Matching-based Scheduling of Asynchronous Data Processing Workflows on the Computing Continuum

Narges Mehran, Zahra Najafabadi Samani, Dragi Kimovski, Radu Prodan
Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Austria
Email: {name}.{surname}@aau.at

***Abstract*—Today's distributed computing infrastructures encompass complex workflows for real-time data gathering, transferring, storage, and processing, quickly overwhelming centralized cloud centers. Recently, the computing continuum that federates the Cloud services with emerging Fog and Edge devices represents a relevant alternative for supporting the next-generation data processing workflows. However, eminent challenges in automating data processing across the computing continuum still exist, such as scheduling heterogeneous devices across the Cloud, Fog, and Edge layers.**

We propose a new scheduling algorithm called C^3-MATCH, based on matching theory principles, involving two sets of players negotiating different utility functions: 1) workflow microservices prefering computing devices with lower data processing and queuing times; 2) computing continuum devices prefering microservices with corresponding resource requirements and less data transmission time. We evaluate C^3-MATCH using real-world road sign inspection and sentiment analysis workflows on a federated computing continuum across four Cloud, Fog, and Edge providers. Our combined simulation and real execution results reveal that C^3-MATCH achieves up to 67% lower completion time than three state-of-the-art methods with $10\,\text{ms}$-$1000\,\text{ms}$ higher transmission time.

***Index Terms*—Computing continuum, Cloud, Fog, Edge, asynchronous communication, scheduling, matching theory.**

I. INTRODUCTION

Modern data processing workflows, such as machine learning (ML) [1], encompass complex workflows for real-time data gathering, storage, and analysis, quickly overwhelming centralized Cloud data centers [2], [3]. Workflow applications often have conflicting needs, such as low communication latency and high computational workload, which require heterogeneous resources distributed across geographical locations. Recently, the *computing continuum* that federates the Cloud services with emerging Fog and Edge devices presents a relevant alternative for data processing operations [4].

Traditional management methods consolidate the applications on a limited number of Fog and Edge resources, leading to over-provisioning and communication bottlenecks [5], [6]. Furthermore, they do not consider asynchronous data exchange patterns, covering 23% of the communication between microservices according to an Alibaba trace analysis [7]. Therefore, challenges in automating data processing workflows across the computing continuum remain [8], including resource allocation, scheduling, and microservices orchestration.

To address these challenges, we developed in previous work an application placement algorithm, named CODA, based on matching theory principles [9]. CODA targets pure placement of application workflows for their entire duration without considering the microservices' earliest start and finish times. This simple placement method often leads to low resource utilization rates and increased execution costs. Finally, CODA assumes synchronous communication among microservices with longer rendezvous delays and lacks mechanisms for asynchronous data exchange and execution.

We address these weaknesses by proposing a novel method and capacity-aware algorithm named C^3-MATCH for scheduling asynchronous data processing workflows, represented as directed acyclic graphs of microservices, in the computing continuum. We model the asynchronous communication between microservices using data element queues, expressing temporary storage of communicating data elements. C^3-MATCH approaches the scheduling problem through a matching theoretic optimization heuristic involving two sets of players [9] negotiating different utility functions:

- *Data processing microservices* that rank the continuum devices based on their data processing and queuing times;
- *Cloud, Fog, and Edge devices* that rank the microservices based on their data transmission times.

C^3-MATCH schedules the microservices on devices based on their mutual preferences, aiming to maximize the aggregate utility of the application and the resource provider.

The main contributions of this work are:

- A model for quantifying the asynchronous data processing and queuing times between various microservices of a distributed workflow application;
- A capacity-aware matching theoretic scheduling algorithm in the computing continuum optimizing the aggregate utility of applications and resource providers;
- A detailed evaluation using two real-world workflows executed on a real testbed federating eleven heterogeneous virtual instances across four geographically distributed providers in the Cloud, Fog, and Edge continuum;

- An empirical analysis of the C^3-MATCH algorithm lowering workflow completion time by $1 - 67\%$ compared to three related methods.

The paper has ten sections. We first survey the related work in Section II and elaborate a formal model for our method in Section III. We describe the C^3-MATCH architecture in Section IV, followed by its scheduling algorithm in Section V and an illustrative example in Section VI. Section VII describes the experimental design of the evaluations, including two case study descriptions VII-D. Section VIII elaborates on the simulation results, validated by real-world evaluations in Section IX. Section X concludes the paper.

II. RELATED WORK

This section reviews the workflow application scheduling state-of-the-art in Cloud, Fog, and Edge infrastructures.

