Occamy: Latency-Aware Scheduling with Controlled Degradation for Mixed-Criticality AI at the Edge

Yinan Cao*, Patient Ntumba†, Etienne Rivière*, Ramin Sadre*, Stefano Secci†
*UCLouvain, Louvain-la-Neuve, Belgium. firstname.lastname@uclouvain.be
†Cnam, Paris, France. firstname.lastname@cnam.fr

***Abstract*—We study edge AI inference systems deployed across geographically distributed edge sites and serving latency-sensitive requests from spatially distributed users. Each application can feature multiple DNN model variants with different trade-offs among inference quality, inference time, and resource usage. When resources are insufficient, these variants enable graceful degradation of inference quality while maintaining strict adherence to tail latency Service Level Objectives (SLO). We present Occamy, a priority-aware scheduler that iteratively searches for a globally optimal deployment and introduces graceful degradation when necessary. For each application, Occamy jointly considers DNN model variant selection, placement across available edge sites, and scaling of application instances. This multi-dimensional optimization ensures continuity of service under P99 SLOs, even under resource pressure and scarcity. We evaluate Occamy on a real distributed edge testbed using Kubernetes, with realistic network latency and user request traces. Our results show that Occamy effectively preserves P99 latency guarantees for high-priority applications, applies minimal degradation to low-priority applications when necessary, and improves overall resource efficiency compared to state-of-the-art schedulers for edge workloads.**

***Index Terms*—Edge Computing, Controlled Degradation, Instance Scheduling, Latency-Sensitive, Queuing Model**

I. Introduction

Edge computing has gained significant momentum over the past decade, driven by the need to move computation closer to end users to reduce network latency, reduce pressure on cloud resources, and address growing concerns about data sovereignty and privacy. Several edge infrastructure initiatives have emerged [1], [2], and telecom operators are also integrating edge resources into 5G and beyond network infrastructures [3]. The proximity of edge infrastructures to users particularly benefits applications that require fast response and define their Service Level Objectives (SLOs) in terms of the tail latency they can tolerate, typically expressed as a bound on a high-percentile (e.g., P99) of the latency distribution. An increasing portion of these workloads consists of inference tasks using various AI models. We focus on AI applications relying on deep neural networks (DNNs) in this work.

Despite being well-suited for latency-sensitive applications, edge infrastructures are inherently resource-constrained compared to traditional cloud data centers and can quickly become saturated under high load, leading to performance degradation [4]. These limitations are particularly critical for AI applications considered in this work, as DNN-based inference is computationally demanding. These computational requirements make it challenging to meet latency SLOs in saturated edge environments. Existing DNN optimization techniques, such as compression and pruning [5], or quantization [6], can reduce computational costs, but they typically target a single DNN for a specific hardware or deployment configuration.

In this work, we consider AI applications that offer multiple DNN model variants, each offering a different trade-off among inference quality, inference time, and resource cost. We propose adaptive application execution through controlled degradation, in which a lighter, less accurate model variant can be deployed under high computational demand to satisfy application SLOs, albeit at the cost of reduced model inference quality. Here, inference quality refers to the task-specific performance of a DNN model, measured using standard AI metrics (e.g., classification accuracy or mean average precision), while inference time denotes the time required to process an input and produce an output [7].

Furthermore, we consider an edge infrastructure provider that operates multiple geographically distributed edge sites, allowing deployment of applications to serve users in different locations. In this context, applications can have different priorities depending on their criticality. For example, a high-priority application could be real-time video analytics for public safety, processing live camera feeds to detect accidents or other unusual events. A lower-priority application could be augmented reality (AR) content recommendation, which analyzes user context and preferences to provide AR experiences in venues such as museums or exhibitions.

During peak load, the resource demand from all applications may exceed what the edge infrastructure can provide, resulting in contention among co-located applications. This often leads to service disruptions and widespread violations of tail-latency SLOs for AI applications [8], [9]. To mitigate such violations, AI applications may attempt to scale out by launching additional instances at neighboring edge sites. While this can increase resource availability, it inevitably also increases network latency, exacerbates contention, and may trigger cascading SLO violations across the system. In such cases, edge infrastructure providers may need to evict low-criticality AI applications to free resources for higher-priority ones [10]–[12]. However, such evictions significantly degrade user experience and can destabilize the system due to frequent reallocations and churn.

Contribution. We propose addressing resource contention in distributed edge environments hosting latency-sensitive AI

applications by jointly considering application priorities, tail latency SLOs, and the availability of multiple DNN model variants. Rather than evicting applications when resources become scarce, we exploit controlled, latency-aware degradation to maintain system feasibility and service continuity. We realize this approach with Occamy, a scheduler that computes globally near-optimal deployment plans for AI applications across geographically distributed edge sites, based on accurate models of end-to-end latency and awareness of heterogeneous resource availability. By adapting application execution to changing resource conditions, Occamy enables stable and efficient operation under both normal and peak workloads. Occamy builds upon the following contributions:

1) We develop a queueing-theoretic model that accurately predicts P99 user-perceived request latency by jointly capturing network, queuing, and service (inference) times under contention, enabling enforcement of end-to-end tail latency SLOs.
2) We formulate the global scheduling problem of AI applications as an optimization problem that enforces latency SLOs and edge resource capacity constraints while jointly minimizing customizable deployment cost metrics. We map this problem to a multi-dimensional multiple-choice knapsack problem (MMKP) and show it is NP-hard.
3) We design a priority-aware scheduling algorithm that restores feasibility under resource scarcity via model-switching, i.e., degradation of inference quality under priority constraints, ensuring that low-criticality applications absorb resource shortfalls before higher-criticality ones.
4) We evaluate Occamy on a real distributed edge testbed through a series of experiments spanning multiple scenarios of progressively increasing complexity, using network latencies, representative AI applications, and user-request traces derived from a real-world dataset [13]. Occamy accurately predicts application latencies and computes deployment plans that satisfy P99 latency SLOs, using model switching only when necessary to minimize inference quality degradation. In contrast, state-of-the-art baselines violate tail-latency SLOs under the same experimental conditions: INVAR [14] and Kubernetes HPA [15] incur violations of up to 418% and 198% in the single-site scenarios, while MVSP [16] exhibits violations exceeding 500× the SLO in multi-site scenarios, due to its lack of latency modeling. Local-first heuristic leads to highest resource cost and infeasibility under resource scarcity.

The paper is organized as follows. Section II reviews related work, Section III presents the system model, the problem challenges, and the problem formulation. Section IV describes the proposed Occamy algorithm. Section V evaluates its effectiveness. Finally, Section VI concludes the paper.

II. Related work

We analyze related work on latency-sensitive and mixed-criticality applications, as well as controlled application degradation. Table I summarizes the gap of the related work with respect to Occamy.

Work	Model Variants	Controlled Degradation	Distributed Edge	P99 Latency Prediction	Mixed-Criticality
CLOVER [17]	✗	✓	✗	✓	✗
EcoEdgeInfer [18]	✗	✗	✗	✓	✗
HyperDrive [19]	✗	✗	✓	✗	✗
INFaaS [20]	✗	✓	✗	✗	✗
InfAdapter [21]	✗	✓	✗	✓	✗
InferRouter [22]	✓	✓	✗	✓	✗
LayerCake [23]	✓	✓	✗	✓	✗
MVSP [16]	✓	✗	✓	✗	✗
Occamy	✓	✓	✓	✓	✓

TABLE I: Comparison of Occamy with related work.

Scheduling of latency-sensitive applications. A large body of work focuses on the scheduling and placement of latency-sensitive applications across cloud and edge infrastructures. Cloud-oriented schedulers leverage autoscaling, request replication, or feedback-driven resource allocation to minimize tail latency for multi-tenant applications, often assuming abundant computing resources and stable network conditions [24]–[26]. In the edge-cloud context, scheduling becomes more constrained: systems must balance limited compute capacity, dynamic network latency, and geographically distributed demand. Prior research explores queueing-theoretic scheduling, predictive resource provisioning, and distributed placement strategies to satisfy tail latency (i.e., P95/P99) SLOs for inference and interactive workloads [14], [17], [27]–[30]. These works primarily rely on cloud resources to absorb excessive workloads when edge resources are insufficient. Furthermore, although they address multiple application scheduling, none of them consider integrating criticality-aware application management with model variant selection. More recently proposed, HyperDrive [19] is an SLO-aware scheduler for serverless functions across a distributed edge-cloud-space continuum. It does not, however, target model-variant degradation.

Scheduling mixed-criticality applications. Scheduling applications with different priorities has been widely studied in real-time and embedded systems, where workloads with varying priority or safety levels must coexist on shared resources. Classical mixed-criticality approaches ensure that high-criticality applications meet their SLO even under overloaded resources, often by reducing or suspending low-criticality tasks [12]. More recent work extends mixed-criticality concepts to cloud and edge environments, adapting resource allocation policies to enforce differentiated service guarantees under contention. Priority-aware or hierarchical schedulers allow high-criticality applications to reserve resources or preempt lower-criticality ones [10], [11], while still maintaining degraded but uninterrupted service. In contrast, we focus on AI-based applications with multiple model variants, in which service times, inference quality, and resource footprints vary with the selected model. Our approach addresses the scheduling of mixed-criticality AI applications by leveraging these variant-dependent characteristics to enable controlled application degradation, ensuring continuous service under resource constraints.

