LASSY: a Latency-Aware SLOs-Sufficing Scheduling System for the Cloud/Edge 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*—Despite the advancements in cloud computing, cloud-hosted applications face significant network latency challenges, particularly for users distant from data centers. Edge computing has emerged as a solution to mitigate these issues by decentralizing processing, thereby reducing latency and enhancing user experience. However, the limited resources of edge data centers require careful scheduling of application instances to preserve the benefits of edge computing.**

This paper presents the Latency-Aware SLO-Sufficing Scheduling System (LASSY), a novel approach that considers network and queueing latencies in scheduling decisions for cloud/edge continuum environments. LASSY utilizes queueing theory to predict the tail latency experienced by users of latency-sensitive services such as Edge AI and optimizes service deployment across cloud and edge nodes to meet Service Level Objectives (SLOs). Our contributions include detailed latency modeling, an optimization algorithm that minimizes resource costs while ensuring SLO compliance, comprehensive experiments conducted on a testbed under realistic network emulation, and comparison with a state-of-the-art scheduling model. We evaluated LASSY under real-world latency conditions using two applications: a picture thumbnailing service and an Edge AI OCR service. We demonstrate LASSY's ability to achieve the desired service quality by effectively managing latency and resource allocation.

***Index Terms*—Cloud/Edge Computing, Instance Scheduling, Latency-Sensitive, Queueing Model**

I. Introduction

Cloud computing has transformed how we manage computing resources, offering scalability and efficiency. However, applications deployed in the cloud can face network latency issues for users far from cloud data centers, impacting user experience. Edge computing addresses this problem by decentralizing computing and data storage, bringing resources closer to users. This proximity significantly reduces latency and proves particularly advantageous for latency-sensitive applications such as AR/VR, IoT, and, more importantly, AI-powered services that have witnessed an explosive growth in demand [1].

However, integrating edge computing with existing cloud systems presents significant resource management and application scheduling challenges. Edge sites are geographically closer to users but typically have fewer computational and storage resources than cloud data centers. For edge computing to be beneficial, these properties must be considered when scheduling an application, i.e., when deciding at which locations of the cloud/edge continuum to provision resources for the application. While deploying an application on an edge site instead of in the cloud reduces the network latency for the users close to that site, the resources allocated at that site might not be sufficient to serve all requests in a timely manner, resulting in explosively growing response times. In the worst case, the resulting end-to-end latency[1] can be higher than for the same service deployed in the cloud [2]. The traditional cloud computing approach to improving response time, namely to scale the application out or up, is only possible to a limited extent at an edge site due to its aforementioned resource constraints. If there are multiple edge sites, another option is to redirect the users to a less busy but more distant edge site. Whether the improvement in processing time offsets the increased network latency must be weighed up. Also, the additional load on the more distant site may, in turn, worsen the response time for the users close to that site.

This shows that finding a global solution that delivers satisfactory performance to all users requires careful planning. Most existing work in latency-aware scheduling is either focused on cloud-only infrastructures [3]–[5] or optimizations between application components [6]–[8] and ignores the user distance to the cloud/edge continuum.

In this paper, we address the problem of scheduling latency-sensitive applications, such as AI services, on cloud/edge infrastructures and present LASSY, a new Latency-Aware SLOs-Sufficing Scheduler. LASSY computes placement plans where the end-to-end latencies of a service meet service level objectives (SLO) while minimizing the customizable operational costs. To this end, we formulate the scheduling problem as an optimization problem and use queueing models to predict end-to-end latency. In contrast to existing work based on the assumption that only the average latency is important [9], our scheduler can handle SLOs that impose tight limits on end-to-end *tail latency*, expressed as upper bounds on the k-th percentile of the latency. We validate the accuracy and effectiveness of LASSY in realistic experiments on a real-world testbed, along with two representative stateless appli-

[1]We define the end-to-end latency as the time from sending a request to receiving the response. We avoid the terms *service response time* and *end-to-end service time* sometimes used in the literature since they can be confused with similarly named concepts in queueing theory.

cations: a picture thumbnailing service and an AI-powered Optical Character Recognition (OCR) service.

The main contributions of our paper are:

1) We define the scheduling problem for the cloud/edge continuum with tail latency constraints as an optimization problem using queueing theory to predict the end-to-end latency distribution.
2) We present LASSY, our scheduler for latency-sensitive applications that computes SLO-respecting placement plans for the cloud/edge continuum while minimizing operational costs.
3) We evaluate our scheduler through experiments in a real-world testbed with two representative applications and demonstrate its effectiveness and efficiency. We chose and tested two representative optimization goals: minimizing the allocated resources (number of instances) or minimizing total operation cost. A comparison with the state-of-the-art approach proposed in the literature shows LASSY is better suited for scheduling latency-sensitive applications in realistic scenarios.
4) Our experiments show that the placement plans computed by LASSY for real-world applications can reduce the 99th percentile tail latency by up to 50% during high concurrency periods compared to other schedulers. Furthermore, we illustrate the robustness and effectiveness of LASSY under different optimization objectives in multi-site experiments.

This paper is organized as follows: Section II provides an overview of related work. Section III presents the context and the challenge of managing the cloud/edge continuum. Section IV describes our scheduling algorithm. Section V evaluates the performance of our scheduler in a Kubernetes testbed and provides a discussion of our approach. Section VI concludes the paper and presents future work.

II. Related Work

Academic literature has extensively explored approaches to optimizing the performance and usage of cloud/edge continuum infrastructures. These approaches can be broadly categorized into two groups: the first involves improving performance through more effective resource allocation, while the second focuses on optimizing application latency.