1) Processing time reduction: Menouer [10] presented a scheduling model based on preference, similar to the ideal decision-making algorithm (Topsis) that optimizes multiple criteria, including resource utilization and energy consumption. Wu et al. [11] proposed a metaheuristic-based scheduling method that investigates task dependencies in a data processing workflow and prioritizes the completion of a higher number of microservices execution. Although these works employ multi-criteria scheduling methods, they neglect the transmission time and latency from the data producers to the Cloud in an extended Fog and Edge environment.

2) Queuing time reduction: Abdelwahab et al. [12] designed a publish-subscribe flocking clustering algorithm to replicate the application among the Edge devices minimizing the end-to-end latency. Gupta et al. [13] proposed a publish-subscribe architecture for processing data and latency-sensitive applications on a geo-distributed Edge infrastructure. Pusztai et al. [14] presented a greedy integer linear programming method for asynchronous application placement with minimal resource cost, which forwards microservices to the Cloud if the Edge devices cannot host them. These works just utilized a single element queue shared among all the microservices; however, we model a separate queue for each microservice.

3) Transmission time reduction: Fahs and Pierre [6] proposed a Fog scheduling model that minimizes user-to-replica latency and balances the application workloads. Lujic et al. [15] designed a scheduling method for analytics workflows named SEA-LEAP, which reduces the completion time and considers the locality of Edge devices to data producers. However, these works focus on latency-sensitive applications in Fog and Edge environments isolated from the Cloud.

4) Data traffic reduction: Arkian et al. [16] presented a geo-distributed stream processing model to sustain a sufficient parallel throughput among Edge devices, optimizing the network latency and resource utilization. Tamiru et al. [17] designed and integrated a scheduler in the Kubernetes orchestration platform that models the workload requirements, resource capacity, and the ingress traffic to multiple computing clusters. Mehran et al. [9] proposed a two-sided matching and resource allocation model for streaming applications that improves the data processing time of microservices and residual bandwidth of Cloud and Fog devices. However, these methods target specific data processing tools on the Fog or ignore the data processing and transmission times as conflicting objectives.

5) C^3-MATCH contribution: Related methods model the workflow scheduling as an optimization problem that minimizes the data transmission and processing times but neglects asynchronous data exchange and device utilization in the computing continuum. We extend these methods by researching a scheduling method based on a two-sided matching algorithm that *maximizes the aggregation of two opposite stakeholder interests*: 1) minimization of data processing and queuing times from the workflow owner perspective; 2) minimization of data transmission time from the resource provider perspective.

III. MODEL

This section presents the formal model underneath our work.

A. Application model

1) Data processing workflow: is a directed acyclic graph $\mathcal{W} = (\mathcal{M}, \mathcal{E}, \mathcal{M}_{\texttt{src}}, \mathcal{M}_{\texttt{snk}})$, consisting of:

a) A set of $\mathcal{N}_{\mathcal{M}}$ interconnected data processing microservices*:* $\mathcal{M} = \{m_i \mid 0 \leq i < \mathcal{N}_{\mathcal{M}}\}$;

b) A set of data flows*:* $\texttt{data}_{ui}$ streaming from an upstage microservice $m_u \in \mathcal{M}$ to a downstage microservice $m_i \in \mathcal{M}$: $\mathcal{E} = \{(m_u, m_i, \texttt{data}_{ui}) \,|(m_u, m_i) \in \mathcal{M} \times \mathcal{M}\}$. A data flow consists of a sequence of $\texttt{Size}_{ui}$ *data elements* $e = \texttt{data}_{ui}[x]$, where $1 \leq x \leq \texttt{Size}_{ui}$. We further denote a generic data element as e to simplify the notation;

c) A set of producers*:* $\mathcal{M}_{\texttt{src}} \subset \mathcal{M}$ generating data that require further processing from the workflow application. A producer has no upstage microservices: $\nexists\,(m_i, \texttt{src}, \_) \in \mathcal{E}, \forall\ \texttt{src} \in \mathcal{M}_{\texttt{src}} \wedge m_i \in \mathcal{M}$;

d) A set of consumers*:* $\mathcal{M}_{\texttt{snk}} \subset \mathcal{M}$ collecting and presenting the application output. A consumer has no downstage microservices: $\nexists\,(\texttt{snk}, m_i, \_) \in \mathcal{E}, \forall \texttt{snk} \in \mathcal{M}_{\texttt{snk}} \wedge m_i \in \mathcal{M}$.

