ClusterLess: Deadline-Aware Serverless Workflow Orchestration on Federated Edge Clusters

Reza Farahani*, Mario Colosi†, Ilir Murturi‡, Stefan Nastic*, Massimo Villari†, Schahram Dustdar*, Radu Prodan§
*Distributed Systems Group (DSG), TU Wien, Vienna, Austria
† MIFT Department, University of Messina, Messina, Italy
‡ Department of Mechatronics, University of Prishtina, Prishtina, Kosova
§ Department of Computer Science, University of Innsbruck, Innsbruck, Austria

arXiv:2605.04310v1 [cs.DC] 5 May 2026

***Abstract*—The recent convergence of edge computing, serverless execution, and Kubernetes (K8s)-based container orchestration has enabled the processing of application workflows close to data sources. While effective within a single-edge cluster, existing schemes do not generalize to federated multi-edge environments, where multiple workflows execute concurrently under strict end-to-end (E2E) deadline constraints. This paper introduces `ClusterLess`, a deadline-aware serverless workflow *orchestration* method for federated multi-edge K8s clusters. `ClusterLess` manages the E2E lifecycle of workflow execution, including dependency analysis, execution-mode selection, and resource-aware placement. To this end, it integrates structured *intra-cluster* orchestration with a leader-selected, *super-master*–driven *inter-cluster coordination* layer, determining where and how each workflow function should be executed across the federated edge clusters. We implement `ClusterLess` using OpenFaaS as the serverless execution substrate and Argo for workflow management, and deploy it on a realistic testbed of *six* edge clusters comprising 64 heterogeneous edge nodes. Experimental results with concurrent serverless workflows, spanning 18 workload configurations across different input sizes and deadline classes, show that `ClusterLess` reduces workflow completion time by up to $40\,\%$, increases deadline satisfaction from below $50\,\%$ to over $90\,\%$, and confines deadline violations to single-digit seconds compared to four baseline methods.**

***Index Terms*—Edge Computing; Serverless Computing; Kubernetes; Workflow; Multi-Cluster Orchestration.**

I. Introduction

Recent industry analyses predict that over $50\,\%$ of critical enterprise applications will run outside centralized public clouds or traditional data centers by 2027, reflecting the accelerating shift toward distributed edge computing [1]. In parallel, serverless computing has become one of the de facto cloud execution models, with production traces reporting billions of function invocations per day on commercial platforms [2], [3]. To support this growth at the edge, Kubernetes (K8s), predominantly employed as the orchestration layer, provides container-based isolation, rapid scaling, and uniform resource management across heterogeneous edge clusters [4], [5]. While effective within a single edge cluster, these mechanisms do not readily generalize to federated multi-edge environments, where application workflows with strict end-to-end (E2E) deadlines must be executed concurrently across clusters with diverse compute capacities and network conditions [6]. In such scenarios, cold starts, resource fragmentation, and inter-cluster communication delays can degrade performance, posing challenges to deadline-aware orchestration [7], [8], [9].

Serverless workflows, typically expressed as directed acyclic graphs (DAGs), require fine-grained, function-level orchestration that respects dependency order, deadline constraints, and interference from concurrent executions. Misplacing an upstream function can propagate delays across the workflow. for example, placing f_1 in a workflow $f_1 \rightarrow f_2 \rightarrow f_3$ on a congested compute instance delays all downstream functions and can violate the E2E deadline despite available downstream resources. Although recent efforts have explored multi-cluster K8s [10], [11], [12], orchestration support for serverless workflows across multiple edge clusters remains limited. Moreover, existing centralized orchestration methods [13] often overlook workflow-level deadlines, dependency structures, and cross-cluster heterogeneity, limiting their effectiveness in real multi-edge environments with dynamic loads and asymmetric computing and networking conditions.

To address these challenges, we introduce `ClusterLess`, a deadline-aware serverless workflow orchestration method for federated multi-edge K8s clusters. `ClusterLess` orchestrates the execution of workflow functions by jointly considering dependency constraints, workflow deadlines, and interference from concurrent workflows. Each cluster runs a local master that performs *intra-cluster orchestration*; for each function invocation, this master selects among four execution modes using native K8s and serverless mechanisms: 1) *warm execution* on edge nodes already hosting an active function instance and offering the lowest completion time; 2) *warm scaling* when existing deployments are saturated but additional replicas remain deadline-feasible; 3) *cold scaling* on suitable edge nodes when warm execution and autoscaling risk violating the workflow deadline; 4) *offloading* when no local execution option can satisfy the workflow deadline. In the offloading mode, control is transferred to *inter-cluster orchestration*, where one cluster master is dynamically elected as a logically central *super-master*. The super-master operates alongside its local master responsibilities, aggregates cluster-level state, and evaluates feasible execution placements across clusters based on communication delay, deployment overhead, queueing state, and execution time, minimizing E2E workflow latency while respecting deadline constraints. It is re-elected

upon failure or overload, preserving orchestration continuity.

To our knowledge, `ClusterLess` is the first orchestration method that jointly incorporates workflow-level deadline guarantees, DAG-aware function scheduling and execution, cross-cluster coordination, and resource heterogeneity into a unified orchestration model for K8s-based edge serverless environments. We implement `ClusterLess` using OpenFaaS as the serverless execution substrate and Argo for workflow management, and deploy it on a real-world multi-cluster edge testbed comprising six K8s clusters. The testbed consists of 64 heterogeneous edge instances, including Jetson-class devices (Nano, Orin Nano, AGX), Raspberry Pis, and x86-based virtual machines. We evaluate `ClusterLess` using concurrent serverless workflows with different deadline tightness and input sizes. Experiments show that `ClusterLess` lowers completion time by up to $40\,\%$ and raises deadline satisfaction from below $50\,\%$ to over $90\,\%$ under heterogeneous workloads.

II. Related Work

Multi-cluster K8s solutions interconnect and coordinate separate clusters, enabling seamless workload placement and migration (e.g., microservices) across heterogeneous infrastructures ranging from the edge to the cloud. Michalke et al. [11] evaluated three multi-cluster connectivity solutions (Submariner, Clusternet, Skupper), demonstrating that inter-cluster communication overhead significantly affects latency and throughput for distributed microservices. Bachar et al. [12] introduced a multi-cluster optimized service-selection system for geo-distributed K8s deployments that employs a centralized broker with Domain Name System (DNS)-based routing to balance cost and latency, ignoring workflow dependencies and deadline constraints. Park et al. [14] introduced a scheduler for K8s-based multi-replica services that profiles each replica's performance and dynamically routes requests to the one with the lowest predicted E2E latency. Early Cloud Native Computing Foundation (CNCF) initiatives such as KubeFed provided foundational support for propagating K8s resources across clusters, but focused primarily on resource replication rather than fine-grained orchestration. More recent systems, such as Karmada [15], enable coarse-grained cross-cluster service discovery and failover, yet remain agnostic to serverless workflow orchestration and deadline guarantees.

State-of-the-art limitations: Existing systems address microservice placement and cross-cluster connectivity, yet lack deadline-aware orchestration for serverless workflows. By ignoring deadlines, function dependencies, and inter-cluster latency, they fall short in resource-limited edge environments.

Several serverless platforms like Knative and Fission leverage K8s primitives for elastic containerized function lifecycle management and resource allocation, but face challenges such as cold-start latency, resource oversubscription, and limited support for workflow-aware or dependency-sensitive orchestration [16]. Lin and Glikson [17] tackled Knative's cold-start bottleneck by introducing a warm pool of pre-provisioned function containers, reducing response latency for sporadic or latency-sensitive invocations. Cvetković et al. [18] introduced a K8s-inspired cluster manager for function scheduling that employs a centralized function scheduler and a lightweight runtime for function invocation throughput, in contrast to traditional layered designs. Simion et al. [19] extended Knative with edge-aware offloading by leveraging latency estimates and location-aware placement, improving throughput for IoT workloads. López et al. [20] proposed Triggerflow, a trigger-based serverless workflow orchestrator for K8s/Knative, but it emphasizes event-driven control-flow extensibility rather than deadline-aware federated multi-edge orchestration. Serenari et al. [21] proposed GreenWhisk, an Apache OpenWhisk-based system evaluated on Raspberry Pi edge clusters, enabling carbon-aware energy-aware function placement. Poggiani et al. [13] proposed live migration of multi-container K8s pods across clusters via container-level checkpointing to preserve state and reduce cold-start overheads, focusing on stateful relocation rather than workflow-level serverless orchestration.

State-of-the-art limitations: Most existing systems improve per-function scheduling, but operate strictly within single clusters and offer no distributed orchestration across multiple K8s clusters. They also overlook functional dependencies, workflow-level deadlines, and cross-cluster computing and bandwidth constraints, limiting their use for concurrent serverless workflows in multi-cluster edge environments.