AI application degradation. Application degradation allows applications to continue operating under reduced performance

or limited features [8], [12]. For DNN inference applications, three main approaches exist [31]: inference quality degradation, which switches to lower-quality model variants to reduce resource use; network latency degradation, which uses distant nodes to access more resources at the cost of higher latency; and traffic rate reduction, which lowers request wait times but requires replication for service continuity. Each approach trades off resource usage, latency, and quality.

Several systems implement these ideas. INFaaS [20] automatically generates and selects model variants based on inference quality and latency requirements, using ILP-based autoscaling for large-scale inference. CLOVER [17] dynamically adjusts mixtures of application model variants to meet strict P99 latency SLOs, while InfAdapter [21] formulates an ILP to maximize AI application quality under the same constraints. These systems assume centralized cloud resources and concurrent execution of model variants, which is often infeasible at the edge.

Furthermore, InferRouter [22] achieves local graceful degradation on a *single* edge node by jointly using multiple model variants when predicted queueing delays threaten P99 latency SLOs, minimizing tail latency while maintaining high average inference quality. LayerCake [23] routes AI inference requests across heterogeneous resources, including edge node (on-device) and cloud node, by selecting the model variant and execution location that maximizes inference quality while satisfying latency constraints. EcoEdgeInfer [18] dynamically tunes hardware and software parameters to jointly optimize P99 inference latency and energy consumption on a single edge device, but does not consider multi-site placement, model variant selection, or mixed-criticality scheduling.

Closest to our work is MVSP [16], a heuristic for model variant selection and placement in distributed edge environments subject to end-to-end latency constraints. The approach incrementally places and replicates applications to meet resource and workload demands, evaluating solutions on resource cost and average latency. Unlike our approach, however, it does not address controlled degradation, mixed-criticality workloads, or P99 latency prediction.

III. Problem Statement

This section introduces our system model, highlights key challenges, and formulates our global scheduling problem for AI applications on geographically distributed edge sites.

A. System model

The edge infrastructure model that we consider aligns with the model used by related work [4], [14], [27], [32], [33]. An edge infrastructure provider operates a set of geographically distributed edge sites ST, strategically located in or near metropolitan areas to reduce end-to-end latency and serve dense user populations. User requests in these regions are typically spatially correlated, resulting in groups of users who experience similar network latencies [34]. We model these as user groups LUG. Requests from a user group can be served by any single edge site, provided the corresponding AI application has been scheduled and deployed on that site.

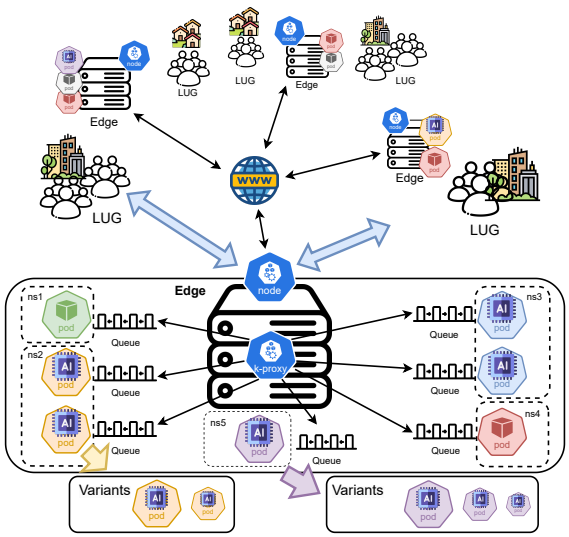


Fig. 1: A typical edge infrastructure with AI workloads and their variants.

Each site $st \in ST$ offers limited computing capacity $nslots_{st}$, expressed in normalized resource slots (CPU and memory). Edge sites host a set of AI-based applications APP, deployed as stateless containerized services.

Each application $app \in APP$ may offer $D_{app} \geq 1$ model variants that trade off inference quality, latency, and resource consumption. A variant is indexed by its degradation level $d \in \{1, \ldots, D_{app}\}$, where higher degradation levels correspond to lighter, faster, but less accurate variants. Let σ^d_{app} be the number of resource slots one instance of app requires at level d. Applications with $D_{app} = 1$ do not support degradation and only run a single model.

In this work, priority reflects the joint criticality of an application with respect to both inference quality and latency tolerance: a high-priority application is one for which neither inference quality degradation nor latency violations are acceptable, while a low-priority application can tolerate switching to a lower-quality model variant under resource pressure while still meeting its latency SLO. This joint treatment is natural for DNN inference workloads, where model variants simultaneously affect inference quality, inference time, and resource footprint [7]: an application that is sensitive to inference quality loss is typically also the one whose service is most critical to preserve end-to-end. The priority level $prio_{app}$ is assigned by the infrastructure provider based on their knowledge of application requirements, and is not an intrinsic property of the scheduling algorithm.

When resources are scarce, lower-priority applications may operate in degraded mode by switching to a less resource-intensive model variant, while higher-priority applications are preserved at their nominal inference quality. This degradation capability is a first-class property of the system: each appli-

cation exposes it through its set of model variants and its assigned priority level.

We target deployments using Kubernetes [35], a widely used open-source orchestration platform for containerized applications that is representative of modern industry practice for managing stateless services at the edge. 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* [36], routes incoming user requests to individual pods using a random dispatch policy. Once assigned, a request is queued locally at the selected pod with no subsequent rebalancing, as illustrated in Figure 1.

B. Challenges

The key challenge is globally scheduling AI applications across geographically distributed edge sites under fluctuating demand, limited capacity, and heterogeneous priorities: when a site overloads, the system must choose between adjusting behavior locally (reducing replicas or switching model variants by priority) and offloading to more distant, less-loaded sites, but offloading raises network latency, tightening the latency budget for queueing and inference and often requiring more instances to absorb the same load, so decisions at one site affect the feasibility and cost at others and no site can be optimized in isolation.

When the total resource demand across applications exceeds the available capacity of edge sites, no placement plan can simultaneously satisfy all SLOs. A naive response would be to evict lower-priority applications entirely, but this disrupts service availability and damages user experience. Instead, lower-priority applications should continue operating in degraded mode, switching to lighter model variants that consume fewer resources, while higher-priority applications are preserved at their nominal inference quality. This inference-quality-aware degradation allows the system to absorb demand fluctuations without resorting to eviction or service interruption. However, it significantly complicates scheduling decisions, which must now jointly determine placement, model variant selection, and priority-based degradation ordering across all applications.

Each application maintains a request queue and processes inference requests arriving from multiple LUG. As a result, the end-to-end latency users experience is a combination of network latency, queuing delay, and inference time. The scheduler must therefore jointly determine where applications are placed, how many resources they are allocated, and which model variants they execute, so as to ensure that high-priority applications consistently meet their latency objectives, lower-priority applications degrade gracefully rather than being dropped, and overall edge resource utilization is efficient.

C. Problem formulation

We model the global scheduling of AI applications across geographically distributed edge sites as an optimization problem. We formulate a hierarchical optimization problem that captures deployment constraints and end-to-end tail latency SLOs using queueing theory, ensuring priority-based service preservation while minimizing deployment costs. Given the resource constraints at the edge, the scheduler follows an iterative process: it evaluates the feasibility of hosting AI applications at their highest inference quality and systematically applies degradation based on criticality when necessary. Within each iteration, the objective is to minimize total resource usage costs across edge sites while satisfying strict tail-latency SLOs for the selected model variants. The parameters and decision variables are summarized in Table II.

Name	Description
$st \in ST$	Edge sites
$nslots_{st}$	Available resource slots at site st
$lug \in LUG$	Set of user groups
$app \in APP$	AI applications
$lugapp_{lug}$	Application used by LUG lug
D_{app}	Number of model variants per app
$d \in [1, D_{app}]$	Degradation level / model variant index
σ^d_{app}	Slots demand of app at degradation level d
μ_{app}	Service rate of application app instance
$prio_{app}$	Priority level of application app
$APP_p \subseteq APP$	Set of applications belonging to priority class p
K	Total number of priority classes
$l_{lug,st}$	Network latency between lug and site st
eel^{max}_{app}	Max. P99 end-to-end tail latency SLO per app
λ_{lug}	Request rate of user group lug
$\Lambda_{st,lug}$	Maximum arrival rate for the instances at site st can accept without breaking the SLO for user group lug
$ninsts_{st,app}$	Number of instances of app deployed at site st
$lugsite_{lug}$	Edge site assigned to serve user group lug
$z_{st,lug}$	Whether user group lug is assigned to site st
$w_{st,app}$	Queueing delay for application app at site st
$L_{st,app}$	Set of user groups sending requests for app to site st
y^d_{app}	1 if app runs at degradation d, 0 otherwise

TABLE II: System model parameters and decision variables.