2) Dependency level: $l\,(m_i)$ of a microservice $m_i \in \mathcal{M}$ is the maximum number of data flow microservice connections separating it from a producer in $\mathcal{M}_{\texttt{src}}$ [18]:

$$l\,(m_i) = \begin{cases} 0, & m_i \in \mathcal{M}_{\texttt{src}}; \\ \max\limits_{(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} l\,(m_u) + 1, & m_i \notin \mathcal{M}_{\texttt{src}}. \end{cases}$$

3) Resource requirements: $\texttt{req}\,(m_i, e)$ for proper processing of a data element $e = \texttt{data}_{ui}[x]$ $(1 \leq x \leq \texttt{Size}_{ui})$ by a microservice m_i is a triple representing the minimum processing load $\texttt{CPU}\,(m_i, e)$ (in million of instructions (MI)), memory $\texttt{MEM}\,(m_i, e)$ and storage $\texttt{STOR}\,(m_i, e)$ sizes (in MB):

$$\texttt{req}\,(m_i, e) = (\texttt{CPU}\,(m_i, e)\,, \texttt{MEM}\,(m_i, e)\,, \texttt{STOR}\,(m_i, e))\,.$$

B. Resource model

We model the computing continuum as a heterogeneous set of capacity-constrained devices.

1) Devices: $\mathcal{D} = \{d_j | 0 \leq j < \mathcal{N}_{\mathcal{D}}\}$ represent a set of $\mathcal{N}_{\mathcal{D}}$ Cloud, Fog, and Edge resources in the computing continuum.

2) Capacity: $c_j = (\texttt{CPU}_j, \texttt{MEM}_j, \texttt{STOR}_j)$ of a device $d_j \in \mathcal{D}$ is a vector representing its available processing speed $\texttt{CPU}_j$ (in MI per second), memory $\texttt{MEM}_j$ and storage $\texttt{STOR}_j$ sizes, depending on its utilization. A device can be in three states based on an *availability threshold* θ of its resources:

a) Under-utilized: indicating positive available capacity: $c_j > 0 \iff \texttt{CPU}_j > \theta \wedge \texttt{MEM}_j > \theta \wedge \texttt{STOR}_j > \theta$;

b) Fully-utilized: indicating nearly zero free capacity: $c_j \approx 0 \iff 0 \leq \texttt{CPU}_j \leq \theta \vee 0 \leq \texttt{MEM}_j \leq \theta \vee 0 \leq \texttt{STOR}_j \leq \theta$;

c) Over-utilized: indicating over-committed capacity: $c_j < 0 \iff \texttt{CPU}_j < 0 \vee \texttt{MEM}_j < 0 \vee \texttt{STOR}_j < 0$.

C. Processing model

We define two metrics for asynchronous processing of a data element $e = \texttt{data}_{ui}[x]$.

1) Element processing time: $t_p(m_i, e, d_j)$ by a microservice m_i on a device d_j is the ratio between its computational workload $\texttt{CPU}(m_i, e)$ (in MI) and the processing speed $\texttt{CPU}_j$:

$$t_p(m_i, e, d_j) = \frac{\texttt{CPU}(m_i, e)}{\texttt{CPU}_j}.$$

2) Processing rate: τ_{ui} of the data elements $e \in \texttt{data}_{ui}$ by a microservice m_i on a device d_j is their inverse average processing time:

$$\tau_{ui} = \frac{\texttt{Size}_{ui}}{\sum\limits_{e \in \texttt{data}_{ui}} t_p(m_i, e, d_j)}.$$

D. Arrival model

1) Element transmission time: $t_c(m_u, e, m_i)$ from an upstage microservice m_u to a downstage microservice m_i aggregates the latency and ratio of the data element size to the network bandwidth between their hosting devices d_k and d_j:

$$t_c(m_u, e, m_i) = \texttt{LAT}_{kj} + \frac{\texttt{sizeof}(e)}{\texttt{BW}_{kj}},$$

where $\texttt{BW}_{kj}$ and $\texttt{LAT}_{kj}$ represent the network bandwidth and the round-trip latency between the devices $d_k = \texttt{sched}(m_u)$ and $d_j = \texttt{sched}(m_i)$.