TABLE I: Related work comparison (SL: Serverless; P: Pod; WF: Workflow; MC: Multi-cluster; AT: Autoscaling; ND: New Deployment; DL: Deadline-aware; Intra: Intra-cluster; Inter: Inter-cluster).

Work	*SL*	*Processing*		*MC*	*AT*	*ND*	*DL*	*Orchestration*		*Evaluation infrastructure*
		P	*WF*					*Intra*	*Inter*	
[11]	✓	✓	×	✓	×	×	×	×	✓	2 k3s clusters, Knative.
[12]	×	✓	×	✓	×	×	×	×	✓	Simulation and 5 K8s clusters.
[13]	✓	✓	×	✓	×	✓	×	×	✓	2 K8s clusters on OpenStack.
[14]	×	✓	×	×	×	✓	×	✓	×	Single K8s cluster (5 nodes).
[17]	✓	✓	×	×	✓	✓	×	✓	×	Single K8s cluster, Knative.
[18]	✓	✓	×	×	✓	✓	×	✓	×	93-node cluster, Knative.
[19]	✓	✓	×	✓	×	✓	×	×	✓	4 RPis, x64 node, VM, Knative.
[20]	✓	✓	✓	×	✓	✓	×	✓	×	K8s cluster (5 nodes) and IBM cloud.
[21]	✓	✓	×	×	✓	✓	×	✓	×	10 RPis and 10 servers, OpenWhisk.
`ClusterLess`	✓	✓	✓	✓	✓	✓	✓	✓	✓	64 Jetsons, RPis, VMs, OpenFaaS, Argo.

III. Problem Formulation

A. Function and workflow model

We consider a set of *serverless workflows* $\mathcal{W}$ executed at different time intervals. Each workflow $w \in \mathcal{W}$ is modeled as a DAG $G_w = (F_w, E_w)$, where F_w denotes the set of workflow functions and $E_w \subseteq F_w \times F_w$ denotes dependency edges, such that $(g, f) \in E_w$ implies that function f can start only after g completes. Each workflow w arrives at time A_w and is subject to a strict end-to-end (E2E) deadline D_w, where delays in individual functions may propagate and result in workflow-level deadline violations. Each function $f \in F_w$ is characterized by: 1) computational demands, given by C^f (CPU-seconds) and M^f (bytes of memory); and 2) input and output data sizes $(X^f_{\text{in}}, X^f_{\text{out}})$ in MB, which determine inter–function communication overhead. Multiple workflows may execute concurrently and contend for compute, memory, and network bandwidth resources across federated edge clusters.

B. Cluster model

We consider a set of N federated K8s edge clusters $\mathcal{K} = \{K_1, \ldots, K_N\}$ deployed at distinct locations. Each clus-

ter $K_n \in \mathcal{K}$ comprises: 1) a *local master* M_n responsible for *intra-cluster* orchestration decisions, including execution-mode selection (warm execution, warm scaling, cold scaling, or offloading); and 2) a set of *worker nodes* $\mathcal{Z}_n = \{z_{n1}, \ldots, z_{n|\mathcal{Z}_n|}\}$ At time t, each worker $z_{ni} \in \mathcal{Z}_n$ provides available CPU and memory capacities $C_{ni}(t)$ and $M_{ni}(t)$, and maintains a local execution queue, inducing a queuing delay $Q_{ni}(t)$ due to concurrent workloads. Let $\Gamma_{ni}(t)$ denote the number of active function instances on worker z_{ni} at t, and let R_{ni} be its maximum concurrency capacity. We define the normalized load of cluster K_n at time t as:

$$\mathrm{L}_n(t) = \frac{1}{|\mathcal{Z}_n|} \sum_{z_{ni} \in \mathcal{Z}_n} \frac{\Gamma_{ni}(t)}{R_{ni}}. \tag{1}$$

which captures the average utilization of worker-level concurrency capacity and enables load comparison across clusters.

While each local master independently performs intra-cluster orchestration, deadline-feasible execution cannot always be guaranteed locally due to resource contention, cold-start overheads, or bursty arrivals. To orchestrate *inter-cluster* across clusters, the system maintains a *logically centralized* coordinator, referred to as the *super-master*, elected among the local masters $\{M_1, \ldots, M_N\}$. We model the super-master as a time-indexed selection function $SM(t_e) \in \mathcal{K} \cup \{\emptyset\}$, where $t_e = e \cdot \Delta T$ denotes a discrete control epoch and $SM(t_e) = \emptyset$ represents a transient state in which no eligible super-master is available (e.g., due to failures or overload). Each local master M_n periodically emits heartbeat messages to signal its availability. Let t_n^{HB} denote the most recent heartbeat received from M_n. Cluster K_n is considered *alive* at epoch t_e if:

$$\mathrm{Alive}_n(t_e) = \begin{cases} 1, & t_e - t_n^{HB} \le T_{\text{fail}}; \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

where T_{fail} is the failure-detection timeout. Thus, only clusters that satisfy Eq. 3 are eligible to act as super-master:

$$\mathrm{Alive}_n(t_e) = 1 \quad \wedge \quad \mathrm{L}_n(t_e) \le l, \tag{3}$$

where l shows the acceptable super-master overhead.

C. Intra-cluster model

When a function $f \in F_w$ becomes *ready* for execution (i.e., all predecessor functions in E_w have completed), the local master M_n of the cluster K_n to which the request is initially submitted performs an *intra-cluster orchestration* decision. For each ready function f, the local master selects an execution mode em^f from the following set:

1) Warm execution: serves f by an already active container on worker z_{ni}, incurring only queuing delay $Q_{ni}(t)$ and execution time $\mathcal{T}_{ni}^f$.

2) Warm scaling: spawns an additional replica of f on z_{ni} using existing images, incurring an autoscaling delay WS_{ni}^f before execution.

3) Cold scaling: deploys f from scratch (image pull, initialization, and container setup) on z_{ni}, incurring a cold-start overhead CS_{ni}^f prior to execution.

4) Offloading: forwards the execution request to the super-master $SM(t)$ for placement on a remote cluster.

If f is assigned locally to worker z_{ni} under execution mode em^f, its *serving time* χ_n^f is given by:

$$\chi_n^f = \begin{cases} Q_{ni}(t) + \mathcal{T}_{ni}^f, & em^f = \text{warm execution}; \\ Q_{ni}(t) + WS_{ni}^f + \mathcal{T}_{ni}^f, & em^f = \text{warm scaling}; \\ Q_{ni}(t) + CS_{ni}^f + \mathcal{T}_{ni}^f, & em^f = \text{cold scaling}. \end{cases} \tag{4}$$

For dependent functions $g \to f \in E_w$, the data-transfer delay θ^{gf} is:

$$\theta_n^{gf} = \begin{cases} \dfrac{X_{\text{out}}^g}{b_{n,i,j}(t)}, & g \text{ and } f \text{ run on } z_{ni} \neq z_{nj} \text{ in } K_n; \\ 0, & g \text{ and } f \text{ run on the same worker } z_{ni}. \end{cases} \tag{5}$$

where $b_{n,i,j}(t)$ is the available intra-cluster bandwidth. We define $\mathcal{S}_n^f$ as the start time of f conditional on executing in K_n defined in Eqs. (6):

$$\mathcal{S}_n^f = \begin{cases} A_w, & \nexists\, g \to f \in E_w; \\ \max\limits_{g \to f \in E_w} \left\{ \mathcal{F}_n^g + \theta_n^{gf} \right\}, & \exists\, g \to f \in E_w. \end{cases} \tag{6}$$

where $\mathcal{F}_n^g$ denotes the completion time of predecessor function g when executed in K_n, computed by Eq. (7).

$$\mathcal{F}_n^g = \mathcal{S}_n^g + \chi_n^g. \tag{7}$$

D. Inter-cluster model

If the local master M_n cannot place a ready function $f \in F_w$ within K_n such that the workflow deadline constraint can still be satisfied, it offloads f to the super-master $SM(t_e)$ at control epoch t_e. For each candidate cluster $K_{n'} \in \mathcal{K} \backslash \{K_n\}$, $SM(t_e)$ calculates the function completion time of f as:

$$\mathcal{F}_{n'}^f = \max\Big(\mathcal{S}_{n'}^f,\, t_e\Big) + \delta_{nn'} + \chi_{n'}^f. \tag{8}$$

where $\mathcal{S}_{n'}^f$ is computed via Eq. (6) based on the remote cluster state, $\delta_{nn'}$ is the inter-cluster transfer delay from cluster K_n to $K_{n'}$ and $\chi_{n'}^f$ is the serving time of f in cluster $K_{n'}$, computed using the same execution modes as in Eq. (4) based on the remote cluster state. The super-master considers the set of feasible target clusters only when the cluster satisfies the deadline-feasibility constraint:

$$\mathcal{K}^f = \Big\{ K_{n'} \in \mathcal{K} \setminus \{K_n\} \;\Big|\; \mathcal{F}_{n'}^f \le A_w + D_w \Big\}. \tag{9}$$

If $\mathcal{K}^f \neq \varnothing$, $SM(t_e)$ selects the destination cluster K_{n^*} as:

$$K_{n^*} = \arg \min_{K_{n'} \in \mathcal{K}^f} \mathcal{F}_{n'}^f. \tag{10}$$

The selection of K_{n^*} constitutes a binding inter-cluster placement decision, and function f is dispatched to cluster K_{n^*} for execution. If $\mathcal{K}^f = \varnothing$, no deadline-feasible inter-cluster placement exists; function f is declared infeasible, and workflow w is marked as deadline-violated. In addition, when multiple offloading requests are pending at $SM(t_e)$, it processes them according to an earliest-deadline-first policy.

