnailing* is network-latency-dominated with very short service times, while *MatrixProduct* is compute-intensive. *OCR* and *Translator* represent resource-heavy edge AI services with relatively long response times, powering scenarios such as augmented reality and mobile applications.

We use three runs of five minutes each for all experiments. We discard the first 10% of measurements to eliminate warm-up effects. We report averages over these three runs. Confidence intervals were always negligible, hence we do not report them in the figures.

Baselines. We compare Scylla to two baselines. The first one is INVAR [9]. This scheduler generates placement plans for edge-cloud scenarios, focusing on a single application and SLO objectives based on mean end-to-end latency. The second is Kubernetes' built-in Horizontal Pod Autoscaling (HPA) [15], one of the most commonly deployed resource allocation mechanisms in industry. HPA triggers the scale-out (i.e., adds instances) of applications reactively when their average CPU utilization over a specified observation period exceeds a threshold. To account for our focus on latency-sensitive applications, we set this threshold to 60% average CPU rather than the default 80% [5]. In contrast with Scylla and INVAR, HPA does not use a model of application performance but only reacts to system-level metrics. For multi-site multi-application scenarios in Section V-D, we also compare the placement plans computed by Scylla to a *local-first heuristic* which abides by the SLO requirements of each application, but uses only resources on the closest edge site to each LUG and only uses the cloud resource if resources at that edge site are insufficient, mimicking a situation where application developers would be in charge of deciding where to place their own application, without global decision making.

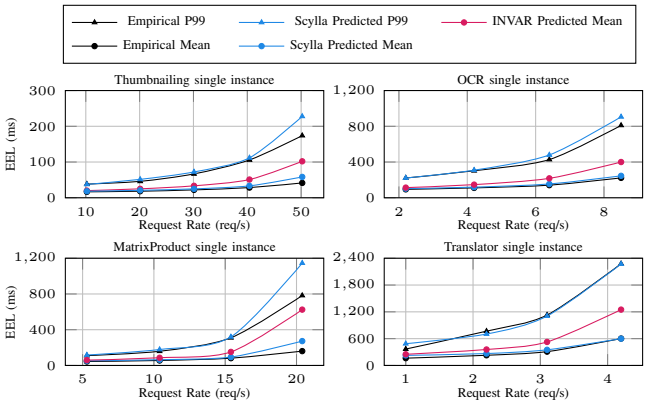


Fig. 2: Evaluation of performance and prediction accuracy for applications' single instances used in isolation and local setting.

B. Baseline applications performance prediction

Our first experiment targets RQ1. It evaluates the quality of Scylla latency predictions and compares them to those of INVAR. More precisely, we evaluate whether the queueing model used by Scylla is suitable to predict the tail latency of requests sent to an application instance. To this end, we deploy a single instance of each target application and send requests at a controlled mean rate with a Poisson-distributed inter-arrival time. We consider a local setting with no network latency to focus on the estimation of queuing delays.

Figure 2 shows the measured mean and P99 end-to-end latencies (EEL) for the four applications with increasing request rates. The blue and red curves are the predictions reported by Scylla's $M|D|1$ model and INVAR's $M|M|c$, respectively. Note that INVAR does not provide a prediction of tail latency but only of its mean.

Scylla accurately predicts the tail latency for lower to medium request rates, with an average relative prediction error of 11% across all four applications, only overestimating it slightly for high request rates close to the saturation point. In contrast, INVAR's prediction of the average latency is relatively inaccurate, with an average relative error of 49%. This is due to the use of exponentially distributed service times in INVAR's model and to the mismatch between the single-queue-multiple-server model used and how a Kubernetes installation dispatches requests to pods, an element that INVAR's simulation-only evaluation does not consider.