1) Resource constraints: To schedule AI-based applications with heterogeneous priorities and user demand across multiple edge sites, the scheduler determines the number of instances, $ninsts_{st,app}$, of application app to deploy on each site st. The limited computational capacity of each site constrains these decisions. Specifically, the total resource consumption of all deployed application instances must not exceed the available resource slots at site st, as expressed by Constraint (1) (σ^d_{app} reflects the resource consumption of application app at degradation level d):

$$\sum_{app \in APP} ninsts_{st,app} \cdot \sigma^d_{app} \leq nslots_{st}, \quad \forall st \in ST \qquad (1)$$

2) End-to-end tail latency SLO: We capture SLO requirements as end-to-end latency constraints. For each application $app \in APP$, we denote by eel^{max}_{app} the maximum acceptable end-to-end tail latency, independent of the application priority and degradation level. The end-to-end latency experienced by requests from local user group lug served by application app

at edge site st is decomposed into three components: (i) the round-trip network latency $l_{lug,st}$ between lug and site st, (ii) the service time $\frac{1}{\mu_{app}}$ of an application instance with service rate μ_{app}, and (iii) the queueing delay $w_{st,app}$ incurred under FIFO request processing at service rate μ_{app}.

We assume that each user group lug generates requests with average rate λ_{lug} and is assigned to exactly one application $lugapp_{lug} \in APP$ and one serving site $lugsite_{lug} \in ST$, without loss of generality, since the same population of physical users can be represented by multiple user groups targeting different applications and sites. As each user group aggregates a collection of real-world clients, we model the resulting request arrivals from lug to application app as a Poisson process with rate $\lambda_{lug,app} \in \mathbb{R}_+$.

The SLO requirement 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 the desired target threshold, which we set, without loss of generality, to 0.99 in the following. This can be denoted by the constraint:

$$\mathbb{P}(l_{lug,st} + w_{st,app} + \frac{1}{\mu_{app}} \leq eel_{app}^{max}) \geq 0.99 \text{ for } lug \in LUG \tag{2}$$

On the network link between a user group lug and an edge site st, we assume that request and response sizes are small compared to the available bandwidth. Moreover, network latencies are relatively stable, allowing us to approximate them with a constant equal to their average, denoted by $\bar{l}_{lug,st}$.

For each application, the service time $\frac{1}{\mu_{app}}$ depends on the selected degradation level, i.e., on the deployed model variant. We assume the service time is deterministic for a given application and degradation level d. This assumption is justified by DNN models' characteristics, such as the number of layers and input resolution are fixed for a given variant (e.g., YOLOv8 [37] or ResNet [38]), which leads to nearly constant inference times with very small variance [7]. Moreover, this deterministic behavior is reinforced by the fact that all application replicas are allocated identical computing resources based on the associated model variant's resource demand.

Given that service time and network latency are deterministic, each application instance can be accurately modeled as an M/D/1 queueing system with service rate μ_{app}. Arrivals follow a Poisson process with rate $\lambda_{st,app}^{inst}$, defined as:

$$\lambda_{st,app}^{inst} = \frac{\sum_{lug \in L_{st,app}} \lambda_{lug}}{ninsts_{st,app}}, \tag{3}$$

where $L_{st,app} = \{lug \in LUG \mid lugsite_{lug} = st \wedge lugapp_{lug} = app\}$ is the set of LUGs sending requests to app on site st.

Following the request dispatching mechanism of Kubernetes (see Section III-A), the corresponding queueing time $w_{st,app}$ is non-stationary, as it depends on the request arrival rate $\lambda_{lug,app}$. Therefore, the waiting time becomes the only component that must be controlled to satisfy the SLO constraint in Constraint (2). Accordingly, from the same constraint, we deduce a time budget T, defined as the maximum allowable waiting time within the end-to-end latency bound eel_{app}^{max}:

$$T = eel_{app}^{max} - \bar{l}_{lug,st} - \frac{1}{\mu_{app}}, \tag{4}$$

The end-to-end tail latency SLO constraint in Constraint (2) requires that the queueing delay distribution of requests processed by an application app does not exceed the residual latency budget T with a probability of at least 0.99. For an M/D/1 queue, the probability distribution of the waiting time can be approximated using the classical Erlang formula [39], combined with Equation (4):

$$\mathbb{P}(w_{st,app} \leq T) = \left(1 - \frac{\lambda_{st,app}^{inst}}{\mu_{app}}\right) \cdot \sum_{r=0}^{\lfloor T \cdot \mu_{app} \rfloor} e^{-\lambda_{st,app}^{inst}(r/\mu_{app} - T)} \frac{(r/\mu_{app} - T)^r}{r!} (\lambda_{st,app}^{inst})^r \tag{5}$$

Equation (5) makes Constraint (2) non-linear. To make the problem tractable for linear solvers such as Gurobi [40], our implementation 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

$$\mathbb{P}(l_{lug,st} + w_{st,app} + \frac{1}{\mu_{app}} \leq eel_{app}^{max}) \Big|_{\lambda_{inst,st} = \Lambda_{st,lug}} = 0.99 \tag{6}$$

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 [27]. Occamy calculates $\Lambda_{st,lug}$ for all possible pairs of sites and users with SciPy's Brent solver [41]. Constraint (2) can then be replaced by the simpler constraint

$$z_{st,lug} \cdot \lambda_{inst,st} \leq \Lambda_{st,lug} \text{ for all } st \in ST, lug \in LUG \tag{7}$$

where the binary variable $z_{st,lug} = 1$ if $lugsite_{lug} = st$, 0 otherwise. And each user group lug is assigned to exactly one site st:

$$\sum_{st \in ST} z_{st,lug} = 1 \quad \forall lug \in LUG \tag{8}$$

3) Optimization objective: Let y_{app}^d be a binary decision variable such that $y_{app}^d = 1$ if application app is deployed at degradation level $d \leq D_{app}$, and $y_{app}^d = 0$ otherwise. We enforce Constraint (9) to ensure that each application is deployed with only one model variant:

$$\sum_{d=1}^{D_{app}} y_{app}^d = 1, \quad \forall app \in APP. \tag{9}$$

We formulate the scheduling objective as a hierarchical multi-objective optimization. The primary objective is to preserve application quality according to a strict priority hierarchy, while the secondary objective minimizes the overall operational deployment cost. Let $APP_p \subseteq APP$ denote the set of applications belonging to priority class $p \in \{1, \ldots, K\}$,

where $p = 1$ represents the highest priority and $p = K$ the lowest. We define the aggregate degradation of priority class p as:

$$\Phi_p = \sum_{app \in APP_p} \sum_{d=1}^{D_{app}} d \cdot y_{app}^d. \tag{10}$$

The overall optimization problem is then formulated as the following lexicographic program:

$$\begin{aligned} \min_{\text{lex}} \quad & \big(\Phi_1, \Phi_2, \ldots, \Phi_K, \mathcal{C}\big) \\ \text{s.t.} \quad & \mathcal{C} = \sum_{st \in ST} \sum_{app \in APP} c_{st} \cdot ninsts_{st,app} \cdot \sigma_{app}^d. \\ & (1),\ (7),\ (8),\ (9). \end{aligned} \tag{11}$$

Solving (11) yields an optimal placement plan, which jointly determines (i) the selected model variant (degradation level) for each application via y_{app}^d, (ii) the deployment location of each application instance across edge sites via $ninsts_{st,app}$ and (iii) the mapping of user groups to edge sites via $lugsite_{lug}$.

The lexicographic formulation (11) enforces a strict priority hierarchy in the degradation process. Specifically, degradation is minimized first for applications in the highest-priority class APP_1. Only among solutions that achieve the minimum feasible degradation for APP_1 do we consider minimizing degradation in APP_2, and so forth. Consequently, applications in priority class p are degraded only if the problem remains infeasible even after all applications in lower-priority classes $APP_{p+1}, \ldots, APP_K$ have been fully degraded.

C gives the overall cost of a solution, assuming that each edge site $st \in ST$ incurs a cost c_{st} per allocated computational resource slot. The global optimization objective is therefore to minimize the total resource cost fee. In the following, We refer to this objective as "min_cost". In the experiments in Section V, we also consider an objective "min_slots" that only aims to minimize the total number of slots used, i.e., $\sum_{st \in ST} \sum_{app \in APP} ninsts_{st,app} \cdot \sigma_{app}^d$.

4) Problem complexity: By relaxing the SLO constraints, the formulated global scheduling problem can be mapped to a multi-dimensional multiple-choice knapsack problem (MMKP), which is NP-hard [42]. In this formulation, each application corresponds to an item with a choice among alternative model variants, each characterized by a resource requirement vector, while edge sites correspond to bins with limited capacity. Introducing tail-latency SLOs and priority-aware degradation further increases the problem complexity, making it challenging to be solved optimally.

IV. Scheduling with Occamy

Solving the problem defined in Section III optimally is computationally intractable at scale due to its NP-hard nature and the combinatorial interaction between application placement with priority, SLO, resource constraints, and quality degradation levels. We propose a scalable solution approach that combines controlled application degradation with heuristic placement optimization.