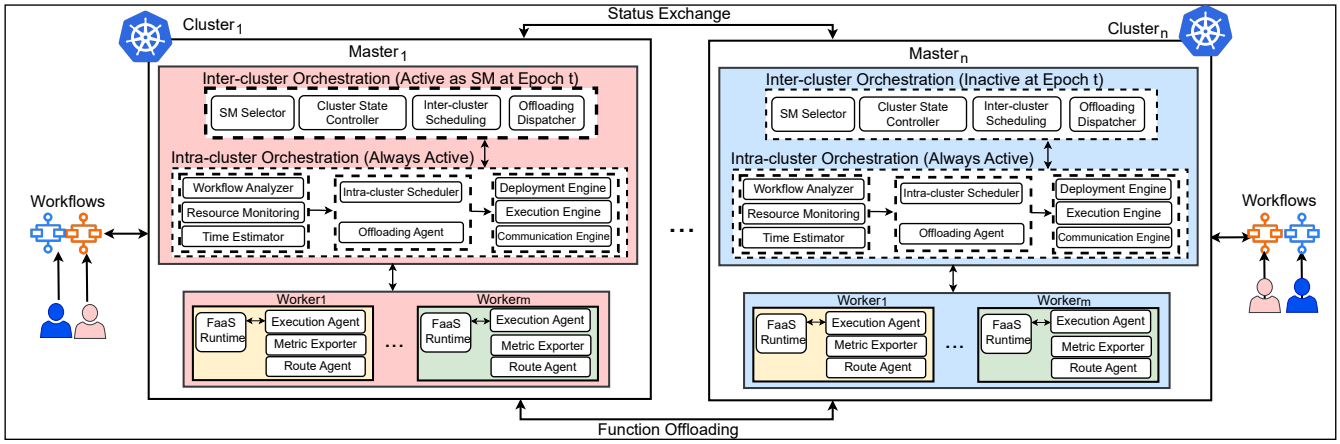


Fig. 1: ClusterLess system architecture.

E. Resource feasibility model

A placement decision for function f on worker z_{ni} at time t is *resource-feasible* only if sufficient compute and memory resources are available on the worker node:

$$C^f \leq C_{ni}(t), \qquad M^f \leq M_{ni}(t), \tag{11}$$

In addition, data transfers induced by workflow dependencies must be *bandwidth-feasible*. For a dependency $g \to f \in E_w$ executed on workers z_{ni} and z_{nj} within the same cluster K_n, the required data transfer is feasible only if:

$$X^g_{\text{out}} \leq b_{n,i,j}(t), \tag{12}$$

where $b_{n,i,j}(t)$ denotes the currently available intra-cluster bandwidth between the two workers at time t. For inter-cluster execution, when g is executed in cluster K_n and f is offloaded to cluster $K_{n'}$, the dependency transfer is feasible only if:

$$X^g_{\text{out}} \leq b_{n,n'}(t), \tag{13}$$

where $b_{n,n'}(t)$ denotes the available inter-cluster bandwidth between clusters K_n and $K_{n'}$.

F. Completion-time model

For each function $f \in F_w$, the orchestration process (intra- or inter-cluster execution) induces a unique execution cluster. Accordingly, the effective completion time of f is:

$$\mathcal{F}^f = \begin{cases} \mathcal{F}^f_n, & f \text{ is executed in } K_n; \\ \mathcal{F}^f_{n^*}, & f \text{ is offloaded and executed in } K_{n^*}. \end{cases} \tag{14}$$

The end-to-end (E2E) completion time of workflow w is defined in Eq. (15) and is considered *deadline-feasible* if it satisfies the workflow deadline.

$$C_w = \max_{f \in F_w} \mathcal{F}^f \leq A_w + D_w. \tag{15}$$

IV. ClusterLess Architecture

Fig. 1 depicts the ClusterLess architecture spanning N federated K8s-based edge clusters. Each cluster consists of a *local master* responsible for control-plane decisions and a set of *worker nodes* that execute serverless functions.

1) Local master nodes: support two logically distinct orchestration paths, corresponding to intra- and inter-cluster decision-making, which jointly enable deadline-aware workflow execution across the federation.

a) Intra-cluster orchestration: is *always active* on every master and realizes local workflow execution decisions within a cluster. Upon workflow submission, the *workflow analyzer* parses the workflow structure and dependency relations, while the *resource monitoring* module continuously tracks worker-level compute, memory, queuing, and bandwidth conditions. Based on this information, the *time estimator* derives execution-time estimates for different execution modes (warm execution, warm scaling, cold scaling). The *intra-cluster scheduler* then determines a feasible worker and execution mode for each ready function. When no such placement exists locally, the *offloading agent* escalates the function to the inter-cluster orchestration path. The selected decisions are enforced through the *deployment*, *execution*, and *communication* engines, which collectively realize the selected placement, execution mode, and dependency-aware data transfers on worker nodes.

b) Inter-cluster orchestration: is deployed on every master but *remains inactive* unless the master is designated as the super-master. When active, it enables orchestration by aggregating cluster-level state and arbitrating offloaded functions. The *SM selector* maintains cluster liveness and load information, while the *cluster state controller* constructs a global view of the federation. Using this information, the *inter-cluster scheduler* evaluates offloaded functions across clusters considering their execution feasibility and deadline urgency, and the *offloading dispatcher* communicates the resulting placement decisions back to the destination masters. Inter-cluster orchestration relies on two logical communication paths: a *status exchange* disseminates load and liveness information among masters, and a *function offloading* transfers execution requests and placement decisions across clusters.

2) Worker nodes: implement the data-plane components required to execute functions delegated by the local master. Each worker hosts a lightweight serverless *FaaS runtime* and

a minimal set of agents, including an *execution agent* for triggering function execution, a *metric exporter* for reporting execution and resource statistics, and a *route agent* for managing dependency-aware data transfers between functions on the appropriate intra- or inter-cluster links.

V. ClusterLess Decision-Marking Algorithms

A. Super-master maintenance

Alg. 1 presents the epoch-based super-master maintenance procedure, ensuring that ClusterLess operates under a responsive, load-aware, and fault-tolerant coordinator for inter-cluster orchestration. The algorithm takes as input the set of clusters $\mathcal{K}$, the control epoch length ΔT, the heartbeat failure timeout T_{fail}, and the admissible coordination-load threshold l, and publishes the super-master $SM(t_e)$ at each epoch t_e. At system startup ($e = 0$), each cluster initializes its heartbeat timestamp and computes its normalized load $\mathrm{L}_n(t_0)$ based on worker concurrency (lines 2–4). The cluster with the minimum load is deterministically selected as the initial super-master (line 5), yielding a lightweight coordinator at bootstrap. At each subsequent epoch $t_e = e \cdot \Delta T$, clusters update their load estimates if a heartbeat is received during $(t_{e-1}, t_e]$; otherwise, the previous value is retained to avoid oscillations under transient reporting delays (lines 6–12).

Cluster liveness is then evaluated using the timeout condition in Eq. (2) (lines 13–14). The current super-master is validated against the eligibility constraint in Eq. (3); it is invalidated if it is unavailable, not alive, or exceeds the admissible load threshold (line 15). If invalid, the algorithm deterministically re-selects the alive and load-eligible cluster with the minimum normalized load (lines 16–17); if no such cluster exists, no super-master is assigned for the epoch (line 19), deferring inter-cluster coordination to subsequent epochs. Otherwise, the previous super-master is retained unchanged (line 21). At the end of each epoch, the resulting $SM(t_e)$, either newly selected, retained, or empty, is published to the inter-cluster orchestration layer (line 22), enabling continuous, stable, and epoch-consistent decisions.