Wang *et al.* [9] propose the INVAR scheduler. Similar to us, they employ queueing theory to devise deployment plans. In contrast to our approach, their primary goal is to identify deployment plans that fit a given budget while minimizing the mean end-to-end latency across all users. The focus of INVAR on the cost and the mean latency is appropriate for the IoT setting targeted by the authors. Still, it is not helpful for other workloads, such as AI services and interactive applications, where the goal is to deliver a high Quality of Service (QoS), typically represented by a low tail latency [10], [11]. Users may experience unacceptably high latencies under peak load conditions, even though the mean latency appears satisfactory. Secondly, INVAR relies on a $M|M|c$ queueing model that assumes service times follow an exponential distribution and a single queue per site in which incoming requests wait to be dispatched to an available service instance. However, in modern cloud computing, particularly within containerized microservice environments, traffic is evenly distributed across all instances within a site [12], [13]. Furthermore, INVAR's evaluation is based on simulations using a proprietary parameter set, which does not allow drawing conclusions on its real-world applicability.

Mittal *et al.* [14] propose Mu for autoscaling serverless applications on resource-constrained edge sites. Its focus is on the latency inside the site; users' latency to the edge is not considered. Solutions to improve resource allocation have been proposed to avoid delays caused by resource depletion and competition for computing resources among applications. FIRM [3] ensures high utilization and reduced SLO violations by leveraging telemetry data and machine learning techniques. Similarly, Cilantro [4] uses online learning mechanisms to map resource allocation to performance outcomes in real-time, optimizing resource use for varying user-defined objectives. Sora [5] is a management framework that dynamically adjusts soft resource allocations (like server threads and database connections) for critical microservices. These works target cloud-only infrastructures.

Han *et al.* [6] propose a learning-based scheduling framework designed to optimize system throughput by employing a multi-agent actor-critic algorithm and graph neural networks for effective resource and request dispatch management. In their model, requests are forwarded from an edge site to the cloud if the edge site does not have sufficient resources, whereas our goal is to serve requests close to the user.

Adeppady *et al.* [15] propose the iPlace system to reduce the deployment delay for cloud-based applications. The Phare scheduler [16] focuses on scheduling multi-component applications across federated edge sites. Multi-component applications consist of components that perform inter-application API calls. To achieve good end-to-end latency for such applications, finding deployment plans that optimize the inter-application communication is essential. A similar work is done for microservices by Polaris [7] and Atlas [8]. Our work considers monolithic applications where the network latency to the user is important.

Existing solutions predominantly focus either on resource allocation across the continuum or solely on reducing latency without specifically considering the user location or the impact of precise tail latency constraints, which are crucial for both users and application providers. Moreover, the majority of these solutions is validated only through simulations rather than in actual cloud/edge infrastructure settings, rendering their findings less persuasive. A summary of the related work is presented in Table I.

III. Problem Statement

In this section, we will describe the targeted cloud/edge continuum scenario and explain the challenge of application scheduling that we address in this paper.

Name	Scope	End-to-end tail Latency	Real-world validation
INVAR [9]	Cloud/Edge	×	×
FIRM [3]	Cloud	×	✓
Cilantro [4]	Cloud	×	✓
KaiS [6]	Cloud/Edge	×	✓
SORA [5]	Cloud	✓	✓
Polaris [7]	Cloud/Edge	×	×
Atlas [8]	Cloud	×	✓
iPlace [15]	Cloud/Edge	×	×
Phare [16]	Cloud/Edge	×	×
Mu [14]	Cloud/Edge	×	✓
LASSY	Cloud/Edge	✓	✓

TABLE I: Related work summary.

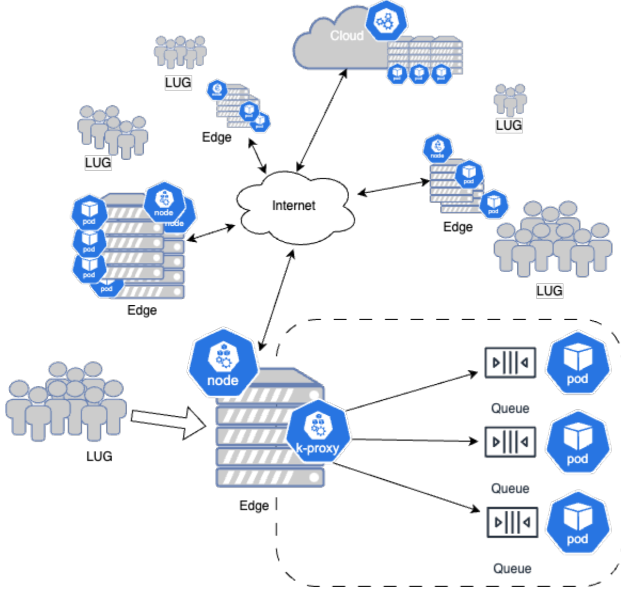

Fig. 1: A typical cloud/edge continuum infrastructure

A. The cloud/edge continuum

The cloud/edge continuum (see Figure 1) considered in this paper consists of multiple sites interconnected by dedicated connections or via the Internet. When a service has been deployed at a site, users can call it by sending requests and getting a response. Cloud sites are typically large data centers with massive computing resources, while edge sites are small data centers or server stacks with constrained resources. Geographically speaking, cloud sites are often far away from the densely populated areas where most users are located, while edge sites are closer. Consequently, the cloud has significantly higher network latencies for users than edge sites.

Because users are concentrated in densely populated areas, such as villages or cities, we can identify groups or clusters of users with similar network latencies to different cloud and edge sites. In the following, we call such a group of users a *Local User Group* (LUG) and we will treat it as a single entity. Various techniques are available to determine the network latencies and thus the LUGs, such as passive network measurements, active measurements (“ping”), or indirect methods based on network coordinates [17]. Other sources of information can also be used, such as a user’s location in a cellular network. If the network latencies experienced by a user significantly change, it can be reassigned to another, more similar LUG or form a new LUG with other users, consequently triggering a rescheduling. Since the concept of local user groups has been extensively used [9], [18] and justified in previous studies [19], [20], we consider the concrete mechanism to identify them to be outside the scope of this paper.