Algorithm 1 Priority-Based Controlled Degradation

$\mathcal{S}_{\text{deg}} \leftarrow \emptyset$ ▷ Optimal solution with degradation
$\mathcal{F} \leftarrow \emptyset$ ▷ Set of fully degraded applications
$\mathcal{P} \leftarrow$ priority levels sorted in ascending order (lowest $\rightarrow$ highest)
$p \leftarrow 1$ ▷ Index of current priority level
$\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[p]\}$ ▷ Applications at current priority level
$d \leftarrow 1 \quad \forall app \in APP$ ▷ Current degradation stage of each application
while $\mathcal{A} \neq APP$ **do**
  for $app \in \mathcal{A}$ **do**
    $D_{app} \leftarrow$ maximum degradation level of app
  $D_{\max} \leftarrow \max_{app \in \mathcal{A}} D_{app}$
  for $deg \leftarrow 1$ to $D_{\max}$ **do**
    for $app \in \mathcal{A}$ **do**
      if $d < D_{app}$ **then**
        $\sigma_{app} \leftarrow \sigma_{app(deg)},\ \mu_{app} \leftarrow \mu_{app(deg)},\ d \leftarrow deg$
      else
        $\sigma_{app} \leftarrow \sigma_{app(D_{app})},\ \mu_{app} \leftarrow \mu_{app(D_{app})},\ \mathcal{F} \leftarrow \mathcal{F} \cup \{app\}$
    $\mathcal{S}_{\text{deg}} \leftarrow H_{\text{opt}}(APP)$ ▷ Find solution with current degradations
    if feasible solution exists **then**
      return $\mathcal{S}_{\text{deg}}$
  if $(\forall app \in \mathcal{A}, app \in \mathcal{F})$ **AND** $(p < |\mathcal{P}|)$ **then**
    $p \leftarrow p + 1$
    $\mathcal{A} \leftarrow \{app \in APP \mid prio_{app} = \mathcal{P}[p]\}$

Our approach, Occamy, builds around a priority-based degradation mechanism that progressively adapts application configurations when resources are insufficient. At each degradation step, a heuristic placement solver is invoked to determine whether a feasible deployment exists under the current configuration. This design allows the system to preserve application feasibility and SLO compliance while minimizing quality loss, especially for high-priority applications.

Note that some of the model parameters, namely the request rates $\lambda_{lug,app}$, the average network latencies $\bar{l}_{lug,st}$, and the service rates μ_{app}, have to be determined before scheduling with Occamy can take place. We consider the techniques used to establish these parameters to be outside the scope of this paper, as they have been discussed at length in the existing literature, such as direct and indirect techniques for estimating network latencies [43] and arrival and service rates [32].

A. Priority-based controlled degradation

Algorithm 1 constitutes the main solution procedure of Occamy. It aims to restore feasibility when resource constraints prevent deploying all applications at their highest inference quality levels. The algorithm operates by progressively degrading applications through DNN model switching. Degradation is applied stepwise, starting with the lowest-priority applications and proceeding to higher-priority ones only when necessary, following the priority ordering defined by the infrastructure provider to reflect the relative criticality of their applications. At each step, the algorithm invokes H_{opt}, the heuristic placement solver (see Section IV-B), to verify whether a feasible deployment exists under the current configuration.

In particular, Algorithm 1 groups the applications by priority levels and processes them in ascending order, from lowest to highest priority. Initially, all applications operate with

their highest inference quality model variant (i.e., for each application app, $d = 1$, line 6). For a given priority level, the algorithm increases the degradation level of applications at that priority by switching them to next lower-cost DNN variants (i.e., for each application app, $d \leftarrow d+1, d \leq D_{app}$). Each degradation step updates the corresponding service rates and resource slot demands to reflect the performance characteristics of the selected model variant (line 11–14).

After adjusting the degradation levels, the algorithm invokes H_{opt} to determine the best scheduling solution under the current configuration (line 17). H_{opt} minimizes the resource usage cost (Equation 11) while satisfying both the end-to-end SLO and resource capacity constraints. If a solution exists, it is returned immediately as $\mathcal{S}_{\text{deg}}$ (lines 18–19). If an application reaches its maximum degradation level (i.e., $d = D_{app}$), it can no longer be degraded and is added to the set $\mathcal{F}$ of fully degraded applications (lines 15–16). If all applications at the current priority level reach their maximum degradation level and no feasible solution is found, the algorithm proceeds to the next higher priority level and repeats the process (lines 20–22) while keeping the lower-priority applications degraded. This ensures that lower-priority applications are degraded first, while higher-priority ones are degraded only when necessary.

By construction, a higher-priority application is degraded only after every application at each lower-priority level has been fully degraded and no feasible deployment exists. Consequently, resources exhausted during degradation of lower-priority applications cannot cause inadvertent degradation of higher-priority ones outside of this strict ordering.

B. Heuristic solver

The H_{opt} solver invoked in line 20 of Algorithm 1 computes, for a given degradation configuration, a placement plan minimizing cost $\mathcal{C}$ as defined in Equation (11). We denote an algorithm that computes the optimal solution to this problem as G_{opt}. Unfortunately, for larger-scale scenarios with numerous sites, applications, and user groups, the exponential growth in the computational time of G_{opt} is prohibitive.

One way to reduce computation time is to use a greedy algorithm instead: 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. Therefore, we design a heuristic algorithm H_{opt} (Algorithm 2) to solve the scheduling problem. Instead of treating applications in a certain order, H_{opt} proceeds as follows. Inspired by *Monte Carlo sampling* [44], we introduce a sampling parameter $p \in [0, 1]$ that specifies the percentage of permutations of the order of applications to explore (line 4–5). For each permutation π, H_{opt} schedules the applications one by one using the greedy algorithm outlined above (line 9) and runs the optimal solver G_{opt} on the sub-problem with one application (line 12), in this way finding a locally optimal application placement and user mapping solution $\mathcal{S}'$. Occamy specifically uses the Gurobi Optimizer [40] for G_{opt}. If the cost $\mathcal{C}$ of $\mathcal{S}'$ is smaller than the best one found so far (line 16), the result of the algorithm is updated (lines 17–18).

Algorithm 2 Heuristic Solver H_{opt}

```
1: p ∈ (0, 1]                                   ▷ percentage of permutations to explore
2: S ← ∅                                        ▷ optimal solution among all permutations
3: c ← ∞                                        ▷ cost of optimal solution
4: Π ← permutations of order(APP)
5: Π_sampled ← select ⌊p × |Π|⌋ rand. permutations from Π
6: for π ∈ Π_sampled do
7:     available_resources ← nslots_ST
8:     S' ← ∅                                   ▷ solution for permutation π
9:     for app in π do
10:        Run G_opt(app, available_resources)
11:        if scheduling is feasible then
12:            Update S' with solution found by G_opt for app
13:            Update available_resources
14:        else
15:            Break and try next permutation
16:    if cost C of S' < c then
17:        S ← S'
18:        c ← C for S'
19: return S
```

V. Evaluation

We conduct an extensive experimental evaluation of Occamy to answer the following research questions (RQs):

- **RQ1:** How accurate are Occamy's latency predictions and queueing model in realistic Kubernetes-based deployment settings?
- **RQ2:** How well does Occamy's controlled degradation mechanism handle resource scarcity?
- **RQ3:** How does Occamy scale to complex scenarios with multiple sites, applications, and LUGs, including highly skewed workload distributions?
- **RQ4:** What is the impact of different global optimization objectives on system behavior?
- **RQ5:** How closely does the heuristic algorithm approximate the optimal solution?

We support reproducibility and provide our code, the experimental setup, and instructions to reproduce our work [45].

A. General experimental setup

Testbed. All experiments are conducted on a physical testbed without relying on simulations. We use 50 cluster nodes in total, each equipped with an 18-core Intel Xeon CPU and 96 GB of RAM. We deploy Kubernetes and logically partition the nodes into edge sites and LUGs using *labels*. The Kubernetes control plane runs on a dedicated node.

LUGs and load injection. We emulate WAN latencies between LUGs and sites using a network topology generator based on Euclidean network coordinates [43], enforced with *tc*. Workloads are generated using *httperf* [46], which issues requests following a Poisson arrival process, modeling the behavior of independently acting users.

Resource allocation. We allocate resources at the granularity of *slots*, each providing 1 CPU core and 1 GB of RAM. Each application instance is deployed as a Kubernetes pod assigned

Applications	Priority	Variants	Slots per instance	Service rate (req/s)
OCR	High	-	2	10
ResNet	Medium	ResNet-large	10	13
		ResNet-medium	8	21
YOLO	Low	YOLO-large	16	6
		YOLO-medium	14	10.5
		YOLO-small	12	21.5
		YOLO-mini	10	36

TABLE III: AI applications and their resource requirements.

a fixed, integer number of slots. We configure Kubernetes to use the static CPU management policy [47], guaranteeing exclusive core access per pod and eliminating performance interference from CPU sharing.