B. Intra-cluster orchestration

Alg. 2 presents the intra-cluster orchestration procedure executed by the local master M_n when a workflow w is submitted to cluster K_n at time t. The algorithm takes as input the workflow DAG G_w, its arrival time A_w, deadline D_w, and the target cluster K_n, and outputs the local orchestration plan $LOrch$, specifying for each function $f \in F_w$ its execution mode em^f, local placement z^f (if any), and the corresponding start and completion times $(\mathcal{S}_n^f, \mathcal{F}_n^f)$. At initialization (line 1), all functions are marked as local ($ext = 0$), assigned the default execution mode *warm execution*, and left unscheduled with infinite completion time. The algorithm then iteratively refines $LOrch$ while the workflow-level completion time exceeds the deadline, i.e., $\textsc{WFinish}(LOrch) > A_w + D_w$ (line 2), where $\textsc{WFinish}()$ implements Eq. (15), ensuring correctness for arbitrary DAG structures, including parallel terminal branches. In each iteration, $\textsc{IdentifyBottleneck}(LOrch)$ is invoked

Algorithm 1: Super-master maintenance.

Input: $\mathcal{K} = \{K_1, \ldots, K_N\}$, ΔT, T_{fail}, l
Output: $SM(t_e)$

1 $e \leftarrow 0 \quad t_0 \leftarrow 0$
2 **forall** $K_n \in \mathcal{K}$ **do**
3 $\quad t_n^{HB} \leftarrow t_0$
4 $\quad \mathrm{L}_n(t_0) \leftarrow \dfrac{1}{|\mathcal{Z}_n|} \sum\limits_{z_{ni} \in \mathcal{Z}_n} \dfrac{\Gamma_{ni}(t_0)}{R_{ni}}$
5 $SM(t_0) \leftarrow \arg\min\limits_{K_n \in \mathcal{K}} \mathrm{L}_n(t_0)$
6 **for** $e \leftarrow 1$ **to** ∞ **do**
7 $\quad t_e \leftarrow e \cdot \Delta T$
8 $\quad$ **forall** $K_n \in \mathcal{K}$ **do**
9 $\quad\quad$ **if** $t_n^{HB} > t_{e-1}$ **then**
10 $\quad\quad\quad \mathrm{L}_n(t_e) \leftarrow \dfrac{1}{|\mathcal{Z}_n|} \sum\limits_{z_{ni} \in \mathcal{Z}_n} \dfrac{\Gamma_{ni}(t_e)}{R_{ni}}$
11 $\quad\quad$ **else**
12 $\quad\quad\quad \mathrm{L}_n(t_e) \leftarrow \mathrm{L}_n(t_{e-1})$
13 $\quad$ **forall** $K_n \in \mathcal{K}$ **do**
14 $\quad\quad \mathrm{Alive}_n(t_e) \leftarrow \mathbb{I}\left[t_e - t_n^{HB} \le T_{\text{fail}}\right]$
15 $\quad$ **if** $SM(t_{e-1}) = \emptyset \ \vee\ (\mathrm{Alive}_{SM(t_{e-1})}(t_e) = 0 \ \vee\ \mathrm{L}_{SM(t_{e-1})}(t_e) > l)$ **then**
16 $\quad\quad$ **if** $\exists\, K_n \in \mathcal{K} :\ \mathrm{Alive}_n(t_e) = 1 \ \wedge\ \mathrm{L}_n(t_e) \le l$ **then**
17 $\quad\quad\quad SM(t_e) \leftarrow \arg\min\limits_{K_n \in \mathcal{K}:\mathrm{Alive}_n(t_e)=1 \wedge \mathrm{L}_n(t_e) \le l} \mathrm{L}_n(t_e)$
18 $\quad\quad$ **else**
19 $\quad\quad\quad SM(t_e) \leftarrow \emptyset$
20 $\quad$ **else**
21 $\quad\quad SM(t_e) \leftarrow SM(t_{e-1})$
22 $\quad \textsc{Publish}(SM(t_e))$

(line 3) to extract three components: a committed partial plan $LOrch_c$, a bottleneck function f^{bn} selected among non-offloaded functions with the largest current completion time, and a pending set P containing all other functions whose execution may be affected by changes to f^{bn}. To ensure progress, functions that have already been evaluated under all execution modes without yielding an improvement are excluded from bottleneck selection in the current refinement cycle; however, a function may be selected again as a bottleneck if updates to predecessor functions reduce its effective completion time.

The algorithm initializes a control flag $success$ (line 5) to track whether an improving execution-mode update for the bottleneck function can be found. While no improvement is achieved and the bottleneck has not been escalated to offloading (lines 6–8), the execution mode of f^{bn} is deterministically advanced using $\textsc{NextMode}()$ following the ordered policy *warm execution→warm scaling→cold scaling→offloading*. For each mode, $\textsc{ApplyMode}()$ evaluates feasibility using the serving-time model in Eq. (4), the dependency constraints in Eqs. (6)–(7), and the resource and bandwidth feasibility conditions in Eqs. (11)–(13). The flag $success$ is set to 1 only if a feasible and improving update is obtained; offloading succeeds only upon acknowledgement from the super-master. If an improvement is found, the updated bottleneck decision is committed to $LOrch$ (lines 9–10). Otherwise, if all execution modes fail to yield a feasible update, the pending set P is restored unchanged (lines 11–13), and the algorithm proceeds to the next iteration without altering prior decisions.

After committing the bottleneck update, the algorithm orchestrates the remaining pending functions (lines 15–20). For

Algorithm 2: Intra-cluster orchestration by M_n at time t.

Input: w with $G_w = (F_w, E_w)$, A_w, D_w, K_n, t
Output: $LOrch$

1 $LOrch \leftarrow \{(w, f, ext = 0, z^f = \emptyset, em^f = \text{warm exec}, \mathcal{S}_n^f = \infty, \mathcal{F}_n^f = \infty) \mid f \in F_w\}$
2 **while** WFINISH($LOrch$) $> A_w + D_w$ **do**
3 $(LOrch_c, f^{bn}, P) \leftarrow$ IDENTIFYBOTTLENECK($LOrch$)
4 $LOrch \leftarrow LOrch_c$
5 $success \leftarrow 0$
6 **while** $(success = 0) \ \wedge \ (em^{f^{bn}} \neq \textit{offloading})$ **do**
7 $em^{f^{bn}} \leftarrow$ NEXTMODE($em^{f^{bn}}$)
8 $(success, fs^{bn}, P) \leftarrow$ APPLYMODE($f^{bn}, em^{f^{bn}}, LOrch, P, K_n, t$)
9 **if** $success = 1$ **then**
10 $LOrch \leftarrow LOrch \cup \{fs^{bn}\}$
11 **else**
12 $LOrch \leftarrow LOrch \cup P$
13 **continue**
14 $LOrch_p \leftarrow \varnothing$
15 **forall** $fs \in P$ **do**
16 $(w, f, ext, z^f, em^f, \mathcal{S}_n^f, \mathcal{F}_n^f) \leftarrow fs$
17 **if** $ext = 1$ **then**
18 **continue**
19 $\mathcal{I}^f_{\text{avail}} \leftarrow$ GETAVAIL(f, em^f, K_n, t)
$fs' \leftarrow$ FUNCORCH($LOrch \cup LOrch_p,\ fs,\ \mathcal{I}^f_{\text{avail}},\ K_n,\ t$)
20 $LOrch_p \leftarrow LOrch_p \cup \{fs'\}$
21 $LOrch \leftarrow LOrch \cup LOrch_p$
22 **return** $LOrch$

each function $fs \in P$ that is not marked external, M_n retrieves the set of available execution instances via GETAVAIL() (line 19), restricting evaluation to runnable pods or scalable targets. The function-level orchestration module FUNCORCH() then selects a resource- and bandwidth-feasible placement and computes $(\mathcal{S}_n^f, \mathcal{F}_n^f)$ according to Eqs. (4)–(7) and Eqs. (11)–(13) (line 20). These decisions are merged into $LOrch$ (lines 20–21), completing one refinement step. The process terminates once the workflow deadline is satisfied, and the final plan $LOrch$ is returned (line 22).

C. Inter-cluster orchestration

Alg. 3 presents the inter-cluster orchestration procedure executed by the super-master $SM(t_e)$ at control epoch t_e for handling the set of offloaded functions $\mathit{Offload}(t_e)$. The algorithm takes as input the set of pending offload requests $\mathit{Offload}(t_e)$ and the federated cluster set $\mathcal{K}$, and outputs the global orchestration plan $GOrch(t_e)$, specifying one destination cluster per offloaded function. At the beginning of the epoch, all offload requests are ordered by earliest-deadline-first (EDF) using the workflow deadline $A_w + D_w$ (line 1), prioritizing functions on critical deadline paths. The global decision set $GOrch(t_e)$ is then initialized to empty (line 2). The super-master evaluates all candidate destination clusters for each offload request in EDF order (lines 3–4) for each candidate, computes the dependency-aware earliest start time $\mathcal{S}_{n'}^f$ based on predecessor completion times and inter-cluster transfer delays (lines 5–8), followed by the corresponding remote completion time $\mathcal{F}_{n'}^f$ at epoch t_e (line 9).