In this paper, we consider services providing AI-powered and similar functionalities to users. Such services can be efficiently implemented as stateless and monolithic applications that can be deployed across different sites as containerized instances. The statelessness implies that any instance can handle a request, an application model generally considered modern and efficient [2], [3], [5], [9]. Each instance is set to consume a fixed and identical amount of the available computing resources (CPU and RAM) at a site. To adapt to the workload, the application can be scaled out or in by adding or removing instances. However, we assume that the computation resources assigned to an instance are fixed, i.e., we do not scale up or down an instance.

Kubernetes [21] is a widely used, open-source container orchestration platform that automates containerized applications’ deployment, scaling, and management. In Kubernetes, a site’s physical or virtual machines are called *nodes*, and application instances deployed on the nodes are called *pods*. In Kubernetes, or systems derived from it, services are typically facilitated by a service proxy (i.e., kube-proxy), which by default dispatches incoming requests sent by users to service instances in a round-robin or random manner. If the instance to which the request has been dispatched is busy, the request will be queued at the instance, as depicted in Figure 1. This routing mechanism has a very low overhead.

B. Challenge

Application owners aim to provide their users with a certain quality of service. An SLO delineates the acceptable performance boundaries. A common way to set such a performance boundary is to define the maximum tolerated θth percentile eel^{θ}_{max} of the end-to-end latency, i.e., the SLO specifies that the probability that the end-to-end latency of a request is less than eel^{θ}_{max} must be greater than or equal to $\theta\%$.

To keep the network latency low for a specific group of users, the obvious place to deploy application instances would be at the edge site closest to that group. However, usage requests can exceed the capacity of an edge site during peak demand times [8], resulting in high queueing delays that negate the advantage of the low network latency. Scaling out an overloaded application, i.e., allocating more computing resources at the site, is often impractical due to resource constraints. Alternatively, one could send the requests to a more remote edge site with greater resource availability. While this would solve the problem of a lack of resources, it would also increase network latency.

LASSY’s goal is to produce a placement plan that determines (a) the number of application instances to be deployed at the different sites and (b) the mapping of users to sites

Name	Description
$st \in ST$	Cloud and edge sites
$nslots_{st}$	Number of slots of computing resource available at site st
$lug \in LUG$	Local user groups
$l_{lug,st}$	Round trip network latency between users lug and site st
λ_{lug}	Request rate of user group lug
μ	Service rate of an instance
c_{st}	Cost per instance at site st
eel^{θ}_{max}	Maximum tolerated θth percentile of the end-to-end latency defined in the SLO
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
$nappslots_{st}$	Number of instances deployed on each site st
$lugsite_{lug}$	Site that user group lug is redirected to
LUG_{st}	Set of user groups sending requests to site st
w_{st}	Queueing delay at site st
ρ_{st}	Utilization of an instance at site st

TABLE II: Model parameters (top) and variables (bottom)

(i.e., to which site a group of users should send its requests), such that the SLO is respected and operational costs are minimized. Determining such a placement plan is challenging due to two factors. First, the allocation of computing resources must ensure sufficient capacity to handle the workload while efficiently using the available limited resources. Second, the end-to-end latency of a service depends on several factors, namely, the network latency between users and sites, potential queueing delays, and the actual service processing time.

IV. Scheduling with LASSY

To determine an optimal placement plan, LASSY uses an optimization model in which infrastructure resources are modeled as queueing stations. In this section, we formalize the scheduling problem evoked in Section III and show how suitable scheduling fulfilling the SLO and minimizing operational costs can be obtained. The symbols used in the following explanations are summarized in Table II.

A. System and scheduling model

The computation infrastructure of the cloud/edge continuum consists of machines located at the cloud and edge sites. Each machine provides a certain amount of CPU and memory resources. In the model, we abstract from the individual machines and express the available resources at each site $st \in ST = \{st_1, \ldots, st_{|ST|}\}$ as a number $nslots_{st} \in \mathbb{N}$ of uniform computation "slots".

We consider a single monolithic application of which independent instances can be deployed across the infrastructure. Each instance needs exactly one resource slot. The number of slots at site st assigned to instances of the application is $nappslots_{st} \in \mathbb{N} \cup \{0\}$ with the constraint

$$nappslots_{st} \leq nslots_{st}. \quad (1)$$

For the given application, each slot worth of computation resources can serve user-issued requests at a rate of μ. We assume each request requires a constant service time $\frac{1}{\mu}$. This assumption holds for AI-powered services using ML models where the time to serve a request is dominated by a constant inference time independent of the request's content.

As motivated in Section III-A, a local user group $lug \in LUG = \{lug_1, \ldots, lug_{|LUG|}\}$ in our model 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 network latency between a group lug and site st is given by $l_{lug,st}$. Each group lug sends requests to exactly one site $lugsite_{lug} \in ST$. We assume that the number of requests sent by user group lug per time unit follows a Poisson distribution with rate $\lambda_{lug} \in \mathbb{R}_{>0}$. This follows from the idea that user groups represent large numbers of independently acting real-world clients. Requests from users to a site are randomly and evenly dispatched to the $nappslots_{st}$ application instances running at that site.

Scheduling the application means determining (a) how many resource slots to allocate for the application on which site and (b) which LUG should be served by which site. In our model, this means determining $nappslots_{st}$ for each site st and $lugsite_{lug}$ for each user group lug such that the SLO is fulfilled given the available resources.