C. Model evaluation: Single-site scenario

Our second experiment continues the exploration of RQ1 and evaluates whether the latency predictions made by Scylla allow determining the necessary scale-out to meet a target P99 latency SLO. We focus on the *Thumbnailing* application. Results with other applications, not shown due to space constraints, are similar. We set an SLO of $eel^{max} = 100$ ms and let Scylla decide on the number of instances for increasing input request rates. We compare our results to those of INVAR and HPA. For INVAR, which only supports mean latency targets, we set its objective to 50 ms (half the P99 latency

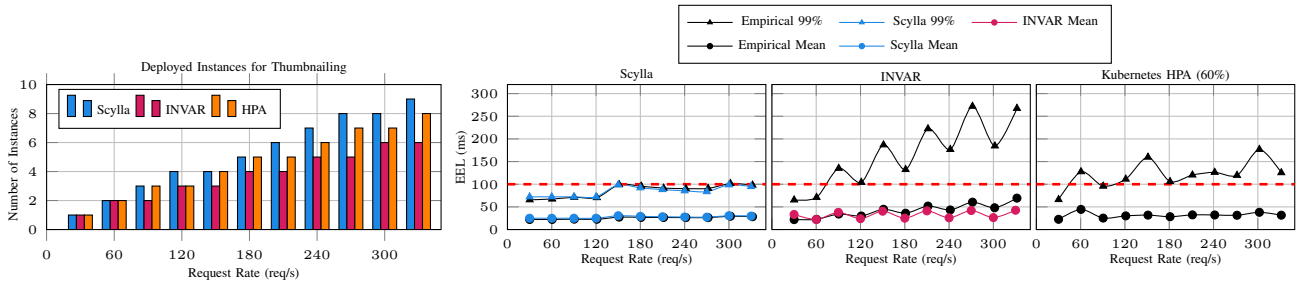


Fig. 3: Empirical and predicted EEL for the single-site scenario using Scylla, INVAR, and Kubernetes HPA to control the number of instances deployed for the *Thumbnailing* application. The dashed red line indicates the P99 SLO for *Thumbnailing*.

			Network latency from LUG to site (ms)									
	Slots	Price	LUGs 1-4	LUGs 5-8	LUGs 9-12	LUGs 13-16	LUGs 17-20	LUGs 21-24	LUGs 25-28	LUGs 29-32	LUGs 33-36	LUGs 37-40
C	72	$1	38	58	34	31	34	60	57	42	60	60
E1	36	$3	56	108	16	49	84	10	61	42	60	110
E2	36	$4	12	70	52	81	40	60	107	20	110	60
E3	36	$2	88	58	48	19	60	60	11	92	16	60
E4	36	$2	44	20	84	51	16	110	57	70	66	10

TABLE II: Resource and pricing for the cloud (C) and edge (E) sites, and relative latencies.

target). As for the previous experiment, network latency is zero between the single LUG and the single site.

Figure 3's left-most plot shows the number of instances deployed by Scylla, INVAR, and HPA. The three other plots show the measured mean and P99 latencies for requests for the three solutions, as well as predictions for Scylla and INVAR. Scylla systematically determines the sufficient number of instances for the *Thumbnailing*, with predictions that closely match the performance of the application and of the Kubernetes load balancing policies. In contrast, the number of instances decided by INVAR is insufficient even if the mean EEL latency stays well below the target 100 ms, leading to P99 SLO violations of up to 171%. Similarly, the number of instances decided by HPA (after convergence of the elastic scaling process) is also insufficient, with violations up to 76%.

We answer positively to RQ1: Scylla latency predictions are accurate compared to Kubernetes and allow effective scale-out decisions for meeting a tail latency SLO for a given request rate to a specific site. INVAR and HPA are unable to provide this guarantee. As these baselines also do not consider the scheduling of multiple applications, as we discuss next, we do not consider them further in our evaluation.

D. Multi-site, multi-application scenario

We now proceed to answering RQ2 with a complex, multi-site, and multi-application scenario.

Infrastructure. We consider one cloud site (C1) and four edge sites (E1-E4). The capacity is set as 36 slots for edge sites and 72 for the cloud site. The "Price" column in Table II indicates the price per slot and per unit of time for the different sites, with the cloud being the cheapest and edge sites being 2 to 4 times more expensive. We select ten locations uniformly distributed in the network coordinate Euclidean space and having varying latencies to the sites, with one close-by edge site and varying latencies to the cloud and other edge sites (the cloud is not necessarily the farthest away site for a location). The network latencies between the locations and the sites are based on the measurements of Ali *et al.* [2] and presented in Table II. At each location, we place four LUGs (one LUG for each of the four applications), thus 40 LUGs in total.