The algorithm then constructs the feasible-improving set $\mathcal{K}^f$ (line 10), containing only those clusters that both satisfy the workflow deadline constraint and strictly improve upon the best local completion time $\mathcal{F}_n^f$. If no such cluster exists, the function is retained at its origin cluster K_n and recorded accordingly in $GOrch(t_e)$ (lines 11–12). Otherwise, the destination cluster that minimizes $\mathcal{F}_{n'}^f$ is selected (lines 14–15). This process is repeated for all offload functions, and $GOrch(t_e)$ is returned at the end of the epoch (line 16).

Algorithm 3: Inter-cluster orchestration by $SM(t_e)$.

Input: t_e, $\mathit{Offload}(t_e) = \{(w, f, K_n, \mathcal{F}_n^f)\}$, $\mathcal{K}$
Output: $GOrch(t_e)$

1 $\mathit{Offload}(t_e) \leftarrow \text{argsort}_{(w,f,K_n,\mathcal{F}_n^f) \in \mathit{Offload}(t_e)}(A_w + D_w)$
2 $GOrch(t_e) \leftarrow \varnothing$
3 **forall** $(w, f, K_n, \mathcal{F}_n^f) \in \mathit{Offload}(t_e)$ **do**
4 **forall** $K_{n'} \in \mathcal{K} \setminus \{K_n\}$ **do**
5 **if** $\nexists\, g \to f \in E_w$ **then**
6 $\mathcal{S}_{n'}^f \leftarrow A_w$
7 **else**
8 $\mathcal{S}_{n'}^f \leftarrow \max_{g \to f \in E_w} \{\mathcal{F}_{n'}^g + \theta_{n'}^{gf}\}$
9 $\mathcal{F}_{n'}^f \leftarrow \max\{\mathcal{S}_{n'}^f, t_e\} + \delta_{nn'} + \chi_{n'}^f$
10 $\mathcal{K}^f \leftarrow \left\{ K_{n'} \in \mathcal{K} \setminus \{K_n\} \;\middle|\; (\mathcal{F}_{n'}^f \leq A_w + D_w) \ \wedge \ (\mathcal{F}_{n'}^f < \mathcal{F}_n^f) \right\}$
11 **if** $\mathcal{K}^f = \varnothing$ **then**
12 $GOrch(t_e) \leftarrow GOrch(t_e) \cup \{(w, f, K_n)\}$
13 **else**
14 $K_{n^*} \leftarrow \arg\min_{K_{n'} \in \mathcal{K}^f} \mathcal{F}_{n'}^f$
15 $GOrch(t_e) \leftarrow GOrch(t_e) \cup \{(w, f, K_{n^*})\}$
16 **return** $GOrch(t_e)$

D. Time complexity

Alg. 1 performs a linear scan over clusters once per epoch, yielding $O(|\mathcal{K}|)$. Alg. 2 updates the local plan by resolving a single bottleneck per iteration across a bounded mode set, giving $O(|F_w|)$ per workflow. Alg. 3 sorts offloaded functions by EDF and evaluates all destination clusters, resulting in $O(|\mathit{Offload}| \log |\mathit{Offload}| + |\mathit{Offload}| \cdot |\mathcal{K}|)$ per epoch.

VI. EVALUATION SETUP

A. Multi-cluster edge testbed

We evaluate `ClusterLess` on a realistic edge testbed comprising six K8s clusters, spanning 50 physical edge devices and 14 edge `KVM`-virtualized machines. All clusters run `Kubernetes 1.32` with `containerd 1.7` and are interconnected via `Submariner` (Globalnet mode), enabling transparent cross-cluster pod addressing, service discovery, and function offloading. Table II summarizes the hardware composition, while Table III details the per-cluster distribution of workers. We employ `Argo Workflows 3.6` to orchestrate workflows, where each workflow step corresponds to a containerized `OpenFaaS` function invoked through HTTPS templates. `OpenFaaS` acts as the FaaS execution substrate for `ClusterLess`, providing per-node function runtimes and exposing warm execution, warm scaling, cold scaling modes. All function images are stored in a `Harbor` registry to ensure consistent versioning and low-latency pulls. Workflow inputs, intermediate artifacts, and final outputs are stored in an S3-compatible `MinIO` backend used by both `Argo` and `OpenFaaS`, guaranteeing storage-consistent, cross-cluster execution and seamless offloading of data-dependent functions.

TABLE II: Hardware composition of the six-cluster edge testbed.

Node type	*Node class*	*CPU cores*	*RAM*	*GPU cores*	*Count*
Intel VMs	XLarge VM	12	32 GB	–	2
	Large VM	8	32 GB	–	2
	Medium VM	4	24 GB	–	6
	Small VM	2	16 GB	–	4
RPis	RPi 4	4	4 GB	–	25
	RPi 4B	4	4 GB	–	4
	RPi 3B+	2	1 GB	–	9
Jetsons	Jetson Nano (JN)	4	4 GB	128	6
	Jetson Orin Nano (JON)	6	8 GB	1024	2
	Jetson Orin AGX (JOA)	12	64 GB	2048	2
AMD server	Physical server	24	32 GB	1536	2

TABLE III: Summary of the six-cluster edge testbed.

Cluster#	*Master*	*Total CPU*	*Total Memory (GB)*	*Workers*								
				VMs				*RPis*			*Jetsons*		
				XL	*L*	*M*	*S*	*4B*	*4*	*3B+*	*JN*	*JON*	*JOA*
C_1	AMD server	82	210	–	–	2	2	1	3	2	2	1	1
C_2	AMD server	76	202	–	–	2	2	1	3	2	2	–	1
C_3	Large VM	54	98	–	–	1	–	1	6	2	1	1	–
C_4	Large VM	36	80	–	–	1	–	1	4	–	1	–	–
C_5	XLarge VM	34	51	–	–	–	–	–	4	3	–	–	–
C_6	XLarge VM	28	48	–	–	–	–	–	4	–	–	–	–

We emulate time-varying network conditions using six independent `4G LTE` bandwidth traces [22], enforced via Linux traffic control with `wondershaper`. Each cluster is assigned a distinct trace, and worker-level traces are phase-shifted to avoid synchronized bandwidth fluctuations. The imposed bandwidth limits uniformly affect both inter-cluster control traffic (e.g., heartbeats and state exchange) and data-plane transfers, ensuring consistent and realistic network dynamics. We collect system telemetry using `Prometheus`, `cAdvisor`, and scripts, providing fine-grained metrics on resource utilization and function lifecycle that `ClusterLess` uses for monitoring during experiments.

B. Case-study serverless workflows

We reimplemented two serverless applications from prior open-source systems: a *text-to-speech censoring (T2SC)* [23] and a *regression-model training* (RT) [24]. We containerized both workflows and made them compatible with `OpenFaaS` and `Argo`, enabling automated DAG execution, artifact propagation, and seamless integration with `ClusterLess` intra- and inter-cluster orchestration schemes. Each workflow is evaluated under three input sizes (*small*, *medium*, *large*) and three deadline classes (*strict*, *moderate*, *lenient*), producing 18 workflow instances per experiment.

1) Text-to-Speech Censoring (T2SC): transforms input text into speech while detecting and censoring profanities. It features an eight-function DAG with a mixture of parallelism and sequential audio-processing steps (Fig. 2 (a)): (a) *GetInput* receives the input text and normalizes it for downstream processing; (b) *T2S* generates a raw speech waveform from the text; (c) *Conversion* transforms the audio to the target format (e.g., sample rate, codec); (d) *Compression* reduces the audio stream size before distribution; (e) *Profanity* runs in parallel with speech generation, detecting profane tokens in the input text; (f) *Merge* joins the compressed audio with the profanity annotations to build a time-aligned censoring map; (g) *Censor* applies muting or beep overlays at profanity locations; (h) *StoreAudio* persists the final censored audio file to the object store. The parallel *text2speech*/*profanity* branch and subsequent merge–censor chain create a critical path

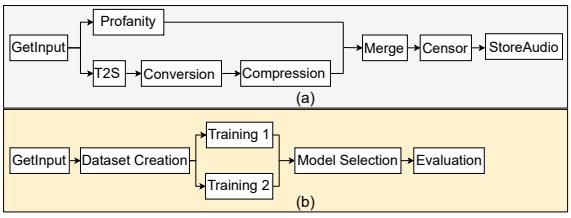


Fig. 2: Case study serverless workflows.

dominated by processing-intensive audio processing, exposing `ClusterLess` ability to coordinate concurrent branches and offload heavy functions under tight E2E deadlines.