2) Arrival time: t_a of a data element e transmitted from an upstage microservice $m_u \in \mathcal{M}$ scheduled on a device d_k to a downstage microservice $m_i \in \mathcal{M}$ on a device d_j is the sum of its processing (if not a producer) and transmission times:

$$t_a(m_u, e, m_i) = \begin{cases} t_c(m_u, e, m_i), & m_u \in \mathcal{M}_{\texttt{src}}; \\ t_p(m_u, e, d_k) + t_c(m_u, e, m_i), & m_u \notin \mathcal{M}_{\texttt{src}}. \end{cases}$$

E. Queuing model

We represent the asynchronous data flow between the workflow microservices [19] using a set of queues $\mathcal{Q} = \{\texttt{qu}_{ui} | (m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}\}$.

1) Element queue: $\texttt{qu}_{ui} = (\lambda_{ui}, \tau_{ui}, \texttt{quSize}_{ui}) \in \mathcal{Q}$ buffers $\texttt{quSize}_{ui}$ data elements $e = \texttt{data}_{ui}[x]$ received by a microservice m_i at a rate λ_{ui} from an upstage microservice m_u and processed at a rate τ_{ui} in FIFO order [20].

2) Arrival rate: λ_{ui} in an element queue $\texttt{qu}_{ui}$ is indirectly proportional to the average data element inter-arrival time between two subsequent data elements from an upstage microservice m_u to the downstage microservice m_i [21]:

$$\lambda_{ui} = \left(\overline{\Delta t_a(m_u, m_i)}\right)^{-1}.$$

3) Element processing and queuing time: $t_{qp}(m_u, e, m_i)$ is the average queuing time of elements $e = \texttt{data}_{ui}[x]$ in the element queue $\texttt{qu}_{ui}$ and processed by a microservice m_i [22]:

$$t_{qp}(m_u, e, m_i) = (\tau_{ui} - \lambda_{ui})^{-1},$$

assuming that the queue size (evaluated in Section VII-B) and the number of elements processed by m_i reaches a steady state, following the $M/M/1$ and the Little's queuing model [23].

F. Execution model

1) Earliest start time: $\texttt{EST}(m_i)$ of a microservice m_i is the minimum arrival time of the first data element from its upstage microservices:

$$\texttt{EST}(m_i) = \begin{cases} 0, & m_i \in \mathcal{M}_{\texttt{src}}; \\ \min\limits_{(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} \{\texttt{EST}(m_u) + t_a(m_u, \texttt{data}_{ui}[0], m_i)\}, & m_i \notin \mathcal{M}_{\texttt{src}}. \end{cases}$$

2) Earliest finish time: $\texttt{EFT}(m_i)$ of a microservice m_i aggregates the earliest start time $\texttt{EST}(m_i)$, and the maximum aggregated transmission, queuing and processing times $T_{cqp}(m_u, \texttt{data}_{ui}, m_i)$:

$$\texttt{EFT}(m_i) = \begin{cases} 0, & m_i \in \mathcal{M}_{\texttt{src}}; \\ \texttt{EST}(m_i) + \max\limits_{(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} \{T_{cqp}(m_u, \texttt{data}_{ui}, m_i)\}, & m_i \notin \mathcal{M}_{\texttt{src}}, \end{cases}$$

$$T_{cqp}(m_u, \texttt{data}_{ui}, m_i) = \sum_{e \in \texttt{data}_{ui}} \{t_{qp}(m_u, e, m_i) + t_c(m_u, e, m_i)\}.$$

3) Completion time: $\texttt{CT}(\mathcal{W}, \mathcal{D})$ is the latest earliest finish time among all its microservices preceding a $\texttt{snk}$:

$$\texttt{CT}(\mathcal{W}, \mathcal{D}) = \max_{\substack{(m_i, \texttt{snk}, \texttt{data}_{i,\texttt{snk}}) \in \mathcal{E} \wedge \\ \texttt{snk} \in \mathcal{M}_{\texttt{snk}} \wedge d_j = \texttt{sched}(m_i)}} \texttt{EFT}(m_i).$$

G. Scheduling model

We define in this section the scheduling problem and the optimization function.

1) Microservice scheduling: function $\texttt{sched} : \mathcal{M} \to \mathcal{D}$ assigns a microservice $m_i \in \mathcal{M}$ to a device $d_j = \texttt{sched}(m_i)$. We represent the scheduling problem as a matching game using two sets of players [24]: 1) the microservices $\mathcal{M}$ of the data processing workflow $\mathcal{W}$, and 2) the devices $\mathcal{D}$. The game aims to find a scheduling function mapping the microservices $m_i \in \mathcal{M}$ to appropriate devices $d_j = \texttt{sched}(m_i) \in \mathcal{D}$ with sufficient capacity using a double ranking and utility strategy for microservices and devices.

a) Allocation set: $\texttt{alloc}[d_j]$ of a device $d_j \in \mathcal{D}$ represents its hosted (scheduled) microservices: $\texttt{alloc}[d_j] = \{m_i \in \mathcal{M} \mid \texttt{sched}(m_i) = d_j\}$.