	#1	#2	#3	#4	#5	#6	#7	#Imp
Thumbnailing	5	10	107	205	304	402	500	**290**
OCR	5	10	19	24	33	41	50	24
MatrixProduct	5	10	29	46	66	82	100	46
Translator	3	5	6	7	8	9	10	7

TABLE III: Volume of requests for each of the four applications in the seven different configurations (in requests/second).

Applications SLOs. We use the four applications and set P99 latencies as follows: $eel^{max} = 100$ ms for *Thumbnailing*, $eel^{max} = 400$ ms for *OCR*, $eel^{max} = 200$ ms for *MatrixProduct*, and $eel^{max} = 800$ ms for *Translator*. Each instance of an application deployed by Scylla uses the number of slots detailed in Section V-A.

Load configurations. We consider a gradual load increase scenario through seven *configurations*, i.e., distribution of request rates from the LUGs to the four applications. Table III provides the *total* request rate for each application in these configurations labelled #1 to #7. This total rate is distributed among LUGs using a normal distribution. Each configuration gradually increases the volume of requests, and thus the necessary resources to serve them.

Impulsive load configuration. In addition, a configuration named #Imp represents an "impulse" scenario based on configuration #4, where the request rate for the *Thumbnailing* application increases by an additional 85 req/s coming from LUG1. This represents a sudden surge in user activity, such as a local event. This selective amplification of load enables the evaluation of Scylla performance under highly skewed demand conditions, where a single user group generates a disproportionate volume of requests. The rightmost column in Table III shows the total request rates from all 10 LUGs for each application under configuration #Imp.

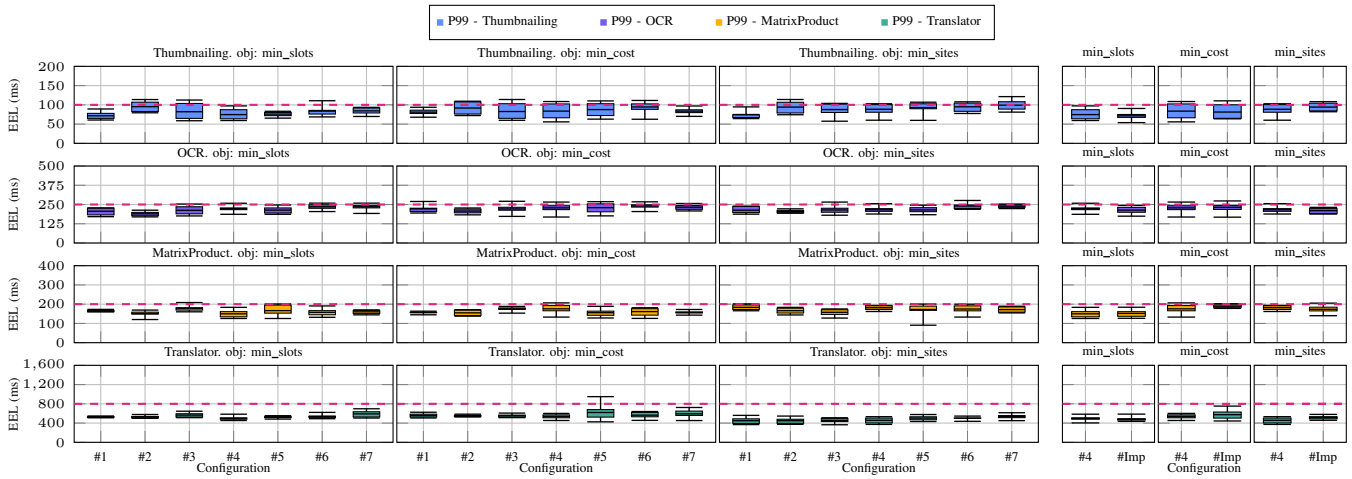
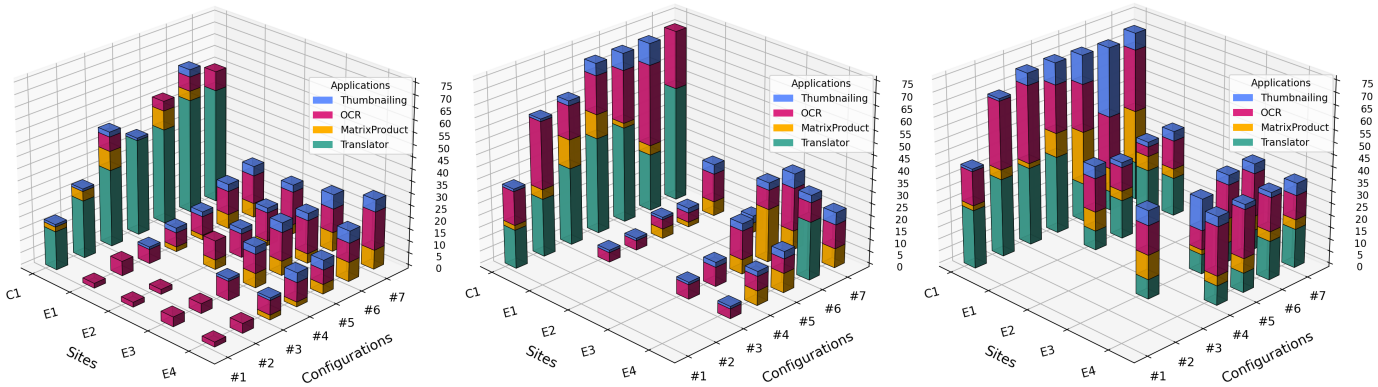