2) Regression Tuning (RT): performs E2E regression model selection on a structured dataset. The workflow follows a branched DAG of six functions (Fig. 2 (b)): (a) *GetInput* receives the raw dataset and prepares it for processing; (b) *Dataset Creation* parses, cleans, and partitions the data; (c) *Training 1* and (d) *Training 2* train two regression models with distinct CPU and memory footprints; (e) *Model Selection* compares model accuracy and selects the superior model; (f) *Evaluation* validates the chosen model on a held-out test set. The parallel training stage creates a fork–join structure that stresses resource allocation, while the final selection–evaluation sequence introduces a delay-sensitive path.

C. Baseline methods

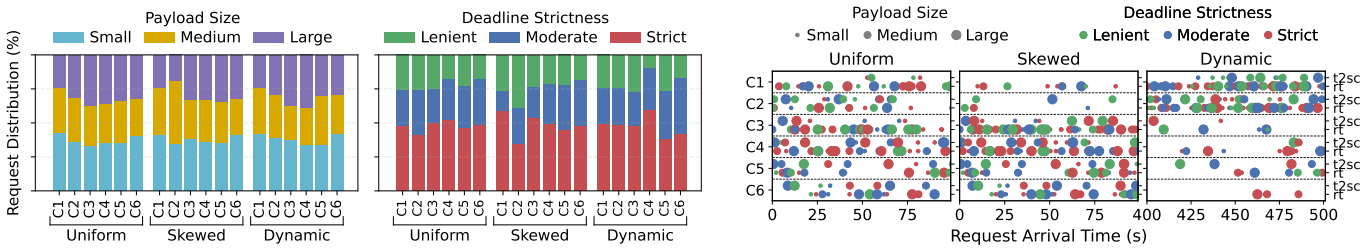
Since no existing serverless workflow orchestration system jointly supports 1) explicit separation between intra- and inter–cluster orchestration and 2) super-master–based coordination across federated clusters, we compare `ClusterLess` (CLU) against four baselines that reflect common design choices in serverless and multi-cluster workflow execution.

1) NKS (Native K8s): represents the standard K8s–Argo execution model, where each workflow function is deployed as a pod and orchestrated independently using default scoring and bin-packing policies. No workflow-level deadline awareness, dependency-aware start-time analysis, execution-mode selection, or inter-cluster orchestration is supported; all functions execute in the submission cluster. Autoscaling relies on the Horizontal Pod Autoscaler, which we configure to follow OpenFaaS-style scaling by comparing the number of active requests against a target concurrency per instance.

2) CLI: isolates `ClusterLess` *intra-cluster orchestration*, enabling dependency-aware earliest-start analysis, resource and bandwidth feasibility checks, and execution-mode selection within a single cluster. However, inter-cluster orchestration is disabled, i.e., if a function cannot be placed locally in a deadline-feasible manner, it is forced to execute on the same cluster, potentially violating the deadline.

3) RRX: extends CLI with deterministic inter-cluster offloading, assigning offloaded functions to clusters in a fixed round-robin order without deadline or load awareness.

4) RNX: augments CLI with inter-cluster offloading by assigning each workflow category (type, payload size, deadline class) to a fixed randomly selected remote cluster. Like RRX, it ignores deadlines and load awareness.



(a) Request size and deadline class distribution per cluster.

(b) Workflow arrival times per cluster.

Fig. 3: Workload composition and temporal arrival behavior across clusters for different arrival rates.

D. Experimental design

We implemented all `ClusterLess` components and algorithms in `Python 3.12`, interacting with the K8s API.

1) Orchestration parameters: We fix the control epoch to $\Delta T = 1\,\text{s}$, the heartbeat timeout to $T_{\text{fail}} = 5\,\text{s}$, and the admissible load threshold to $l = 0.75$.

2) Concurrency limits and resource capacities: For each worker z_{ni}, we set the concurrency capacity R_{ni} proportional to its physical CPU core count, enforcing at most one non-preemptive function per core. The normalized cluster loads $\mathrm{L}_n(t)$ in Eq. (1) and feasibility checks in Eqs. (11)–(13) use the instantaneous computational and bandwidth measurements exported by Prometheus/cAdvisor. Execution modes, warm execution, warm scaling, and cold scaling, are implemented through a custom autoscaling and container-lifecycle controller, designed to replicate `OpenFaaS`-style behavior while allowing explicit control over replica creation, cold-start delays, and concurrency limits on each worker node.

3) Workflow instances: We evaluate all size–deadline combinations defined by the workflow templates, resulting in 18 workload configurations. For the *T2SC* workflow, input sizes are *small* (500 characters), *medium* (1250 characters), and *large* (3750 characters), with deadline classes *lenient*, *moderate*, and *strict* specified as $(130, 180, 150)$, $(100, 130, 180)$, and $(70, 90, 110)$, respectively, where each tuple corresponds to (small, medium, large). For the *RT* workflow, dataset sizes are *small* (25 000), *medium* (100 000), and *large* (250 000) samples, combined with *lenient* $(120, 180, 250)$, *moderate* $(100, 150, 200)$, and *strict* $(80, 110, 150)$ deadlines.

4) Requests arrival model: Workflow requests are generated by first selecting a concrete workflow instance and then assigning an arrival time. Workflow instances are drawn from a *Zipf distribution* [25] over the $K = 18$ size–deadline combinations, with selection probability of i^{th} instance $\mathcal{P}(i) = \frac{1/i^{\alpha}}{\sum_{j=1}^{K} 1/j^{\alpha}}, \alpha = 0.75$. Workflow request arrivals follow a *Poisson* process. We evaluate three arrival regimes by controlling the arrival rate λ over time: a *uniform load* applies a fixed and identical arrival rate $\lambda = 0.33$ across all clusters; b *skewed load* assigns heterogeneous but fixed arrival rates ($\lambda \in \{0.1, 0.05, 0.55, 0.5, 0.4, 0.4\}$) to have persistent spatial load imbalance; and c *dynamic load*, models temporal variations where λ evolves over time, starting from the uniform regime and transitioning to a high-load configuration at predefined time points. The maximum arrival rate is empirically measured and set to 2 based on the highest stable rate observed on the federated testbed.

VII. Evaluation Results

This section compares `ClusterLess` (*CLU*) with baselines, reporting average results and standard deviations.

A. Workload analysis

Fig. 3 summarizes the workload composition and temporal arrival behavior across the six clusters. Fig. 3a shows that the request mix is intentionally controlled and consistent across clusters and arrival regimes: the proportions of payload sizes (small/medium/large) and deadline classes (strict/moderate/lenient) remain stable under different loads. This design isolates the impact of orchestration decisions, such as queueing, execution-mode selection, and offloading, from variations in workload difficulty. Fig. 3b depicts workflow arrival times per cluster. Under the *uniform* regime, arrivals are evenly distributed across clusters over the shown early execution window, yielding balanced concurrency. The *skewed* regime introduces persistent spatial imbalance, with a subset of clusters receiving a disproportionate share of arrivals. The *dynamic* regime shows a later execution window ($t \in [400, 500]$ s) where arrivals become bursty and temporally correlated across clusters, creating short periods of overlapping submissions. This spatial–temporal variability increases instantaneous contention at dependency-constrained stages and thus stresses inter-cluster coordination and deadline-feasible offloading under non-stationary demand.

B. Load-aware super-master analysis

Fig. 4 traces the evolution of orchestration behavior over a 400 s interval as the load on cluster C_1 increases. The shaded area reports the normalized cluster load, while markers show the decision latency of *intra-cluster* and *inter-cluster* orchestration. In the initial phase ($t <$ vertical dashed line), the master of C_1 holds the super-master role and thus performs both local intra-cluster and inter-cluster orchestration. As the arrival rate to C_1 increases (from 0.4 to 0.8), its normalized load rises steadily, reflecting increasing local concurrency and queueing pressure. Throughout this phase, intra-cluster orchestration latency on C_1 remains low and stable, while inter-cluster orchestration incurs higher, but bounded, latency due to its global coordination scope.

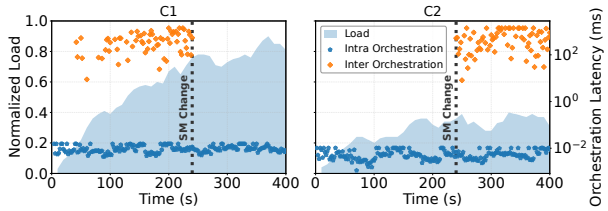


Fig. 4: Super-master behavior under increasing cluster load.

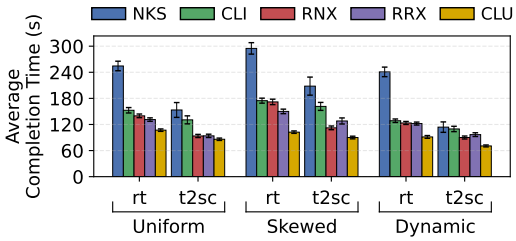


Fig. 5: Average workflow completion time across all clusters.