2) Microservice-side ranking: orders the devices for a microservice m_i in a preference list $\texttt{DPL}[m_i]$ based on the aggregated data processing and queuing time:

$$T_{qp}(m_i, \texttt{data}_{ui}) = \sum_{e \in \texttt{data}_{ui}} t_{qp}(m_u, e, m_i).$$

3) Microservice-side utility: $U_m(m_i, d_j)$ is the gain from scheduling a microservice m_i on a device $d_j \in \texttt{DPL}[m_i]$:

$$U_m(m_i, d_j) = \frac{\left|T_{qp}(m_i, \texttt{data}_{ui}) - \textsc{Last}_{T_{qp}}(\texttt{DPL}[m_i])\right|}{\left|\textsc{First}_{T_{qp}}(\texttt{DPL}[m_i]) - \textsc{Last}_{T_{qp}}(\texttt{DPL}[m_i])\right|},$$

where the $\textsc{First}_{T_{qp}}(\texttt{DPL}[m_i])$ and $\textsc{Last}_{T_{qp}}(\texttt{DPL}[m_i])$ are the lowest, respectively the highest data processing and queuing times in the device preference list. The first device in the preference list has the highest utility, equal to one.

4) Device-side ranking: orders the microservices for a device d_j in a preference list $\texttt{MPL}[d_j]$ based on the data transmission time $T_c(m_u, \texttt{data}_{ui}, m_i)$:

$$T_c(m_u, \texttt{data}_{ui}, m_i) = \sum_{e \in \texttt{data}_{ui}} t_c(m_u, e, m_i).$$

5) Device-side utility: $U_d(m_i, d_j)$ represents the gain of a device d_j for hosting a microservice $m_i \in \texttt{MPL}[d_j]$:

$$U_d(m_i, d_j) = \frac{|T_c(m_u, \texttt{data}_{ui}, m_i) - \textsc{Last}_{T_c}(\texttt{MPL}[d_j])|}{|\textsc{First}_{T_c}(\texttt{MPL}[d_j]) - \textsc{Last}_{T_c}(\texttt{MPL}[d_j])|},$$

where the $\textsc{First}_{T_c}(\texttt{MPL}[d_j])$ and $\textsc{Last}_{T_c}(\texttt{MPL}[d_j])$ are the lowest, respectively, the highest data transmission times.

6) Objective function: of the scheduling algorithm maximizes the aggregate utility $U(\mathcal{W}, \mathcal{D})$ of the average of microservices $m_i \in \mathcal{M}$ and devices $d_j \in \mathcal{D}$ [25]:

$$U(\mathcal{W}, \mathcal{D}) = \tfrac{1}{|\texttt{DPL}|} \cdot \sum_{\substack{m_i \in \mathcal{M} \wedge \\ d_j \in \texttt{DPL}[m_i]}} U_m(m_i, d_j) + \tfrac{1}{|\texttt{MPL}|} \cdot \sum_{\substack{d_j \in \mathcal{D} \wedge \\ m_i \in \texttt{MPL}[d_j]}} U_d(m_i, d_j),$$

where $|\texttt{DPL}|$ and $|\texttt{MPL}|$ are the cardinality of the two lists and:

1) A microservice m_i in $\mathcal{M}$ is scheduled on exactly one device from its preference list:

$$\texttt{sched}(m_i) \in \texttt{DPL}[m_i] \subseteq \mathcal{D} \ \wedge \ |\texttt{sched}(m_i)| = 1;$$

2) A device d_j can execute multiple microservices that are members of its preference list and within its capacity:

$$\texttt{alloc}[d_j] \subseteq \texttt{MPL}[d_j] \subseteq \mathcal{M} \wedge \texttt{req}(m_i, \texttt{data}_{ui}) \ll c_j, \forall m_i \in \texttt{MPL}[d_j];$$

3) The matching game does not contain any pair of microservices and devices that prefer matching each other rather than their current allocations [24].

IV. C^3-Match Architecture

Figure 1 shows the C^3-Match design with five modules.

1) Dependency level identification: component analyzes the microservices data flow and assigns independent microservices to the same level (see Section III-A).