Applications. We select three representative AI applications with heterogeneous resource requirements, two of which are degradable through multiple model variants: (**a**) *OCR* (Tesseract [48], LSTM-based), (**b**) *ResNet* [38] (image classification), and (**c**) *YOLO* [37] (object detection). Model variants are generated using a data-driven profiling approach inspired by EfficientNet [7]. For each degradable AI application, we first identify a baseline DNN architecture via grid search, and then uniformly scale its dimensions (e.g., number of layers and number of channels per layer) to produce multiple variants. Each variant is trained and evaluated on the coco dataset [49]. We retain only the top-performing variants in terms of inference quality. Table III summarizes their characteristics.

We assign application priorities as follows. *OCR* represents a lightweight but mission-critical service (e.g., AR pipelines in medical or industrial contexts) where degradation poses safety risks; it is assigned the highest priority. *ResNet* and *YOLO* are more computationally demanding but less critical, and are assigned medium and low priorities, respectively.

All results use H_{opt} with a sampling ratio of 100%. We evaluate the impact of different sampling ratios in Section V-F. Each experiment is repeated three times for 10 minutes; the first 15% of samples are discarded as warm-up and averages are reported across runs.

Baselines. We compare Occamy against four baselines. (**a**) INVAR [14] generates placement plans for a single application based on mean end-to-end latency SLOs. (**b**) Kubernetes' HPA [15] triggers reactive scale-out when average CPU utilization exceeds a threshold, configured at 60% (vs. the default 80% [25]) to better align with latency-sensitive workloads; it relies solely on system-level metrics without an explicit performance model. (**c**) MVSP [16], introduced in Section II, places model variants randomly and assigns users via a sequential strategy that considers latency and load; it scales or deploys new variants until a feasible solution is found. (**d**) 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 further edges' resource if resources at the closest edge site are insufficient, mimicking the scheduling without global decision making.

We use INVAR in Section V-B to compare prediction accuracy, and both INVAR and HPA in Section V-C for single-application scaling evaluation. Since INVAR does not support multi-application scheduling and HPA lacks explicit performance modeling, we use MVSP and the *local-first heuristic* in Section V-E for placement plan comparison.

B. Baseline applications performance prediction

Our first experiment addresses RQ1 by assessing the accuracy of Occamy's latency predictions and comparing them against those produced by INVAR. Specifically, we evaluate whether the queueing model employed by Occamy can accurately predict the tail latency of requests served by an application instance. To this end, we deploy a single instance of each target application and its variant(s), and generate requests at controlled rates following a Poisson arrival process. We consider a local deployment setting with zero network latency in order to isolate and focus on queuing delays.

Figure 2 shows the measured mean and P99 end-to-end latencies (EEL) for the three applications and their variants under increasing request rates. We note that INVAR only predicts mean latency and does not provide tail latency estimates; moreover, its predictions exhibit substantial inaccuracies, with an average relative error of 49%. In contrast, Occamy accurately predicts tail latency at low to moderate request rates, and only slightly overestimates it as the load approaches the saturation point of the applications. This discrepancy stems from the assumption of exponentially distributed service times in INVAR's model, as well as from the mismatch between its single-queue multi-server abstraction and the actual request dispatching behavior of Kubernetes, a factor that is not captured in INVAR's simulation-based evaluation.

C. Single-site scenario

Our second experiment further investigates RQ1 by evaluating whether Occamy's latency predictions enable accurate scale-out decisions to satisfy a target P99 tail latency. We focus on the *OCR* application for clarity, as results for the other applications, omitted due to space constraints, exhibit similar trends. We set $eel^{max} = 250$ ms and allow Occamy to determine the number of instances required under increasing request rates. We compare these results against INVAR and HPA. Since INVAR only supports objectives expressed in terms of mean latency, we set its target to $eel^{max}/2$, i.e., 125 ms, to obtain a fair comparison. As in the previous experiment, network latency between the single LUG and the single site is set to zero.

Figure 3 reports the number of instances deployed by Occamy, INVAR, and HPA on the left, the measured and predicted mean and P99 end-to-end latencies respectively by INVAR and Occamy on the right. Occamy accurately determines the number of instances required to satisfy the latency SLO, with predictions that closely match the observed measurements. In contrast, INVAR consistently provisions an insufficient number of instances, even when the mean end-to-end latency remains well below the 250 ms target, resulting in

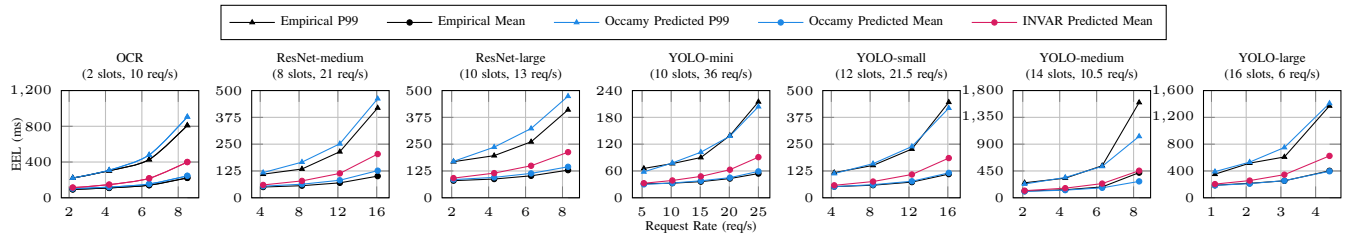


Fig. 2: Performance and prediction accuracy for single instance of the applications and their variants (Table III).

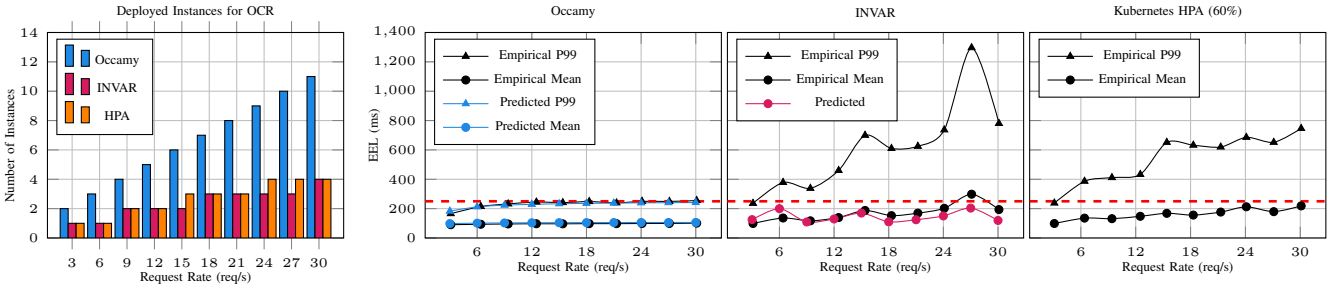


Fig. 3: Empirical and predicted end-to-end latencies (EEL) for the single site scenario using Occamy, INVAR, and Kubernetes HPA to control the number of instances deployed for the *OCR* application.

Series	App	Priority	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
§1	OCR	High	1	2	3	4	5	6	7	8	9	10
	YOLOs	Low	2	2	2	2	2	2	2	2	2	2
§2	OCR	High	2	2	2	2	2	2	2	2	2	2
	YOLOs	Low	2	4	6	8	10	12	14	16	18	20

TABLE IV: Two series of request rates (req/s) for each of the two applications in ten different configurations.

tail latency violations of up to 418%. Similarly, HPA converges to an insufficient scale-out level, leading to tail latency violations of up to 198%. We therefore answer RQ1 affirmatively: Occamy provides accurate latency predictions and enables effective scale-out decisions that satisfy tail latency SLOs for given request rates at a single site, whereas neither INVAR nor HPA can offer this guarantee.

D. Single-site with controlled degradation

Our third experiment addresses RQ2 by assessing the effectiveness of Occamy 's controlled degradation mechanism under resource scarcity and with multiple applications.

We use two applications at two priority levels: *OCR* at high priority and *YOLO* variants at low priority. Each application is requested by a single LUG. Similarly to the previous experiment, network latency is negligible between the LUGs and the single site. We set $eel^{max} = 250$ ms for *OCR* and $eel^{max} = 600$ ms for *YOLO*. We let Occamy decide on the number of instances for each application. If the total resource demand exceeds the site's capacity, Occamy degrades the lower-priority application *YOLO* to its variants to free resources for *OCR* while still satisfying their respective SLOs. As shown in Table IV, we start from a non-degraded scenario with request rates defined by configuration series §1 and §2, and then progressively increase the request rate to *OCR* in §1 and to *YOLO* in §2, while keeping the rate to the other service unchanged. This increase causes total resource demand to exceed the site's 20-slot capacity.