Following the request dispatching mechanism of Kubernetes (see Section III-A), we model the instances at a site as independent queueing stations. Under the assumptions stated above, each of the $nappslots_{st}$ instances at site st can be modeled by an $M|D|1$ queue with service rate μ and arrival rate $\lambda_{inst,st}$. The arrival rate is given by

$$\lambda_{inst,st} = \frac{\sum_{lug \in LUG_{st}} \lambda_{lug}}{nappslots_{st}}, \quad (2)$$

where LUG_{st} is the set of LUGs sending requests to site st:

$$LUG_{st} = \{lug \in LUG \mid lugsite_{lug} = st\} \quad (3)$$

Since we assume that all instances receive an identical slot worth of computation, all requests processed by site st experience the same queueing delay distribution. The end-to-end latency of a request sent by lug to site $lugsite_{lug}$ is composed of the network latency $l_{lug,lugsite_{lug}}$, the queueing delay $w_{lugsite_{lug}}$ of the instance serving the request, and the constant service time $\frac{1}{\mu}$.

As explained in Section III-B, the SLO specifies that the probability that a request experiences an end-to-end-latency of less than eel^{θ}_{max} must be greater than or equal to a defined threshold $\theta\%$. This yields the following set of constraints:

$$P(l_{lug,lugsite_{lug}} + w_{lugsite_{lug}} + \frac{1}{\mu} \leq eel^{\theta}_{max}) \geq \theta\% \quad \text{for all } lug \in LUG. \quad (4)$$

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, similar to previous work [9], we assume that the network latencies between users and sites in the continuum are relatively stable. As explained in Section III-A,

the latencies can be monitored in real-time and the scheduling algorithm can be re-run to adapt to changes. This allows us to approximate $l_{lug,lugsite_{lug}}$ by the average network latency $\bar{l}_{lug,lugsite_{lug}}$ and to write the constraints (4) as

$$P(w_{lugsite_{lug}} \leq eel^{\theta}_{max} - \bar{l}_{lug,lugsite_{lug}} - \frac{1}{\mu}) \geq \theta\% \quad \text{for all } lug \in LUG. \tag{5}$$

The probability $P(w_{st} \leq \cdot)$ in constraints (5) is the queueing delay distribution of the $M|D|1$ queue. For instances at a site st, it is given by [22]

$$P(w_{st} \leq t) = (1-\rho_{st}) \sum_{i=0}^{\lfloor t\cdot\mu \rfloor} e^{-\lambda_{inst,st}(i/\mu - t)} \frac{(i/\mu - t)^i}{i!} \lambda^i_{inst,st} \tag{6}$$

where $\rho_{st} = \frac{\lambda_{inst,st}}{\mu}$ is the utilization of each instance at site st.

B. Scheduling as optimization problem

In general, the scheduling problem presented above will have multiple solutions. An optimization goal is therefore necessary to select the best solution. Since cloud/edge providers often charge fees based on allocated resources, we will focus on minimizing these costs, written as

$$\min \sum_{st \in ST} c_{st} \cdot nappslots_{st} \tag{7}$$

where c_{st} is the cost of using a slot at st. The constraints (1) and (5), together with the optimization goal (7), form a mixed-integer programming problem that has to be solved to obtain optimal slot allocations $nappslots_{st}$ for all sites st and site assignments $lugsite_{lug}$ for all user groups lug.

It should be noted that alternatives to goal (7) can be easily formulated. For example, the application owner could want to minimize the number of used sites, i.e., $\min |\{nappslots_{st} > 0\}|$. In fact, serving users from a small number of sites can be beneficial if deploying the application on a new site entails significant overheads or performance penalties (e.g., due to lower cache efficiency).

In our prototype implementation of LASSY, we solve the optimization problem with the Gurobi Optimizer [23]. The constraints (5) cannot be directly handled by Gurobi because Eq. (6) for $P(w_{st} \leq t)$ is not linear. To overcome this problem, the scheduler first determines the maximum arrival rate $\Lambda_{st,lug}$ that instances at site st can accept without breaking the SLO for user group lug (under the assumption that the site is used by that group) by solving the equation

$$P(w_{st} \leq eel^{\theta}_{max} - \bar{l}_{lug,st} - \frac{1}{\mu})\bigg|_{\lambda_{inst,st}=\Lambda_{st,lug}} = \theta\% \tag{8}$$

If the SLO can be met, this equation has a unique solution for $\Lambda_{st,lug}$ since the queueing delay increases monotonically with the arrival rate. LASSY calculates $\Lambda_{st,lug}$ for all possible pairs of sites and users with scipy's Brent solver [24]. This is fast even for large scenarios (e.g., less than 2.5 seconds for 40 LUGs and 30 sites on a consumer-grade Intel i7 CPU). Constraints (5) can then be replaced by the simpler constraints

$$x_{st,lug} \cdot \lambda_{inst,st} \leq \Lambda_{st,lug} \text{ for all } st \in ST, lug \in LUG \tag{9}$$

where $x_{st,lug}$ is a binary variable that has the value 1 if $lugsite_{lug} = st$, and 0 otherwise.

C. Practical aspects

The optimization problem has several parameters that have to be determined before scheduling, namely the request rates λ_{lug} of the user groups, the average network latencies $\bar{l}_{lug,st}$, and the service time $\frac{1}{\mu}$. As mentioned in Section III-A, we consider the concrete techniques to determine these parameters as outside the scope of this paper and only hint at popular solutions found in the literature. The network latencies can be estimated from network measurements, either directly or indirectly [17]. Arrival rates and the service time can be estimated from previous observations [14]. Mechanisms such as a sliding window can smooth the measurements and trigger rescheduling if the parameters have changed.

V. Experimental Evaluation

In this section, we provide a detailed experimental evaluation of LASSY in scenarios with increasing complexity. It seeks to answer the following questions:

- **Q1:** How accurate is LASSY's queueing model?
- **Q2:** How does our scheduler perform in a realistic scenario with multiple LUGs and edge sites and with more challenging setups, such as a burst of demands from certain LUGs?
- **Q3:** How does the scheduler perform under different optimization goals? We consider two goals in our experiments, namely, minimizing the number of instances or minimizing the total operation cost.