When the load of C_1 reaches the admissible threshold ($l = 0.75$; vertical dashed line), continuing inter-cluster orchestration on the same master would directly compete with local orchestration for control-plane resources. From this point onward, the super-master role is handled by the master of C_2, while C_1 continues exclusively with intra-cluster orchestration. Consequently, inter-cluster orchestration activity disappears from C_1, and its intra-cluster scheduling latency remains unchanged despite sustained workload. This behavior demonstrates that `ClusterLess` confines global coordination to clusters with sufficient capacity headroom, preventing inter-cluster orchestration from amplifying contention and preserving stable intra-cluster orchestration under increasing load.

C. Completion time analysis

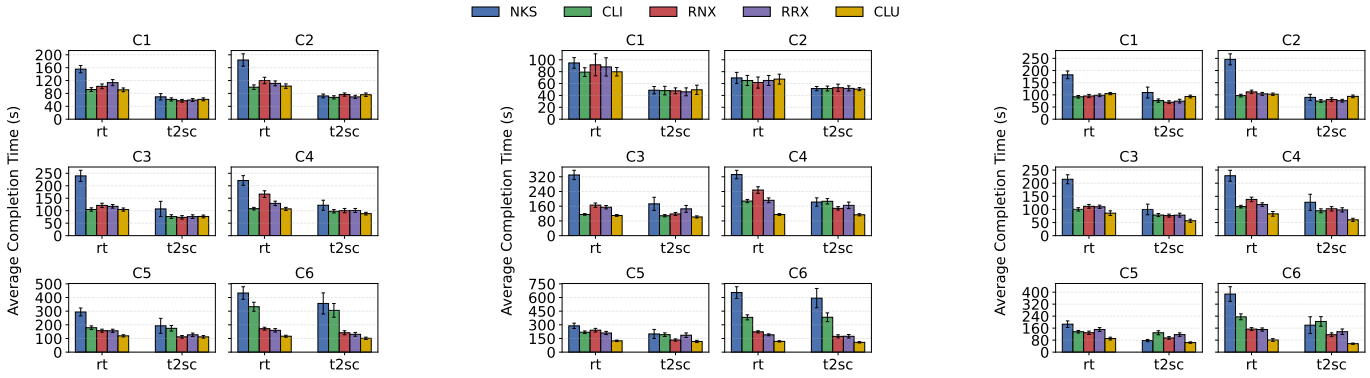
Fig. 6 reports the average workflow completion time per cluster for RT and T2SC workflows under the three arrival regimes, while Fig. 5 summarizes the corresponding behavior across all clusters. Across all regimes, *CLU* achieves the lowest (or tied-lowest) completion time and the smallest variability, indicating robust orchestration under both spatial and temporal load heterogeneity. Under the *uniform* load, arrivals are evenly distributed and completion times remain bounded. *NKS* yields the highest completion times due to the lack of dependency-aware orchestration, execution-mode selection, and offloading, which amplifies queuing along workflow critical paths. *CLI* improves over *NKS* through mode-aware local orchestration but remains constrained under local saturation. *RNX* and *RRX* further reduce completion time by exporting load, yet uninformed target selection introduces unnecessary remote queueing. In contrast, *CLU* achieves the lowest completion times by combining mode-aware intra-cluster execution with deadline-feasible inter-cluster placement, reducing RT completion time by 15 %–21 % relative to *RNX*/*RRX* and by about 10 % for T2SC (Fig. 5). The per-cluster results (Fig. 6) show that these gains are most pronounced on resource-constrained clusters (C_5–C_6).

Under the *skewed* load with persistent spatial imbalance, performance gaps widen markedly. *NKS* exhibits the highest completion times in hotspot clusters (e.g., C_4–C_6), while *CLI* improves local execution but remains constrained by the lack of inter-cluster load redistribution. *RNX* and *RRX* partially alleviate local pressure, yet uninformed target selection results in elevated completion times. In contrast, *CLU* maintains consistently lower RT completion times of about 105 s across clusters, yielding improvements of roughly 30 % over *RRX* and 40 % over *RNX*. Under temporal load variations (i.e., *dynamic* regime), *NKS*, *RNX*, and *RRX* exhibit elevated completion times during burst phases, while *CLI* stabilizes local execution but remains affected by transient saturation. Fig. 5 shows that *CLU* achieves the lowest completion times, improving RT by about 21 %–24 % over *RNX* and *RRX* and T2SC by roughly 22 %–30 %. The per-cluster results in Fig. 6 further show reduced variance across clusters, indicating that *CLU* absorbs bursty arrivals through mode-aware intra-cluster orchestration combined with deadline-aware inter-cluster redistribution.

D. Deadline violation analysis

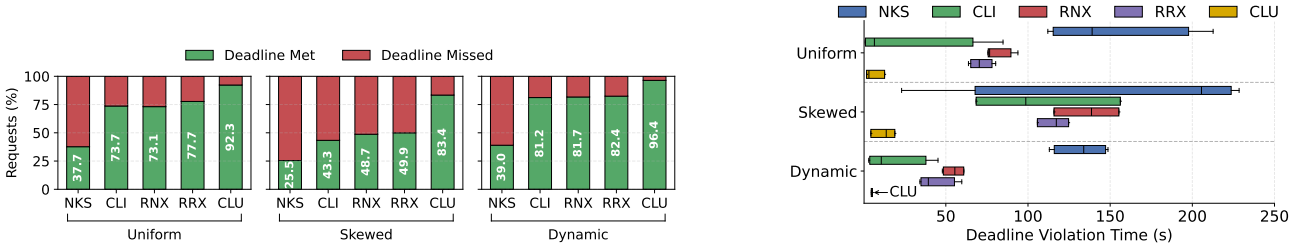
Fig. 7a reports deadline satisfaction rates across arrival regimes. Under the *uniform* regime, *CLU* achieves the highest satisfaction (92.3 %), outperforming *RRX* (77.7 %), *RNX* (73.1 %), and *CLI* (73.7 %), while *NKS* meets only 37.7 % of deadlines. The gap reflects fundamental design differences, i.e., *NKS* suffers from unbounded queueing, *CLI* is constrained by local capacity, and *RNX* and *RRX* incur inefficient remote queueing due to uninformed offloading. In contrast, *CLU* jointly optimizes execution mode and offloading decisions under deadline feasibility, preserving high satisfaction even at moderate load. Under the *skewed* regime, persistent hotspots amplify these effects. As shown in Fig. 7a, satisfaction drops below 50 % for *CLI*, *RNX*, and *RRX*, whereas *CLU* sustains 83.4 %. Fig. 8a explains this gap; for *strict* deadlines under *skewed* regime, *CLU* satisfies 74.5 % of workflows, compared to 35.8 % for *CLI* and 18.2 % for *NKS*. Deadline-aware inter-cluster selection enables *CLU* to relocate only those functions whose end-to-end completion can still meet constraints, preventing deadline loss in overloaded clusters. The *dynamic* regime further stresses temporal adaptability. Fig. 7a shows that satisfaction for *RNX* and *RRX* falls below 83 % during burst phases, while *CLU* reaches 96.4 %.

Fig. 7b quantifies violation severity. *NKS* exhibits the longest violations, frequently exceeding 200 s under *skewed* load due to cascading queue delays along workflow dependencies. *CLI*, *RNX*, and *RRX* reduce violation duration but still incur delays on the order of tens of seconds. In contrast, *CLU* consistently limits violation duration to single-digit seconds, indicating that even when deadlines are missed, violations remain tightly bounded. Finally, Fig. 8b conditions satisfaction on payload size. Under *skewed* load, large-payload workflows achieve below 30 % satisfaction for all baselines, while *CLU* maintains 71 %, showing that *CLU*'s deadline-aware policy



(a) Uniform. (b) Skewed. (c) Dynamic.

Fig. 6: Average workflow completion time per cluster under different arrival regimes.



(a) Deadline satisfaction rate. (b) Violation duration.

Fig. 7: Deadline violation behavior across strategies and arrival rates.

explicitly accounts for both execution and inter-cluster transfer costs, which becomes critical as payload size increases.

E. Offloading behavior and execution mode analysis

Fig. 9a quantifies how offloading-based strategies respond to increasing arrival pressure. Under the *uniform* regime, most workflows are executed locally, but clear differences emerge: *RNX* and *RRX* execute about 81 % of workflows internally, whereas `ClusterLess` executes 86.6 % locally, offloading selectively only when needed. Under the *skewed* regime, offloading becomes essential. *RNX* and *RRX* offload aggressively yet inconsistently, retaining only 65.1 % and 64.2 % of workflows locally, respectively. In contrast, `ClusterLess` maintains a higher internal execution share (76.2 %), indicating deadline-aware, targeted, and stable offloading decisions. Under the *dynamic* regime, `ClusterLess` further increases local execution to 94.4 %, reflecting its ability to absorb bursts through coordinated execution rather than reactive spillover. Fig. 9b reports the execution-mode distribution of `ClusterLess` across clusters and regimes. Warm execution remains dominant, ranging from 65.0 % to 86.1 % under the *uniform* regime, from 67.8 % to 99.5 % under *skewed* load, and from 74.7 % to 96.3 % under the *dynamic* regime. Warm scaling stays bounded, peaking at 19.6 % (*uniform*), 20.1 % (*skewed*), and 20.1 % (dynamic). Cold scaling starts are generally limited but become noticeable on the most constrained cluster: up to 15.3 % in the *uniform* regime and 12.1 % in the *skewed* regime (both on C_6), while remaining below 5.7 % in the *dynamic* regime.