Figure 4 shows the number of slots allocated to each application by Occamy under each configuration series, and the empirical and predicted P99 latencies for requests to each application accordingly. In series §1, all slots are in use from #3. With increasing request rate to *OCR*, Occamy first degrades *YOLO* (#5) to *YOLO-medium* and then to *YOLO-small* (#7), and eventually to *YOLO-mini* (#9) to free resources for *OCR*. While in series §2, with increasing request rate to *YOLO*, the rest of the slots are not enough to allocate one replica of *YOLO-large* from #1, thus Occamy directly degrades *YOLO* to *YOLO-medium* (#2) and similarly, to *YOLO-small* (#4) and *YOLO-mini* (#6). In both series, Occamy successfully guarantees the P99 latency SLO for *OCR* at high priority, while controlling the degradation of *YOLO* by switching to lighter variants at low priority when resources are insufficient. We therefore answer RQ2 affirmatively: the controlled degradation mechanism effectively prioritizes higher-priority applications under resource scarcity while degrading lower-priority applications in a controlled manner.

E. Multi-site, multi-application scenario

We now address RQ3 and RQ4 by evaluating Occamy in a complex scenario involving multiple edge sites and multiple applications.

Infrastructure. We consider five edge sites (E1-E5), each with a capacity of 72 slots. The per-slot pricing is configured to vary across sites as shown in Table V, with E4 and E5 being the most expensive.

We consider ten user locations with diverse latency profiles to the five edge sites, ensuring each location has one nearby edge site and distinct network latencies relative to the others. The network latencies between the locations and the edge sites are sampled from the TCP interactions in a large-scale 5G

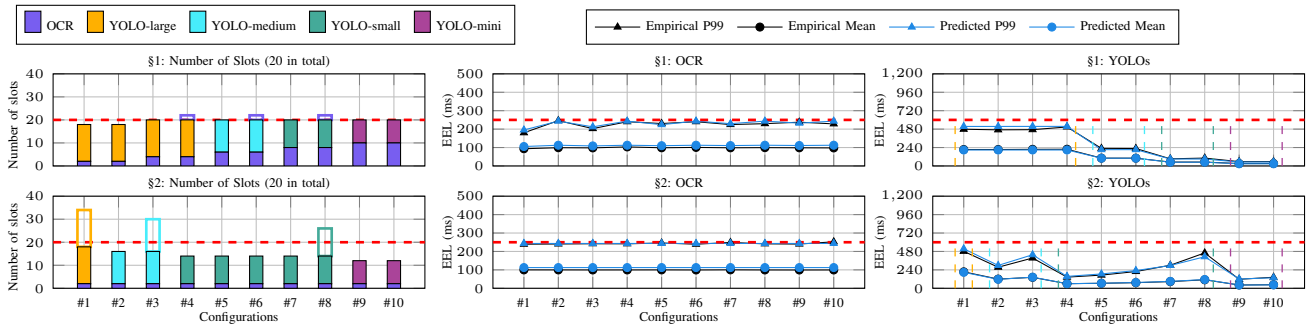


Fig. 4: The number of instances deployed for *OCR* and *YOLO* variants (left) for the single site scenario with graceful degradation using Occamy, along with empirical and predicted P99 and mean end-to-end latencies (EEL) (middle & right). Plots on the top and bottom correspond to the configuration series §1 and §2 in Tables IV.

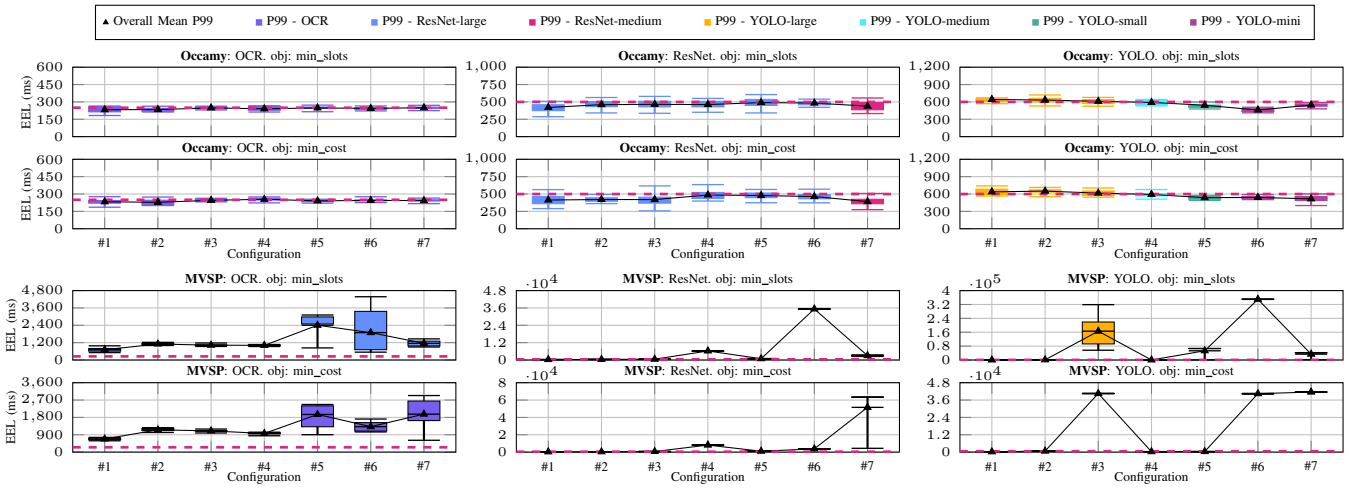


Fig. 5: P99 latency distributions for all LUGs from Occamy (upper) and MVSP (lower) across all seven configurations, for the three target applications and two global optimization objectives.

	Slots	Price	Network latency from LUG to site (ms)									
			LUGs 1-3	LUGs 4-6	LUGs 7-9	LUGs 10-12	LUGs 13-15	LUGs 16-18	LUGs 19-21	LUGs 22-24	LUGs 25-27	LUGs 28-30
E1	72	$2	67	61	40	65	30	40	58	67	80	47
E2	72	$3	117	11	80	115	44	90	58	69	80	27
E3	72	$3	67	61	90	77	80	60	8	19	30	47
E4	72	$4	67	61	30	65	20	40	108	117	130	77
E5	72	$4	17	111	40	27	56	10	58	67	80	97

TABLE V: Resource in slots, pricing, and latencies for sites.

	#1	#2	#3	#4	#5	#6	#7
OCR	13	24	33	40	71	104	**169**
ResNet	13	27	33	42	72	104	127
YOLO	13	26	32	43	72	105	129

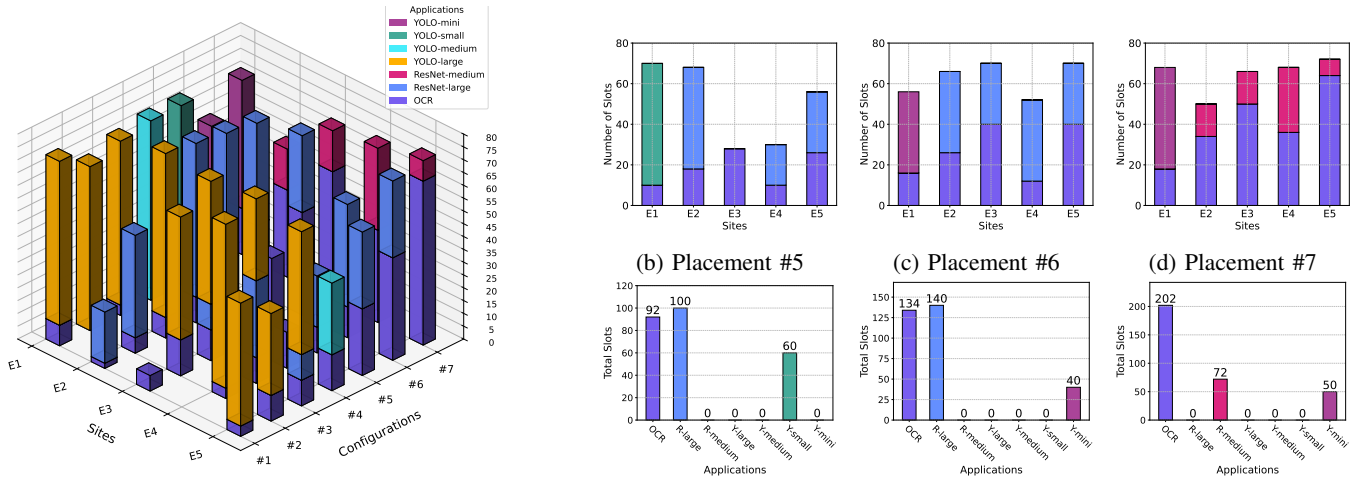
TABLE VI: Total request rates for each of the three applications in the seven different configurations. From #6 to #7 there is a burst in the popularity of the *OCR* application.

network [13], where flows are aggregated into tracking area codes (TACs) to group cells for efficient mobility management (representing LUG) and at the level of remote autonomous systems (ASes) (representing ST). From this trace, we extract latency and session statistics to characterize the interactions between $lug-st$ pairs and generate plausible network latencies and request loads. At each location, we place three lug (one per application), for a total of 30 lug.

SLOs. We set P99 SLOs as $eel^{max} = 250$ ms for *OCR*, $eel^{max} = 500$ ms for *ResNet* and its variant, and $eel^{max} = 600$ ms for *YOLO* and its variants.