We support reproducibility and provide our code, the experimental setup, and instructions to reproduce our work at [25].

A. Experimental setup

1) Infrastructure: We perform our experiments on a dedicated, non-virtualized cluster of the Grid'5000 testbed [26]. Each node has two 8-core CPUs (Intel Xeon E5-2630 v3) and 128 GB of RAM. We deploy each actor of the system, i.e., a cloud/edge site or a LUG, on a dedicated physical node with exclusive access to its resources.

We deploy a Kubernetes cluster with a designated master node and multiple worker nodes. The master node, housing the control plane, is not used for workload execution to ensure the integrity of the experimental results. We control the latency between the LUGs and the different sites in the testbed with the `tc` command. Nodes at the cloud site are configured to have significantly higher network latency to users than the nodes at the edge sites.

2) LUGs: The user groups are emulated by dedicated nodes in the cluster running the `httperf` [27] load generator. This tool is one of the few workload generators capable of sending requests to a target service accurately following a Poisson distribution even at high request rates. The Poisson distribution is used here to emulate the behavior of a large number of independently acting individual users. Emulated network latencies are configured according to the observations of Ali-Eldin *et al.* [2]: 1 to 5 ms from a LUG to its closest edge site and 40 to 60 ms from a LUG to the cloud site.

3) Applications: We have selected two real-world applications for our experiments: thumbnailing and OCR. The thumbnailing application accepts a base64 string of an image and returns the resized image as a base64 string. The OCR application is based on the neural network-based OCR engine Tesseract [28] from Google, which takes a picture, recognizes any text shown in the picture, and returns the result to the user. The applications are deployed in containers that can be easily instantiated and managed by the Kubernetes cluster. Each instance is assigned 1 CPU core and 1 GB of RAM. The choice of these two applications is thoroughly considered. The thumbnailing application represents lightweight applications with very short service time, thus making the network latency clearly visible in the end-to-end latency. The OCR application represents a heavier AI-powered service commonly used in real-world scenarios.

The experiments described in the following sections are performed individually for each application and repeated for different request rate configurations. Each run lasts 5 minutes and is repeated three times. We drop the measurements taken during the first 30 seconds of each run to eliminate warm-up effects. For better readability, we do not show confidence intervals in the figures as the intervals are very small.

Where appropriate, we compare LASSY to the INVAR scheduler [9] that considers the mean end-to-end latency in its decisions. We also show results obtained by using Kubernetes' popular Horizontal Pod Autoscaler (HPA) [29] with autoscaling triggered when the CPU utilization reaches a certain threshold. To adapt it to the considered scenario of tail latency-sensitive applications, we set its threshold to 60% CPU utilization instead of the default 80%.

B. Model evaluation: Single-site scenario

In our first experiment, we evaluate whether the queueing model used by LASSY is suitable to predict the tail latency of requests sent to a site. To this end, we deploy a single instance and five instances, respectively, and use `httperf` to send requests following a Poisson distribution. The network latency is set to zero. We progressively increase the request rate for the thumbnailing and OCR applications. The service rates are 46 requests/s and 7.8 requests/s, respectively. We choose $\theta = 99$ to calculate the tail latency, a value that guarantees a high quality of service for most of the requests.

Figure 2 shows the mean and the 99th percentile (P99) end-to-end latencies (EEL), as measured in the empirical experiments in our testbed, as predicted by LASSY's $M|D|1$ model, and as predicted by the $M|M|c$ model used by the INVAR scheduler. Note that INVAR does not compute the tail latency nor consider it in its scheduling decision.

As can be seen, LASSY accurately predicts the tail latency for lower to medium request rates and overestimates it for (very) high request rates. INVAR's prediction of the average latency is, with an average relative error of 49% (LASSY: 10%), not accurate due to the exponentially distributed service time in their model and the fact that their single-queue-multiple-server model does not match how a real service proxy (see Section III-A) dispatches requests.

In our second experiment, we evaluate whether LASSY can determine the number of instances that are necessary to meet the SLO. We choose an objective of $\theta = 99$ and $eel^{\theta}_{max} =$ 100 ms for the thumbnailing application and $eel^{\theta}_{max} = 400$ ms for the OCR application and let the schedulers control the number of instances deployed in the testbed. We then measure the resulting end-to-end latency for different request rates. The network latency is set to 10 ms.

Figure 3 shows the empirical mean and P99 end-to-end latency measured in the testbed, the latency estimations of LASSY, and the number of deployed instances, as determined by LASSY. We observe that LASSY accurately determines how many instances are necessary in the testbed to ensure that the empirical P99 end-to-end latency stays below the threshold of 100 ms, respectively 400 ms.

In Figure 3, we also show results for INVAR and HPA. INVAR does not support latency SLOs, but could its prediction of the mean latency be used to calculate the number of instances? To answer this question, we give it enough budget to bring the mean latency below the threshold defined in the SLO. The results show that LASSY can reduce the tail latency by up to 30% for the thumbnailing service and up to 50% for the OCR service during high concurrency compared to INVAR and HPA. The number of instances calculated by INVAR and those scaled out by HPA are too low, resulting in a tail latency that significantly exceeds the SLO.

Since neither INVAR nor HPA can handle tail latency SLOs and optimize the number of instances for different objectives, we do not consider them further in the following experiments.