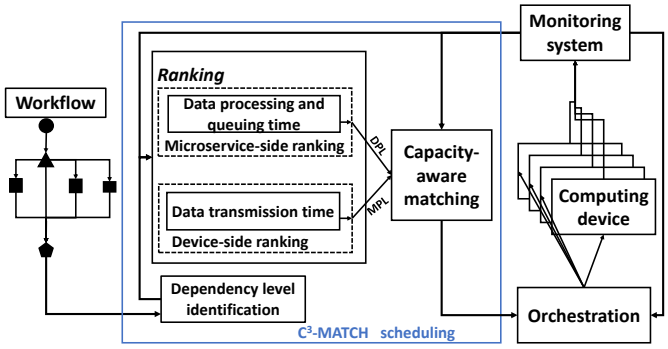


Fig. 1: C^3-Match scheduling architecture.

2) Ranking: component calculates the ordered preference lists for the computing devices and independent microservices in a dependency level of a workflow application, along with the monitoring information, as presented in Section III-G.

3) Capacity-aware matching: applies a game theoretic algorithm to the microservices and their device preference lists in each dependency level, considered as players. The result is a mapping of the microservices in each dependency level to devices for parallel execution based on their earliest start time (see Section III-F).

4) Orchestration: deploys the data processing microservices on the computing continuum devices based on their schedules, and the earliest start and finish times.

5) Monitoring system: observes the performance of the data processing workflow over the devices and reports the information to the C^3-Match scheduler and orchestrator.

V. C^3-Match Scheduling Algorithm

C^3-Match depicted in Algorithm 1 applies matching game principles to schedule the microservices of a data processing workflow $\mathcal{W}$ on the continuum devices $\mathcal{D}$. Firstly, the algorithm calculates an array of dependency levels $\mathcal{L}$ in lines 3–6, where each level $\mathcal{L}[l]$ represents a set of independent microservices separated from a producer in $\mathcal{M}_{\texttt{src}}$ by a maximum of l data flow microservice connections. Then, the algorithm iterates over the dependency levels to create the device preference lists for its independent microservices (line 8) and the microservice preference lists for the devices (line 9). Line 10 finds appropriate mappings for each microservice to the preferred devices and allocates the required memory and storage based on its available capacity. Finally, the algorithm estimates each microservice's earliest start and finish times (lines 11–14) used by the orchestration to reserve and release the assigned devices. The algorithm returns the workflow scheduling Gantt chart in line 16, comprising each microservice's earliest start and finish times.

1) Microservice-side ranking: presented in Algorithm 2, receives the data processing workflow $\mathcal{W}$, a set of devices $\mathcal{D}$, and a microservice set $\mathcal{S}$ within a certain dependency level. The algorithm initializes the empty device preference lists $\texttt{DPL}[m_i]$ for every microservice m_i in line 2. Afterward, each microservice ranks the device in lines 3–9 by first filtering

Algorithm 1 C^3-MATCH scheduling.

Input: $\mathcal{W} = (\mathcal{M}, \mathcal{E}, \mathcal{M}_{\texttt{src}}, \mathcal{M}_{\texttt{snk}})$ ▷ Data processing workflow
$\mathcal{D} = \{d_j \mid 0 \leq j < \mathcal{N}_{\mathcal{D}}\}$ ▷ Cloud, Fog, and Edge device set
Output: $\texttt{sched}(m_i), \texttt{EST}(m_i), \texttt{EFT}(m_i); \forall m_i \in \mathcal{W}$ ▷ Schedule, earliest start time, and earliest finish time of all microservices