Load configurations. We increase the load through seven configurations #1 to #7, with the *total* request rate for each application shown in Table VI. This total rate is distributed among the LUGs according to a normal distribution following the session statistics observed in the interactions between $lug - st$ pairs from trace [13]. In configuration #7, a sharp increase of 30 req/s from LUG1 for *OCR* emulates a sudden surge in user activity, such as a local gathering. This configuration aims to evaluate Occamy's performance under highly skewed demand conditions, where a single user group generates a disproportionate volume of requests.

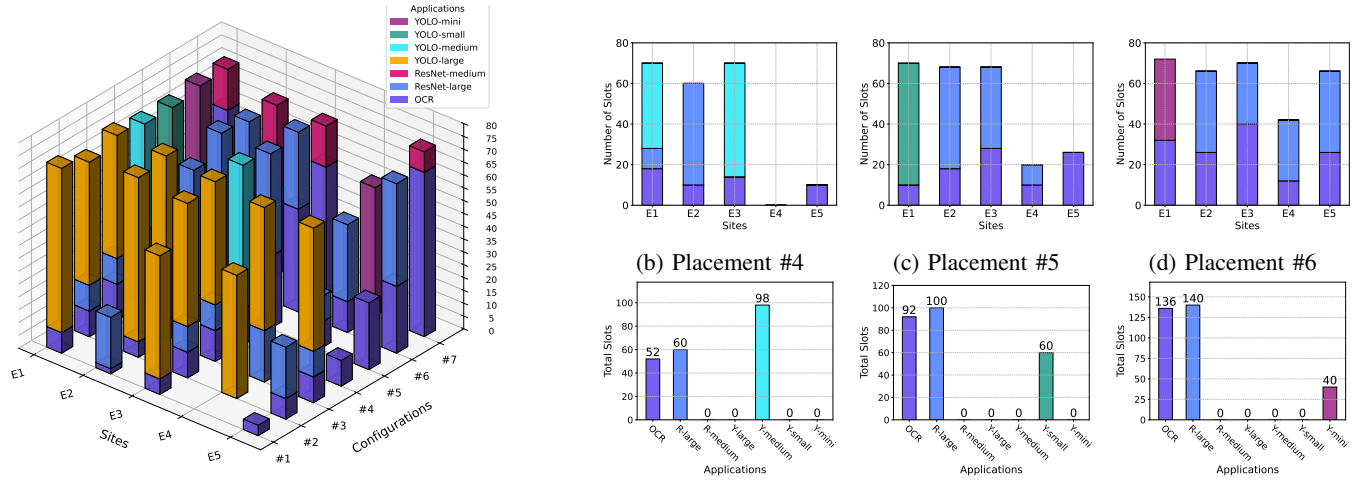
Optimization objectives. We consider two different global optimization objectives "min_slots" (minimizing the total number of slots used) and "min_cost" (minimizing deployment



(a) Obj: min_slots (b) Placement #5 (c) Placement #6 (d) Placement #7

(e) Slots #5 (f) Slots #6 (g) Slots #7

Fig. 6: Placement plans generated by Occamy for multi-site multi-application scenarios under the objective min_slots. Plot (a) shows the overall placement plan for all configurations. Plots (b)–(d) detail the placement plans for configuration #5 to #7 (sliced from the 3D bar chart on the left), while plots (e)–(g) show the corresponding resource slot allocation per application.



(a) Obj: min_cost (b) Placement #4 (c) Placement #5 (d) Placement #6

(e) Slots #4 (f) Slots #5 (g) Slots #6

Fig. 7: Placement plans generated by Occamy for multi-site multi-application scenarios under the objective min_cost. Same layout as Figure 6, but with configurations #4 to #6 to better illustrate the degradation among different steps.

costs by taking per-slot prices into account), as defined in Section III-C3.

Results. Figure 5 shows the distribution of empirical P99 latencies from the placement plans generated by Occamy and MVSP for all configurations and both objectives. For placement plans generated by Occamy, no degradation is required because the aggregate resource demand of all applications can be met by the available resources in configurations #1 and #2. When resource demand increases further, and all sites become fully loaded, Occamy begins to introduce controlled degradation starting from the low-priority level, where *YOLO* is placed, to free resources for higher-priority applications, as observed in configurations #3 to #6. As the resource demand continues to increase and *YOLO* is fully degraded to its *YOLO-mini* variant, Occamy starts degrading the medium-priority level, where *ResNet* is located, from configurations #6 to #7, respectively, to free additional resources for the highest-priority application, *OCR*. This result extends our positive answer to RQ2 to the multi-application scenario.

The placement plans computed by Occamy successfully keep the mean observed P99 latency below the SLO for all applications across almost all configurations, under both optimization objectives. We observe occasional minor SLO violations in certain configurations (up to 18%). For the placement plans generated by MVSP, we observe significant SLO violations across all applications in most configurations and under both objectives (over 500x the SLO), due to their placement plan generation based on latency computation without taking into account the queueing delay. Overall, the mean P99 latencies achieved by configurations selected by Occamy meet the specified requirements. This includes configuration #7, demonstrating that Occamy can effectively generate suitable

Config	Objective	Used slots	Total cost
#5	min_slots	**252**	772
	min_cost	**252**	**732**
	local-first	264	930
#6	min_slots	**314**	1008
	min_cost	316	**984**
	local-first	324	1060
#7	min_slots	**324**	1044
	min_cost	326	**1028**
	local-first	Infeasible	Infeasible

TABLE VII: Global results obtained from Occamy with two optimization objectives and *local-first heuristic* in configuration #5 to #7 as examples. Infeasible indicates that no feasible solution found under the same degradation level.

placement plans across a wide range of scenarios, including demand bursts from specific LUGs, thereby allowing us to answer RQ3 positively.

Under objective min_slots in Figure 6, Occamy minimizes the total number of resource slots used regardless of cost. While under objective min_cost in Figure 7, it avoids the most expensive sites E4 and E5 where possible, but prefers E5 over E2 in configuration #7 because its lower network latency for certain LUGs reduces the number of required resource slots, offsetting the higher per-slot cost.

Table VII contrasts the two objectives with the *local-first heuristic*, which is consistently dominated by Occamy: it uses more resource slots and incurs a higher total cost under configuration #5 and #6, and in configuration #7 the heuristic fails to find any feasible placement within the same degradation level, while both Occamy objectives still return SLO-compliant solutions. This confirms that latency-greedy placement, though intuitive, wastes resources and can even become infeasible under resource scarcity, motivating the need for global scheduling as in Occamy.

This evaluation shows that varying the global optimization objective enables Occamy to produce different SLO-compliant solutions, providing a positive answer to RQ4.

F. Heuristic algorithm evaluation

To answer RQ5, we evaluate the impact of the heuristic algorithm H_{opt} with different sampling ratios compared to the global algorithm G_{opt}. We use 5 sites and 10 user locations as in the previous experiment, except for the number of applications, which we vary from 1 to 6, yielding 6 independent scheduling problems in total, and both objectives as in the previous experiments. We run the algorithms on an Intel CPU with 32 cores and leverage Gurobi's multithreading capability, using all cores when running G_{opt}. For H_{opt}, we explore the set of permutations in 8 parallel processes and give each Gurobi instance 4 cores.

Figure 8 illustrates the computation times for different numbers of applications under both ample and scarce resource conditions under min_slots objective. For G_{opt}, computation times become prohibitive when scheduling more than three applications under ample resources and with two under scarce

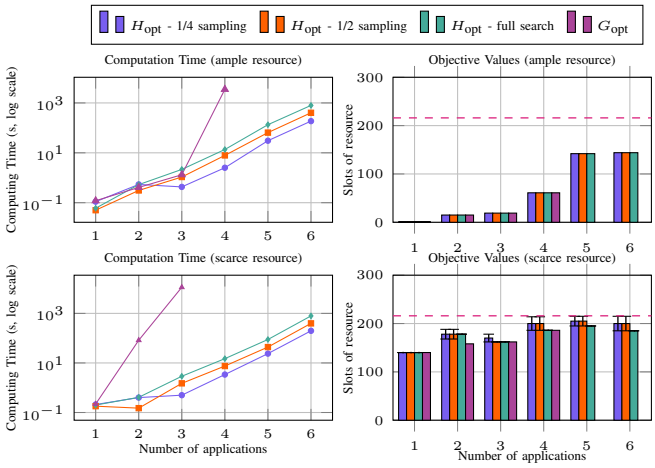


Fig. 8: Computing time (log scale) for H_{opt} and G_{opt} (left) under objective min_slots and results (right) with ample (upper) and scarce resource (bottom).

resources. In contrast, H_{opt} reduces computation time by orders of magnitude. Its runtime scales linearly with the number of samples and remains far below that of G_{opt}. While the sampling ratio has little effect on solution quality under ample resources, it can result in suboptimal solutions under scarce resources, with up to a 16% gap when using a 1/4 sampling ratio, in exchange for the reduced computation time.