C. Model evaluation: Multi-site scenario

Our multi-site scenario features a cloud/edge infrastructure of one cloud site, three edge sites, and five LUGs. LUGs send requests to the application at different rates. The cloud site has ample computing resources, while edge sites are resource-constrained. The network latency setup in this scenario is based on the analyzed real-world data from a previous study [2], where the cloud site generally has a significantly higher network latency than the edge sites to the LUGs and some edge sites have higher latency than the cloud to certain LUGs. The number of slots of computing resources available at each site, the slot costs, and the network latencies are given in Table III. Note that the edge resources are more valuable and, therefore, more expensive. The SLOs for the two applications are identical to the ones used in Section V-B, i.e.,

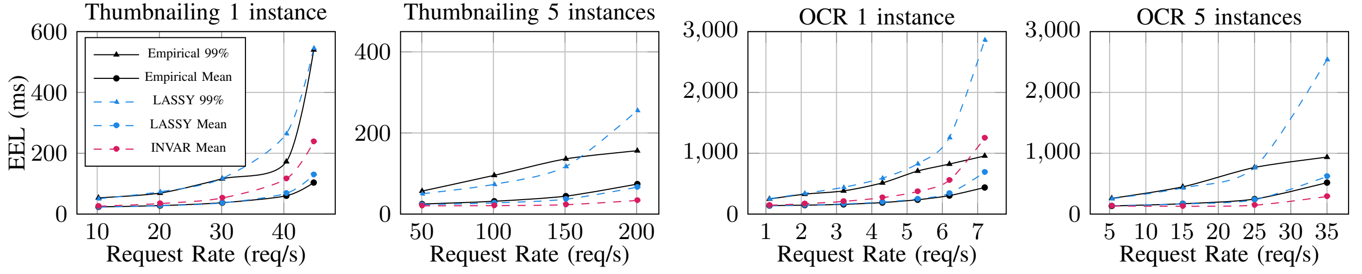

Fig. 2: Predicted and measured end-to-end latencies (EEL) for a single site with a fixed number of instances

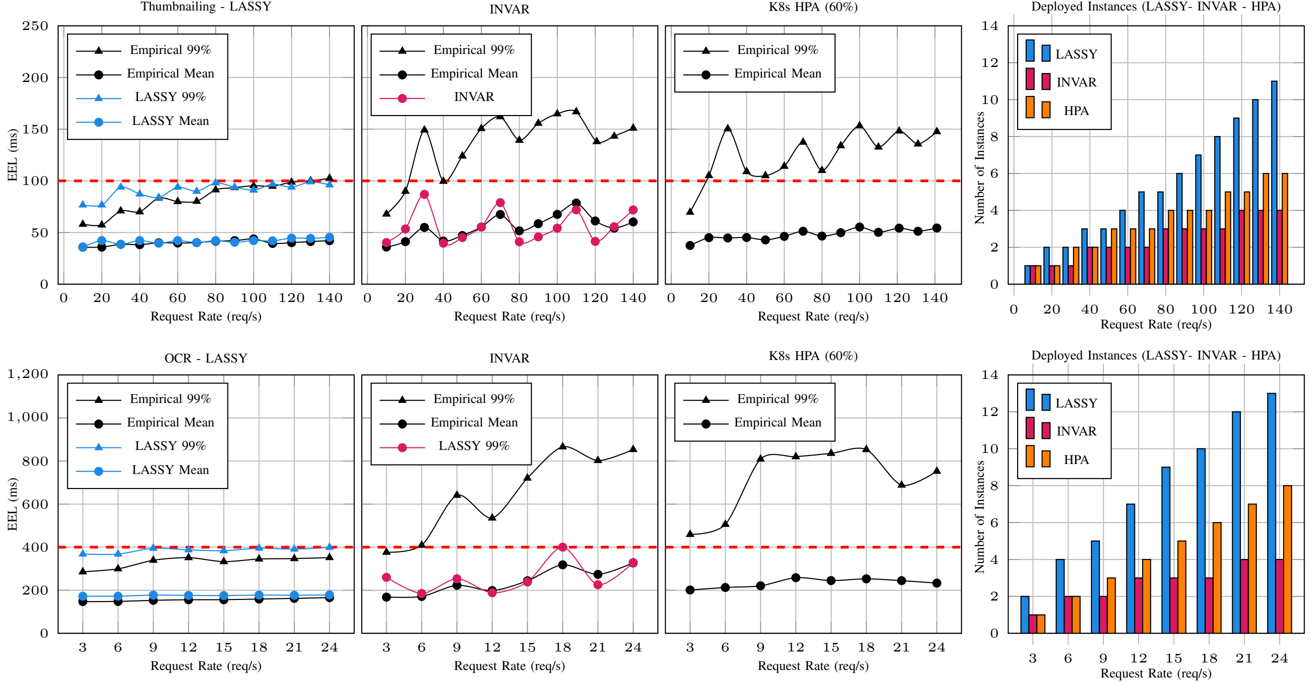

Fig. 3: Predicted and measured end-to-end latencies (EEL) for a single site scenario using LASSY, INVAR and HPA. The schedulers control the number of instances deployed for each application. Black curves indicate the empirical results for the P99 (triangles) and mean (dots) latency for each deployment, while blue and red curves indicate the predictions made by each scheduler. The dashed red line indicates each application's tail latency SLO for $\theta = 99$.

$\theta = 99$ and $eel^{\theta}_{max} = 100$ ms for the thumbnailing application and $eel^{\theta}_{max} = 400$ ms for the OCR application.

Setups	Slots	Pricing	Network latency (ms)				
			LUG1	LUG2	LUG3	LUG4	LUG5
Cloud	15	5	40	50	60	60	50
Edge 1	5	10	5	5	65	65	5
Edge 2	8	8	45	10	5	20	60
Edge 3	8	8	50	60	20	5	10

TABLE III: Resource/latency setup for multi-site experiments

We perform experiments for the thumbnailing and OCR applications with seven different configurations of LUG request rates, as shown in Table IV. The request rate is progressively increased for all LUGs in the first four configurations. In configurations #5 to #7, the request rate of LUG1 is drastically increased to emulate a burst of demand limited to a specific geographical location, for example, caused by a local event.