Fig. 4: P99 latency distributions for the LUGs across the seen configurations and the impulsive configuration, for the four target applications and three global optimization objectives. The red dashed line in each plot indicates the application SLO set for its P99 latency.



(a) Objective: min_slots (b) Objective: min_cost (c) Objective: min_sites

Fig. 5: The placement plan generated by Scylla for all configurations in multi-site multi-app scenario under three optimization objectives. Each bar with a color represents the number of resource slots consumed by an application on each site.

Optimization objectives. We consider three different global optimization objectives. The first one, named "min_slots", aims to reduce the number of used slots overall without consideration of price or the number of activated sites. The objective of min_slots is $\min \sum_{st \in ST} ninsts_{st} \cdot \sigma_{app}$. The second one, named "min_cost", minimizes the total cost of the deployment based on the slot unit price defined in Table II. The objective of min_cost is $\min \sum_{st \in ST} c_{st} \cdot ninsts_{st} \cdot \sigma_{app}$. The third objective, "min_sites", aims to limit the number of active sites, i.e., hosting at least one active slot. This objective allows shutting down unused sites and saving energy. The objective of min_sites is $\min |\{st \in ST \mid \sum_{app \in APP} ninsts_{st,app} > 0\}|$. We emphasize that these three global objectives are intended to be illustrative. Scylla would support the easy integration of additional policies, e.g., based on carbon footprint [37] or the nature of electric sources [38].

Results. Figure 4 shows the distribution of observed P99 latencies for all configurations, while Figure 5 graphically represents the number of slots used by each application on each site for configurations #1 to #7.

Config	Objective	Used slots	Total cost	Used sites
#3	min_slots	**73**	105	4
	min_cost	77	**99**	4
	min_sites	107	143	**2**
	local-first	92	243	4
#7	min_slots	**152**	312	5
	min_cost	156	**276**	5
	min_sites	178	321	**4**
	local-first	160	368	5

TABLE IV: Global results obtained with the three optimization objectives and local-first heuristic as baseline in configuration #3 and #7 as examples.