F. CPU utilization analysis

Fig. 10 shows per-cluster CPU utilization over time under different arrival regimes. Under the *uniform* and *skewed* regimes, all strategies exhibit a similar ramp-up followed by steady operation on most clusters, indicating that resource usage is primarily driven by sustained workload intensity rather than orchestration choices. Under the *dynamic* load, clearer differences emerge. CPU utilization becomes uneven across clusters, reflecting the combined effect of bursty arrivals and inter-cluster execution decisions. *CLU* shows higher utilization on some clusters (i.e., C_1–C_2) while maintaining noticeably lower utilization on others (i.e., C_3–C_6), indicating that execution pressure is redistributed across the federation rather than remaining locally concentrated. In contrast, *NKS*, *RNX*, and *RRX* exhibit more uniformly elevated utilization across clusters, consistent with limited or uninformed load redistribution under *dynamic* demand.

VIII. Conclusion

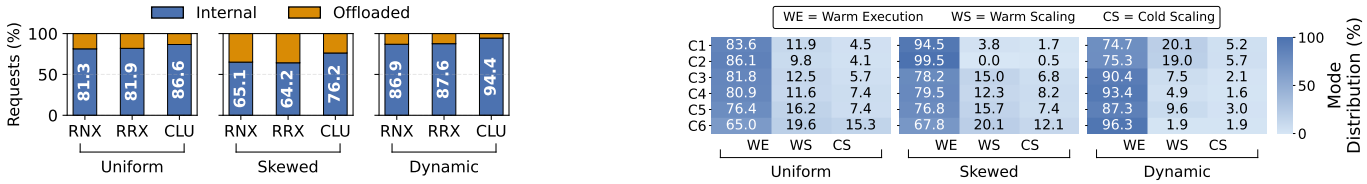
This paper presented `ClusterLess`, a deadline-aware serverless workflow orchestration framework for federated multi-edge Kubernetes clusters. `ClusterLess` combines mode-aware intra-cluster orchestration with super-master–based inter-cluster coordination, accounting for DAG dependencies, workflow deadlines, and heterogeneous compute and network conditions. We implemented `ClusterLess` on six realistic edge clusters using OpenFaaS and Argo and evaluated it with two real workflows and 18



(a) Deadline satisfaction by deadline strictness.

(b) Deadline satisfaction by payload size.

Fig. 8: Deadline satisfaction across strategies and arrival rates under different workload constraints.



(a) Local vs. offloaded function execution.

(b) Execution-mode distribution of `ClusterLess`.

Fig. 9: Offloading behavior and execution-mode selection under different arrival regimes.

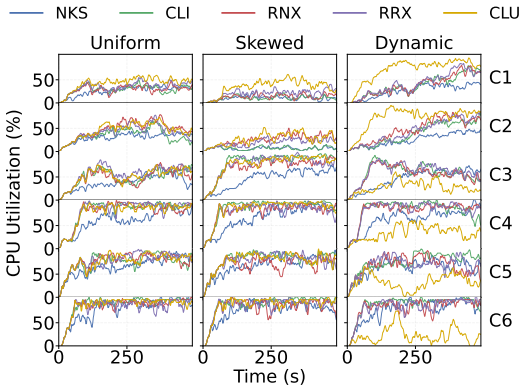


Fig. 10: Per-cluster CPU utilization in different methods.

workload configurations. Results show that `ClusterLess` reduces workflow completion time and deadline violations compared to four baselines. Future work will explore multi-objective and learning-based orchestration.

References

[1] "Gartner." https://www.gartner.com/en/newsroom/press-releases/2023-10-30-gartner-says-50-percent-of-critical-enterprise-applications-will-reside-outside-of-centralized-public-cloud-locations-through-2027.

[2] A. Joosen *et al.*, "How Does it Function? Characterizing Long-Term Trends in Production Serverless Workloads," in *Proc. of the 2023 ACM Symp. on Cloud Computing*, 2023.

[3] R. Farahani *et al.*, "Serverless Workflow Management on the Computing Continuum: A Mini-Survey," in *15th ACM/SPEC Intl. Conf. on Performance Engineering*, 2024.

[4] S. K. Mondal *et al.*, "Kubernetes in IT Administration and Serverless Computing: An Empirical Study and Research Challenges," *The Journal of Supercomputing*, 2022.

[5] M. S. Aslanpour *et al.*, "FaasHouse: Sustainable Serverless Edge Computing through Energy-Aware Resource Scheduling," *IEEE Tran. on Services Computing*, 2024.

[6] C. Carrión, "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges," *ACM Computing Surveys*, 2022.

[7] R. Farahani and R. Prodan, "EnergyLess: An Energy-Aware Serverless Workflow Batch Orchestration on the Computing Continuum," in *IEEE Intl. Conf. on Cloud Computing*, IEEE, 2025.

[8] H. Shafiei *et al.*, "Serverless Computing: A Survey of Opportunities, Challenges, and Applications," *ACM Computing Surveys*, 2022.

[9] S. S. Gill *et al.*, "Modern Computing: Vision and Challenges," *Telematics and Informatics Reports*, 2024.

[10] R. Farahani *et al.*, "Heftless: A Bi-Objective Serverless Workflow Batch Orchestration on the Computing Continuum," in *IEEE Intl. Conf. on Cluster Computing*, IEEE, 2024.

[11] M. Michalke *et al.*, "Evaluating the Impact of Inter-cluster Communications in Edge Computing," in *IEEE Network Operations and Management Symp.*, IEEE, 2025.

[12] D. Bachar *et al.*, "Optimizing Service Selection and Load Balancing in Multi-Cluster Microservice Systems with MCOSS," in *2023 IFIP Networking Conf.*, IEEE, 2023.

[13] L. Poggiani *et al.*, "Live Migration of Multi-Container Kubernetes Pods in Multi-Cluster Serverless Edge Systems," in *Proc. of the 1st Workshop on Serverless at the Edge*, 2024.

[14] H. Park *et al.*, "HEART: Heterogeneous-Aware Traffic Allocation in Multi-Replica Deployments on Kubernetes," in *2025 IEEE 18th Intl. Conf. on Cloud Computing*, IEEE, 2025.

[15] "Karmada." Accessed: 2025-01-10.

[16] D. Balla *et al.*, "Open Source FaaS Performance Aspects," in *2020 43rd Intl. Conf. on Telecommunications and Signal Processing*, IEEE, 2020.

[17] P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-based Approach," *arXiv preprint arXiv:1903.12221*, 2019.

[18] L. Cvetković *et al.*, "Dirigent: Lightweight Serverless Orchestration," in *Proc. of the ACM SIGOPS Symp. on Operating Systems Principles*, 2024.

[19] E. Simion *et al.*, "Towards Seamless Serverless Computing Across an Edge-Cloud Continuum," in *Proce. of the IEEE/ACM 16th Intl. Conf. on Utility and Cloud Computing*, 2023.

[20] P. G. López *et al.*, "Triggerflow: Trigger-based Orchestration of Serverless Workflows," in *Proc. of the 14th ACM Intl. Conf. on Distributed and Event-Based Systems*, 2020.

[21] J. Serenari *et al.*, "GreenWhisk: Emission-Aware Computing for Serverless Platform," in *IEEE Intl. Conf. on Cloud Engineering*, IEEE, 2024.

[22] D. Raca *et al.*, "Beyond Throughput: A 4G LTE Dataset with Channel and Context Metrics," in *Proc. of the 9th ACM Multimedia Systems Conference*, 2018. Available at: https://zenodo.org/records/1219679.

[23] S. Eismann *et al.*, "Predicting the Costs of Serverless Workflows," in *Proc. of the 2020 ACM/SPEC International Conf. on Performance Engineering*. Available at: https://github.com/jacopotagliabue/no-ops-machine-learning, year = 2020.

[24] "Regression Tuning Workflow." Available at https://github.com/jacopotagliabue/no-ops-machine-learning.

[25] L. Cherkasova and M. Gupta, "Analysis of Enterprise Media Server Workloads: Access Patterns, Locality, Content Evolution, and Rates of Change," *IEEE/ACM Trans. on Networking*, 2004.