We test LASSY under two different optimization goals. The first one is $\min \sum_{st \in ST} nappslots_{st}$, i.e., the site-specific pricing is ignored, and the only objective is to minimize the number of instances while meeting the SLO. The second goal is $\min \sum_{st \in ST} c_{st} \cdot nappslots_{st}$ and uses the slot pricing shown in Table III. For each configuration and application, we let LASSY compute a suitable placement plan that indicates how many instances should be deployed at each site and to which site each group of users should send its requests. Following that plan, we run the testbed experiments and measure the resulting end-to-end latencies.

Figure 4 shows the observed mean and P99 tail latencies. As shown in the results, the placement plans computed by LASSY effectively succeed in keeping the tail latency below

Setups	Configurations	Request Rate (req/s)				
		LUG1	LUG2	LUG3	LUG4	LUG5
Thumbnailing	#1	5	5	5	5	5
	#2	15	15	15	15	15
	#3	30	30	30	30	30
	#4	45	45	45	45	45
	#5	60	45	45	45	45
	#6	80	45	45	45	45
	#7	100	45	45	45	45
OCR	#1	1	1	1	1	1
	#2	2	2	2	2	2
	#3	3	3	3	3	3
	#4	4	4	4	4	4
	#5	6	4	4	4	4
	#6	8	4	4	4	4
	#7	10	4	4	4	4

TABLE IV: Request rate setup for multi-site experiments

the SLO threshold in most cases for both optimization goals. In the cases where it fails, the tail latency is only slightly above the threshold. This shows that LASSY can generate suitable placement plans for a wide range of scenarios while optimizing operational costs.

Figure 5 shows the empirical cumulative distribution of the end-to-end latency for the configurations #3, #5, and #7. The curve quickly flattens for larger latencies, making the percentile estimation sensitive to small errors. Despite this difficulty, LASSY demonstrates excellent performance.

Figure 6 visualizes the placement plans computed by LASSY under different optimization goals. As the number of requests increases, more and more application instances are deployed. For the experiments aiming to minimize the number of instances, it can be seen that LASSY only deploys a significant number of instances on the cloud site when the request rate is very high (configuration #7) since edge sites offer better network latencies when the resource is enough. This behavior changes when the goal is to minimize costs. The cloud site has enough resources to provide fast response times that compensate for its higher network latency to the users, and because it is cheaper than the edge sites, it is preferred by the scheduler. The results demonstrate LASSY's flexibility in generating placement plans based on different goals.

Finally, Figure 7 shows the time needed by LASSY to compute a placement plan for the experimental setup with four sites and five LUGs. We have also included the times for significantly more LUGs in the figure. They remain within reasonable limits in all practical cases.

D. Discussion

The experimental results show that LASSY can efficiently schedule applications on cloud/edge infrastructures to meet tail latency objectives and minimize operational costs.

Our model of multiple independent $M|D|1$ queueing stations describes well the behavior of a real-world system using Kubernetes' default dispatching mechanism and running a monolithic AI-based (or similar) service. In other scenarios, it is possible that LASSY overestimates or underestimates the number of needed instances. If the service time is not constant, a potential solution is to perform an upper-bound estimation of the service time from past observations and let LASSY compute a placement plan that allocates enough instances to cover the worst-case situation. Unfortunately, no closed expressions similar to Eq. (6) are available for general service time distributions.

VI. Conclusion and Future Work

We have presented LASSY, a new model-based application scheduler for the cloud/edge continuum. LASSY uses queueing theory to predict the tail end-to-end latency of the application and uses an optimization solver to decide on instance allocations over the cloud/edge sites that minimize operational costs. We validated LASSY through extensive experiments consisting of scenarios with real-world configurations and representative workloads in a distributed testbed. Our experimental results show that LASSY accurately predicts the necessary resources to meet tail latency objectives provided by the application and, when compared to state-of-the-art solutions, it can reduce the P99 tail latency by up to 50% during high concurrency while minimizing operational costs within a reasonable computation time.

We identify several research directions for future work. We want to investigate more complex optimization goals that consider energy consumption and utilization of GPU resources. Secondly, this paper focuses on stateless services that do not rely on information from previous requests. We aim to extend our application model to support stateful services. Lastly, we aim to support the simultaneous scheduling of multiple applications over a shared infrastructure. The literature has shown that the competition for computing resources in a multi-tenant environment can result in performance degradation [8], which we will have to consider.

VII. Acknowledgments

We thank the anonymous reviewers for their valuable feedback, and Pierre Schaus for his advice on an early version of this work.

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 *International Conference for High Performance Computing, Networking, Storage and Analysis (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 (OSDI)*, 2020.