The placement plans (number of instances and assignment of LUGs to sites) decided by Scylla successfully keep the mean of the observed P99 latency below the SLO for all applications, thereby positively answering RQ2. We observe minor violations for the *Thumbnailing* application for the maximum of the tail latency distribution, but not for the other applications, with a single exception for *Translator* under configuration #5 for the min_cost objective. Overall, the tail

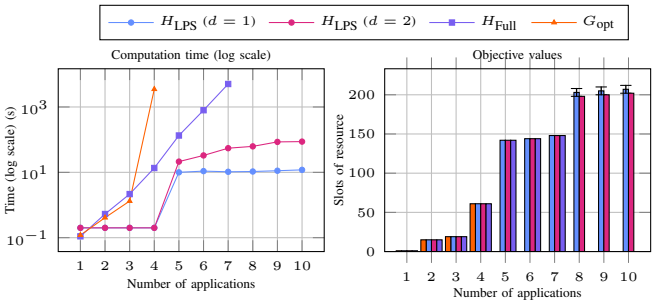


Fig. 6: Computing time (log scale) for H_{LPS} with $d = 1$, $d = 2$, H_{Full} and G_{opt} under the objective min_slot (left), along with the objective values (right). Error bars in the right plot indicate the variation in objective values caused by the random ordering of the remaining applications in LPS at different depths.

latencies observed in configurations decided by Scylla closely match the requirements. This includes the #Imp scenario, represented to the right of Figure 4 and compared to configuration #4, showing that Scylla can effectively generate suitable placement plans for a wide range of scenarios, including demand bursts from specific LUGs.

The results of the three optimization objectives are visible on the distribution of slots per site given by Figure 5. For the min_slots objective, we observe that edge sites are used starting from the first configuration. Scylla places *Translator*, the heaviest application, on the cloud site (but with sufficient resources to compensate for network latency by reducing queuing). Other applications are deployed mostly at the edge, as they require fewer slots than in the cloud for a given workload, thanks to lower networking latencies. The min_cost deploys instances preferentially in the cloud site due to its lower cost, but starts using more expensive edge sites starting from configuration #3, as the closer edge sites E1, E3, and E4 become competitive due to their better resource effectiveness. When cloud resources are insufficient, starting from configuration #4, minimizing this objective results in the balancing of the instances over all sites, using the most expensive E2 only in configuration #7. Finally, the min_site objective result in one site (E2) never being used, but also in more slots and higher costs than the two other policies.

Table IV reports the numerical results under the three optimization objectives together with the local-first heuristic as baseline for configurations #3 and #7. Each objective from Scylla achieves the best performance on its targeted metric. Compared to the optimization-based solutions, the local-first heuristic consistently incurs higher resource consumption and cost. These results demonstrate that Scylla can be guided to select SLO-compliant solutions for all applications while optimizing different global objectives, thereby answering RQ3.

E. Heuristic algorithm evaluation

We finally evaluate the impact of the heuristic algorithm H_{LPS} compared to G_{opt} and H_{Full} to answer RQ4. For all experiments presented above, H_{LPS} and G_{opt} return the same scheduling solution.

To explore this further, we use the same parameters (5 sites and 10 LUGs each application) as with the experiment in the previous subsection except for the number of applications that we vary from 1 to 10 under the min_slots objective. Figure 6 shows the computation times measured on an Intel CPU with 32 cores. We make use of Gurobi's multi-threading capability and let it use all cores when running G_{opt}. For H_{Full} and H_{LPS}, we explore the set of permutations in 8 parallel processes and give each Gurobi instance 4 cores.

We observe that computation times become prohibitive with more than 4 applications for G_{opt} due to the explosion of the search space, and more than 6 applications for H_{Full} due to the factorial growth of the number of permutations, we therefore omit these results for higher numbers of applications. In contrast, the running time for H_{LPS} is significantly reduced (up to 500x), while yielding indistinguishable results when the resources on each sites are relatively sufficient (less than 7 applications). When the resources at edge sites become scarce (beyond 7 applications), placement plans computed by H_{LPS} with $d = 1$ exhibit a larger optimality gap (around 5%) compared to those with $d = 2$, highlighting the inherent trade-off between runtime and solution quality in the heuristic.