The scheduling overhead reported above was not included in the end-to-end latency measurements of Sections V-B and V-E, as Occamy computes placement plans *prior* to experiment execution and holds them fixed. This reflects Occamy's intended operating model: it acts as a *proactive, periodic* scheduler that recomputes deployment plans in response to workload changes (e.g., load shifts between configurations), not as an online, per-request decision maker. Scheduling overhead therefore does not contribute to request latency and is not in conflict with millisecond-scale SLOs. In practice, the seconds-scale overhead of H_{opt} is compatible with periodic reassessment intervals on the order of minutes, which is consistent with the timescales at which workload demand and resource availability change in edge deployments [14].

As a result, using H_{opt} is practical for periodic reassessment of resource and LUG-to-site assignments, even in complex, multi-application scenarios, thus answering RQ5 positively.

VI. Conclusion

We presented Occamy, a scheduler for multiple applications in a distributed edge cluster. Occamy uses queuing models to predict the tail latency of the end-to-end query-response cycle of applications with distributed users. It employs a global scheduling approach that enables meeting each application's tail latency SLO while optimizing overall objectives. Our evaluation demonstrates that Occamy accurately predicts tail latency and effectively allocates resources even in increasingly complex scenarios, outperforming both the closest state-of-the-art approaches and standard elastic scaling mechanisms.

REFERENCES

[1] Hivenet, "Hivenet: Distributed cloud storage & computing," https://www.hivenet.com/, 2026, accessed: 2026-01-06.

[2] Levv.io, "Infrastructure," https://www.levv.io/infrastructure, 2026, accessed: 2026-01-06.

[3] F. Spinelli and V. Mancuso, "Toward enabled industrial verticals in 5G: A survey on MEC-based approaches to provisioning and flexibility," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 1, pp. 596–630, 2021.

[4] 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, pp. 1–12.

[5] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *IEEE/CVF conference on computer vision and pattern recognition*, ser. CVPR, 2018, pp. 2704–2713.

[7] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, ser. ICML. PMLR, 2019, pp. 6105–6114.

[8] J. J. Meza, T. Gowda, A. Eid, T. Ijaware *et al.*, "Defcon: Preventing overload with graceful feature degradation," in *17th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2023, pp. 607–622.

[9] Engineering at Meta. (2025) Journey to 1000 models: Scaling Instagram's recommendation system. [Online]. Available: https://engineering.fb.com/2025/05/21/production-engineering/journey-to-1000-models-scaling-instagrams-recommendation-system/

[10] S. Shen, Y. Feng, M. Xu, Y. Ren *et al.*, "Tango: Harmonious optimization for mixed services in Kubernetes-based edge clouds," *IEEE Transactions on Services Computing*, 2024.

[11] A. A. Amer, I. E. Talkhan, R. Ahmed, and T. Ismail, "An optimized collaborative scheduling algorithm for prioritized tasks with shared resources in mobile-edge and cloud computing systems," *Mobile Networks and Applications*, vol. 27, no. 4, pp. 1444–1460, 2022.

[12] K. Agrawal and S. Abdu Jyothi, "Cooperative graceful degradation in containerized clouds," in *30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2025, pp. 214–232.

[13] P. Ntumba, N.-E.-H. Yellas, S. Bin-Ruba, S. Secci *et al.*, "Data pipeline system designs for in-network learning," in *20th International Conference on Network and Service Management*. IEEE, 2024, pp. 1–9.

[14] B. Wang, D. Irwin, P. Shenoy, and D. Towsley, "Invar: Inversion aware resource provisioning and workload scheduling for edge computing," in *IEEE Conference on Computer Communications*, ser. INFOCOM. IEEE, 2024, pp. 1511–1520.

[15] Kubernetes Documentation, "Kubernetes horizontal pod autoscaling," 2026, accessed: Jan. 23, 2026. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[16] M. Bensalem, J. Dizdarević, and A. Jukan, "DNN placement and inference in edge computing," in *43rd International Convention on Information, Communication and Electronic Technology*, ser. MIPRO, 2020, pp. 479–484.

[17] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Clover: Toward sustainable AI with carbon-aware machine learning inference service," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC. New York, NY, USA: Association for Computing Machinery, 2023.

[18] S. P. Rachuri, N. Shaik, M. Choksi, and A. Gandhi, "Ecoedgeinfer: Dynamically optimizing latency and sustainability for inference on edge devices," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2024, pp. 191–205.

[19] T. Pusztai, C. Marcelino, and S. Nastic, "Hyperdrive: Scheduling serverless functions in the edge-cloud-space 3d continuum," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2024, pp. 265–278.

[20] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *USENIX Annual Technical Conference*, ser. ATC, 2021, pp. 397–411.

[21] M. Salmani, S. Ghafouri, A. Sanaee, K. Razavi, M. Mühlhäuser *et al.*, "Reconciling high accuracy, cost-efficiency, and low latency of inference serving systems," in *3rd Workshop on Machine Learning and Systems*, ser. EuroMLSys, 2023, pp. 78–86.

[22] M. Yuan, L. Zhang, D. Duan, L. Zeng, M.-H. Song *et al.*, "Mitigating tail latency for on-device inference with load-balanced heterogeneous models," *IEEE Transactions on Mobile Computing*, 2025.

[23] S. S. Ogden and T. Guo, "Layercake: Efficient inference serving with cloud and mobile resources," in *IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing*, ser. CCGrid. IEEE, 2023, pp. 191–202.

[24] H. Qiu, S. S. Banerjee, S. Jha *et al.*, "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, pp. 805–825.

[25] J. Liu, Q. Wang, S. Zhang *et al.*, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *24th International Middleware Conference*, 2023, pp. 43–56.

[26] K.-H. Chow, U. Deshpande, V. Deenadayalan *et al.*, "Atlas: Hybrid cloud migration advisor for interactive microservices," in *19th European Conference on Computer Systems*, ser. EuroSys, 2024, pp. 870–887.

[27] Y. Cao, E. Riviere, and R. Sadre, "Lassy: a latency-aware slos-sufficing scheduling system for the cloud/edge continuum," in *25th International Symposium on Cluster, Cloud and Internet Computing*, ser. CCGrid. IEEE, 2025, pp. 276–285.

[28] Y. Han, S. Shen, X. Wang *et al.*, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *IEEE conference on computer communications*, ser. INFOCOM. IEEE, 2021, pp. 1–10.

[29] P. Ntumba, N. Georgantas, and V. Christophides, "Adaptive scheduling of continuous operators for iot edge analytics," *Future Generation Computer Systems*, vol. 158, pp. 277–293, 2024.

[30] Y. Cao, E. Rivière, and R. Sadre, "Scylla: Scheduling multiple latency-sensitive applications in the edge-cloud continuum," in *2026 IEEE/ACM 34rd International Symposium on Quality of Service (IWQoS)*, 2026.

[31] L. Wu, W. Hanafy, T. Abdelzaher, D. Irwin *et al.*, "Faillite: Failure-resilient model serving for resource-constrained edge environments," in *Proceedings of the 2025 ACM Symposium on Cloud Computing*, 2025, pp. 416–429.

[32] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu *et al.*, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *ACM Symposium on Cloud Computing*, ser. SoCC, 2021, pp. 168–181.

[33] 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, pp. 1–7.

[34] S. Lee, Z. Guo, O. Sunercan, J. Ying, T. Kooburat *et al.*, "Shard manager: A generic shard management framework for geo-distributed applications," in *ACM SIGOPS 28th Symp. on Operating Systems Principles*, ser. SOSP, 2021, pp. 553–569.

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

[36] Kubernetes Documentation, "Kubernetes Iptables Proxy Mode," 2026, accessed: Jan. 23, 2026. [Online]. Available: https://kubernetes.io/docs/reference/networking/virtual-ips/

[37] Ultralytics. (2026) YOLOv8. [Online]. Available: https://docs.ultralytics.com/models/yolov8/

[38] PyTorch. (2026) ResNet Models. [Online]. Available: https://docs.pytorch.org/vision/stable/models/resnet.html

[39] A. K. Erlang, "Sandsynlighedsregning og telefonsamtaler," *Nyt tidsskrift for Matematik*, vol. 20, pp. 33–39, 1909.

[40] Gurobi Optimization, LLC. (2026) Gurobi Optimizer. [Online]. Available: https://www.gurobi.com/

[41] SciPy Community. (2026) Brent Optimization Method. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brent.html

[42] H. Kellerer, U. Pferschy, and D. Pisinger, "Introduction to np-completeness of knapsack problems," in *Knapsack problems*. Springer, 2004, pp. 483–493.

[43] 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, pp. 488–503, 2010.

[44] A. Shapiro, "Monte carlo sampling methods," *Handbooks in operations research and management science*, vol. 10, pp. 353–425, 2003.

[45] Occamy github repository. Accessed: Jul. 30, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21705540
[46] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, pp. 31–37, dec 1998.
[47] Kubernetes Documentation, "Kubernetes cpu management policies," 2026, accessed: Jan. 23, 2026. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/
[48] A. Kay, "Tesseract: an open-source optical character recognition engine," *Linux J.*, vol. 2007, no. 159, p. 2, jul 2007.
[49] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.