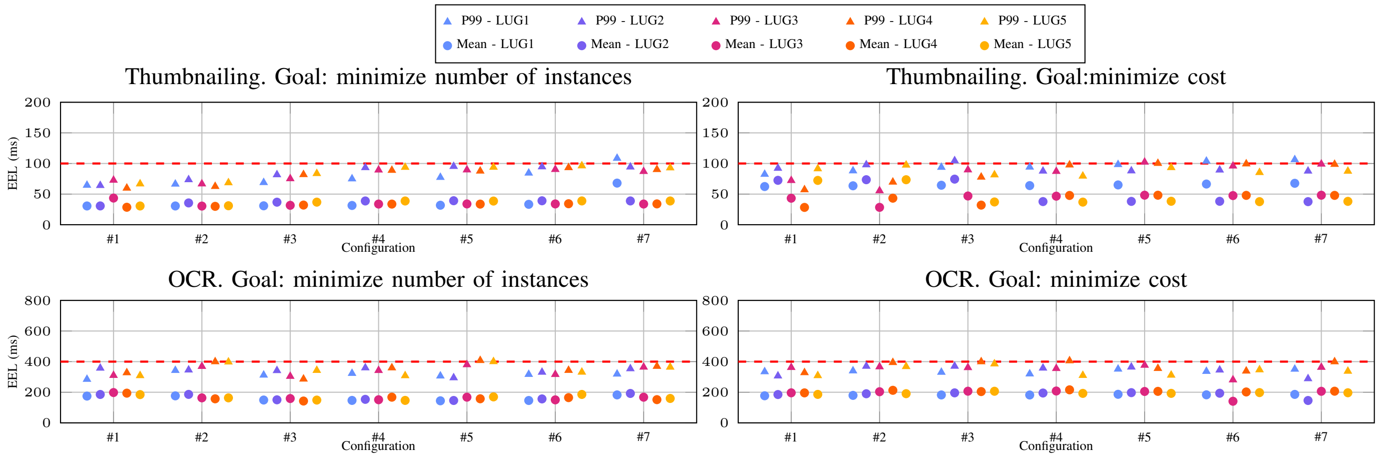

Fig. 4: Empirical mean and P99 end-to-end latency (EEL) for different LUGs, different multi-site configurations, different optimization goals, and the two applications. The dashed line indicates the tail latency SLO for $\theta = 99$.

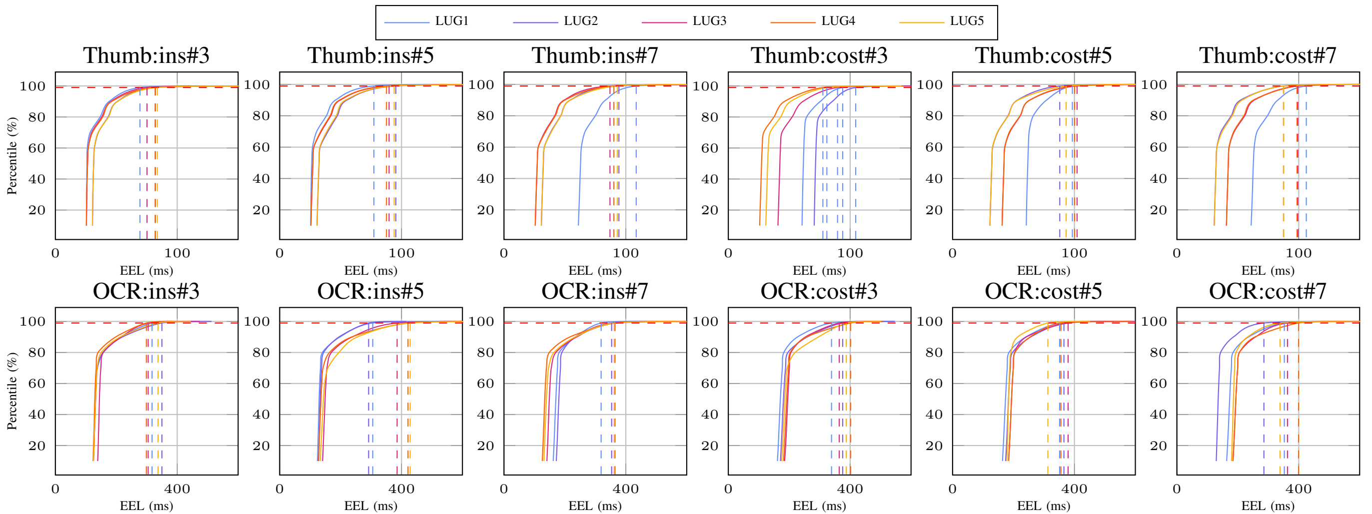

Fig. 5: End-to-end latency distribution under different multi-site configurations for the two applications and the two optimization goals. The label "Thumb:ins#3" means "Thumbnailing application, minimizing the number of instances, configuration #3". The dashed lines indicate the P99 end-to-end latency for the different LUGs.

[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 (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 INFOCOM conference on computer communications*, 2021.

[7] 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 (UCC)*, 2022.

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

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

[10] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, 2013.

[11] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, vol. 61, no. 8, 2018.

[12] M. Autili, A. Perucci, and L. De Lauretis, "A hybrid approach to microservices load balancing," *Microservices: Science and Engineering*, 2020.

[13] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM Conference on Computer Communications*, 2018.

[14] 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*, 2021.

[15] M. Adeppady, P. Giaccone, H. Karl, and C. F. Chiasserini, "Reducing microservices interference and deployment time in resource-constrained cloud systems," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, 2023.

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

[17] B. Donnet, B. Gueye, and M. A. Kaafar, "A survey on network coordi-

Fig. 6: Placement plans of the instances and user-to-site mapping under different configurations and optimization goals. The discs represent the number of instances allocated per site. The diamond markers represent the LUGs assigned to the sites. LASSY could effectively gives different placement plans based on different optimization goals.

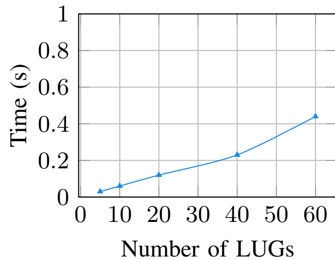

Fig. 7: Computation time of LASSY for four sites

nates systems, design, and security," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, 2010.

[18] Q. M. Tran, P. H. Nguyen, T. Tsuchiya, and M. Toulouse, "Designed features for improving openness, scalability and programmability in the fog computing-based iot systems," *SN Computer Science*, vol. 1, no. 4, 2020.

[19] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *IEEE GLOBECOM Global Communications Conference*, 2020.

[20] H. Zhang, S. Huang, M. Xu, D. Guo, X. Wang, V. C. Leung, and W. Wang, "How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild," in *IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, 2023.

[21] Kubernetes: Production-grade container orchestration. Accessed: Feb. 26, 2025. [Online]. Available: https://kubernetes.io/

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

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

[24] Brent. Accessed: Feb. 26, 2025. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brent.html

[25] LASSY github repository. Accessed: Mar. 10, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15000547

[26] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science. Springer International Publishing, 2013, vol. 367.

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

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

[29] Kubernetes horizontal pod autoscaling. Accessed: Feb. 26, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/