Using H_{LPS} is practical for periodic re-assessment of resource and LUG-to-site assignments even for complex, multi-application scenarios, thus answering RQ4 positively.

F. Discussion

Our model of multiple independent $M|D|1$ queueing stations is well representative of Kubernetes dispatching mechanism and allows modeling the tradeoff between queuing time and network latency that matters greatly when scheduling for tail latency. In some scenarios, Scylla slightly underestimates the number of instances due to jitter in service time. One possible approach is to estimate an upper bound on the service time based on historical data and let Scylla generate a placement plan that provisions sufficient instances to cover the worst-case scenarios.

When resources are scarce, scheduling decisions become tightly coupled and the feasible space shrinks, making application ordering more critical. Our proposed H_{LPS} therefore exposes a controllable depth-runtime trade-off: exploring more prefix permutations improves solution quality and moves closer to the global optimum, while limiting the depth reduces computation time at the cost of potential optimality loss.

Regarding scalability with the number of sites, the combinatorial complexity of H_{LPS} stems from the factorial growth of application permutations, not from the number of sites: each G_{opt} call solves a single-application subproblem whose size grows polynomially with the number of sites, and remains tractable for the site counts typical of edge-cloud deployments.

VI. Conclusion

We presented Scylla, a scheduler for multiple applications in the edge-cloud continuum. Scylla uses queuing models to predict the tail latency of the end-to-end query-response cycle of applications with distributed users. Scylla employs a global

scheduling approach that enables meeting each application's tail latency objective while optimizing overall objectives. The evaluation of Scylla shows that it accurately predicts tail latency and allocates resources in scenarios of increasing complexity, unlike the closest state-of-the-art and standard elastic scaling solutions.

Acknowledgments. This work has been funded by the Wallonia-Brussels Federation, UCLouvain, and University of Namur as part of the ARC project "RAINDROP". Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] R. Singh and S. S. Gill, "Edge AI: a survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, 2023.

[2] A. Ali-Eldin, B. Wang, and P. Shenoy, "The hidden cost of the edge: a performance comparison of edge and cloud latencies," in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2021.

[3] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX symposium on operating systems design and implementation*, ser. OSDI, 2020.

[4] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica, "Cilantro: Performance-Aware resource allocation for general objectives via online feedback," in *17th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2023.

[5] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *24th International Middleware Conference*, 2023.

[6] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for Kubernetes-oriented Edge-Cloud system," in *IEEE conference on computer communications*, ser. INFOCOM, 2021.

[7] K.-H. Chow, U. Deshpande, V. Deenadayalan, S. Seshadri, and L. Liu, "Atlas: Hybrid cloud migration advisor for interactive microservices," in *19th ACM European Conference on Computer Systems*, ser. EuroSys, 2024.

[8] Y. Cao, E. Riviere, and R. Sadre, "LASSY: a latency-aware slos-sufficing scheduling system for the cloud/edge continuum," in *25th IEEE Intl. Symposium on Cluster, Cloud, and Internet Computing*, ser. CCGrid, 2025.

[9] B. Wang, D. Irwin, P. Shenoy, and D. Towsley, "INVAR: Inversion aware resource provisioning and workload scheduling for edge computing," in *IEEE International Conference on Computer Communications*, ser. INFOCOM, 2024.

[10] G. Castellano, S. Galantino, F. Risso, and A. Manzalini, "Scheduling multi-component applications across federated edge clusters with Phare," *Open Journal of the Communications Society*, 2024.

[11] T. Pusztai, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris scheduler: Slo- and topology-aware microservices scheduling at the edge," in *IEEE/ACM 15th International Conference on Utility and Cloud Computing*, ser. UCC, 2022.

[12] Kubernetes: Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[13] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *ACM Symposium on Cloud Computing*, ser. SoCC, 2021.

[14] Q. Zhang, Q. Wu, M. Zhou, J. Wen, and S. Yao, "A communication contention-cognizant scheduling approach for workflow execution across public and private clouds," *IEEE Transactions on Automation Science and Engineering*, vol. 21, no. 4, 2023.