1: **function** C^3-MATCH($\mathcal{W}, \mathcal{D}$)
2: $\texttt{sched}(m_i) \leftarrow \emptyset, \forall m_i \in \mathcal{M}$ ▷ Initialize scheduling function
3: **for all** $m_i \in \mathcal{M}$ **do** ▷ Calculate dependency levels
4: $l(m_i) \leftarrow \max_{(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} l(m_u) + 1$
5: $\mathcal{L}\left[l(m_i)\right] \leftarrow \mathcal{L}\left[l(m_i)\right] \bigcup \{m_i\}$ ▷ Add m_i to its corresponding level
6: **end for**
7: **for all** $l \in [1, \text{SIZEOF}(\mathcal{L})]$ **do**
8: $\texttt{DPL} \leftarrow$ RANKDEVICES($\mathcal{W}, \mathcal{D}, \mathcal{L}[l]$) ▷ Call Algorithm 2
9: $\texttt{MPL} \leftarrow$ RANKMICROSERVICES($\mathcal{W}, \mathcal{D}, \mathcal{L}[l], \texttt{DPL}$) ▷ Call Algorithm 3
10: $\texttt{sched} \leftarrow$ MATCH($\mathcal{W}, \mathcal{D}, \mathcal{L}[l], \texttt{MPL}, \texttt{DPL}, \texttt{sched}$) ▷ Call Algorithm 4
11: **for all** $m_i \in \mathcal{L}[l] \wedge (m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}$ **do**
12: $\texttt{EST}(m_i) \leftarrow \min_{\forall(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} \left(\texttt{EST}(m_u) + t_a(m_u, \texttt{data}_{ui}[0], m_i)\right)$
13: $\texttt{EFT}(m_i) \leftarrow \texttt{EST}(m_i) + \max_{\forall(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} \left(T_{cqp}(m_u, \texttt{data}_{ui}, m_i)\right)$
14: **end for**
15: **end for**
16: **return** $(\texttt{sched}, \texttt{EST}, \texttt{EFT})$;
17: **end function**

Algorithm 2 Microservice-side ranking.

Input: $\mathcal{W} = (\mathcal{M}, \mathcal{E}, \mathcal{M}_{\texttt{src}}, \mathcal{M}_{\texttt{snk}})$ ▷ Data processing workflow
$\mathcal{D} = \{d_j \mid 0 \leq j < \mathcal{N}_{\mathcal{D}}\}$ ▷ Device set
$\mathcal{S} \subset \mathcal{M}$ ▷ Microservice set in a dependency level
Output: $\texttt{DPL}[m_i], \forall m_i \in \mathcal{S}$ ▷ Device preference lists of microservices in $\mathcal{S}$

1: **function** RANKDEVICES($\mathcal{W}, \mathcal{D}, \mathcal{S}$)
2: $\texttt{DPL}[m_i] \leftarrow \emptyset, \forall m_i \in \mathcal{S}$ ▷ Initialize DPL
3: **for all** $m_i \in \mathcal{S}$ **do** ▷ Iterate microservices
4: **for all** $d_j \in \mathcal{D}$ **do**
5: **if** $(\texttt{MEM}(m_i, \texttt{data}_{ui}) < \texttt{MEM}_j) \wedge (\texttt{STOR}(m_i, \texttt{data}_{ui}) < \texttt{STOR}_j)$ **then** ▷ Check resource availability
6: $\texttt{DPL}[m_i] \leftarrow \texttt{DPL}[m_i] \bigcup \left(d_j, \max_{\forall(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} T_{qp}(m_i, \texttt{data}_{ui})\right)$
7: **end if** ▷ Add d_j and its T_{qp} to DPL
8: **end for**
9: **end for**
10: **for all** $m_i \in \mathcal{S} \wedge \texttt{DPL}[m_i] \neq \emptyset$ **do**
11: $\texttt{DPL}[m_i] \leftarrow Sort_{T_{qp}}(\texttt{DPL}[m_i])$ ▷ Rank based on T_{qp}
12: **end for**
13: **return** (DPL);
14: **end function**

those with insufficient memory $\texttt{MEM}(m_i, \texttt{data}_{ui})$ and storage $\texttt{STOR}(m_i, \texttt{data}_{ui})$ capabilities (line 5). Then, it creates a list of tuples for each microservice m_i that associates a device d_j with the maximum data processing and queuing time of the upstage microservices (line 6). Finally, line 11 sorts the device preference lists of each microservice in descending order based on its data processing and queuing time $T_{qp}(m_i, \texttt{data}_{ui})$.

2) Device-side ranking: presented in Algorithm 3 receives as input the device preference lists $\texttt{DPL}[m_i]$ $(\forall m_i \in \mathcal{M})$ computed in Algorithm 2, along with the data processing workflow $\mathcal{W}$, the device set $\mathcal{D}$, and microservices $\mathcal{S}$ within a dependency level. Similarly, the algorithm initializes the empty microservice preference lists $\texttt{MPL}[d_j]$ for each device d_j in line 2. Afterward, each device in the preference list $\texttt{DPL}[m_i]$ of each microservice m_i creates in lines 3–7 a microservice preference list $\texttt{MPL}[d_j]$ associating the maximum data transmission time among all upstage microservices m_u (line 5). Finally, line 9 sorts the microservice preference lists

Algorithm 3 Device-side ranking.