[15] Kubernetes horizontal pod autoscaling. Accessed: Jul. 15, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[16] Q. Wu, M. Zhou, and J. Wen, "Endpoint communication contention-aware cloud workflow scheduling," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 2, 2021.

[17] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *39th International Conference on Distributed Computing Systems*, ser. ICDCS. IEEE, 2019.

[18] Z. Xie, X. Xia, B. Hu, I. Khalil, Z. Wang, G. Cui, G. Xie, and M. Xue, "Optimizing energy efficiency with QoE-awareness in multi-access edge computing," in *33rd IEEE/ACM International Symposium on Quality of Service*, ser. IWQoS, 2025.

[19] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *IEEE/IFIP Network Operations and Management Symposium*, ser. NOMS. IEEE, 2018.

[20] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: a first look at public edge platforms," in *21st ACM internet measurement conference*, ser. IMC, 2021.

[21] Microsoft Learn. (2023) Azure network round-trip latency statistics. Accessed: 2024-10-19. [Online]. Available: https://learn.microsoft.com/en-us/azure/networking/network-performance-monitoring

[22] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, 2022.

[23] M. Bouet and V. Conan, "Mobile edge computing resources optimization: A geo-clustering approach," *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 787–796, 2018.

[24] S. Lee, Z. Guo, O. Sunercan, J. Ying, T. Kooburat, S. Biswal, J. Chen, K. Huang, Y. Cheung, Y. Zhou *et al.*, "Shard manager: A generic shard management framework for geo-distributed applications," in *ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP, 2021.

[25] D. Chou, T. Xu, K. Veeraraghavan, A. Newell, S. Margulis, L. Xiao, P. M. Ruiz, J. Meza, K. Ha, S. Padmanabha *et al.*, "Taiji: managing global user traffic for large-scale internet services at the edge," in *ACM SIGOPS 27th ACM symposium on operating systems principles*, ser. SOSP, 2019.

[26] Kubernetes Iptables Proxy Mode. Accessed: Jul. 15, 2025. [Online]. Available: https://kubernetes.io/docs/reference/networking/virtual-ips/

[27] A. K. Erlang, "Sandsynlighedsregning og telefonsamtaler," *Nyt tidsskrift for Matematik*, vol. 20, 1909.

[28] B. Donnet, B. Gueye, and M. A. Kaafar, "A survey on network coordinates systems, design, and security," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, 2010.

[29] Gurobi Optimizer. Accessed: Feb. 26, 2025. [Online]. Available: https://www.gurobi.com/

[30] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Upper Saddle River, NJ: Pearson, 2020.

[31] Scylla github repository. Accessed: May. 17, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.20259374

[32] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, dec 1998.

[33] A. Kay, "Tesseract: an open-source optical character recognition engine," *Linux J.*, vol. 2007, no. 159, jul 2007.

[34] EasyNMT. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/UKPLab/EasyNMT

[35] J. Tiedemann, M. Aulamo, D. Bakshandaeva, M. Boggia, S.-A. Grönroos, T. Nieminen, A. Raganato Y. Scherrer, R. Vazquez, and S. Virpioja, "Democratizing neural machine translation with OPUS-MT," *Language Resources and Evaluation*, no. 58, 2023.

[36] J. Tiedemann and S. Thottingal, "OPUS-MT — Building open translation services for the World," in *22nd Annual Conference of the European Association for Machine Translation*, ser. EAMT, 2020.

[37] L. Wu, W. A. Hanafy, A. Souza, K. Nguyen, J. Harkes, D. Irwin, M. Satyanarayanan, and P. Shenoy, "Carbonedge: Leveraging mesoscale spatial carbon-intensity variations for low carbon edge computing," in *34th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC, 2025.

[38] G. Perin, M. Berno, T. Erseghe, and M. Rossi, "Towards sustainable edge computing through renewable energy resources and online, distributed and predictive scheduling," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 306–321, 2021.