Input: $\mathcal{W} = (\mathcal{M}, \mathcal{E}, \mathcal{M}_{\texttt{src}}, \mathcal{M}_{\texttt{snk}})$ ▷ Data processing workflow
$\mathcal{D} = \{d_j \mid 0 \leq j < \mathcal{N}_{\mathcal{D}}\}$ ▷ Device set
$\mathcal{S} \subset \mathcal{M}$ ▷ Microservice set in a dependency level
$\texttt{DPL}[m_i], \forall m_i \in \mathcal{S}$ ▷ Device preference lists of microservices in $\mathcal{S}$
Output: $\texttt{MPL}[d_j], \forall d_j \in \mathcal{D}$ ▷ Microservice preference lists of device set $\mathcal{D}$

1: **function** RANKMICROSERVICES($\mathcal{W}, \mathcal{D}, \mathcal{S}, \texttt{DPL}$)
2: $\texttt{MPL}[d_j] \leftarrow \emptyset, \forall d_j \in \mathcal{D}$ ▷ Initialize MPL
3: **for all** $m_i \in \mathcal{S}$ **do** ▷ Iterate microservices
4: **for all** $d_j \in \texttt{DPL}[m_i]$ **do** ▷ Iterate devices in DPL
5: $\texttt{MPL}[d_j] \leftarrow \texttt{MPL}[d_j] \bigcup \left(m_i, \max_{\forall(m_u, m_i, \texttt{data}_{ui}) \in \mathcal{E}} T_c(m_u, \texttt{data}_{ui}, m_i)\right)$ ▷ Add m_i and its T_c to MPL
6: **end for**
7: **end for**
8: **for all** $(d_j \in \mathcal{D}) \wedge (\texttt{MPL}[d_j] \neq \emptyset)$ **do**
9: $\texttt{MPL}[d_j] \leftarrow Sort_{T_c}(\texttt{MPL}[d_j])$ ▷ Rank based on T_c
10: **end for**
11: **return** (MPL);
12: **end function**

Algorithm 4 Capacity-aware matching.

Input: $\mathcal{W} = (\mathcal{M}, \mathcal{E}, \mathcal{M}_{\texttt{src}}, \mathcal{M}_{\texttt{snk}})$ ▷ Data processing workflow
$\mathcal{D} = \{d_j \mid 0 \leq j < \mathcal{N}_{\mathcal{D}}\}$ ▷ Device set
$\mathcal{S} \subset \mathcal{M}$ ▷ Microservice set in a dependency level
$\texttt{DPL}[m_i], \forall m_i \in \mathcal{S}$ ▷ Device preference lists of microservices in $\mathcal{S}$
$\texttt{MPL}[d_j], \forall d_j \in \mathcal{D}$ ▷ Microservice preference lists of device set $\mathcal{D}$
$\texttt{sched}(m_i), \forall m_i \in \mathcal{S}$ ▷ Scheduling function
Output: $\texttt{sched}(m_i), \forall m_i \in \mathcal{S}$ ▷ Updated scheduling function

1: **function** MATCH($\mathcal{W}, \mathcal{D}, \mathcal{S}, \texttt{MPL}, \texttt{DPL}, \texttt{sched}$)
2: $\texttt{alloc}[d_j] \leftarrow \emptyset, \forall d_j \in \mathcal{D}$
3: **for all** $(m_i \in \mathcal{S}) \wedge (\mathcal{S} \neq \emptyset)$ **do** ▷ Allocate all microservices in a level
4: $d_j \leftarrow \text{FIRST}(\texttt{DPL}[m_i])$
5: $c_j \leftarrow (\texttt{CPU}_j, \texttt{MEM}_j, \texttt{STOR}_j)$ ▷ Resources of device d_j
6: **if** $m_i \in \texttt{MPL}[d_j]$ **then** ▷ **State a): Under-utilization**
7: $\texttt{sched}(m_i) \leftarrow d_j$ ▷ Schedule m_i on d_j
8: $\texttt{alloc}[d_j] \leftarrow Sort_{T_c}(\texttt{alloc}[d_j] \cup m_i)$ ▷ Add m_i to alloc. list
9: $c_j \leftarrow c_j - \texttt{req}(m_i, \texttt{data}_{ui})$ ▷ Allocate device capacity
10: **if** $c_j < 0$ **then** ▷ **State b): Over-utilization**
11: $m_l \leftarrow \text{LAST}(\texttt{alloc}[d_j])$ ▷ Select m_l with highest transmission
12: $\texttt{sched}(m_l) \leftarrow \emptyset$ ▷ Unschedule m_l from d_j
13: $\texttt{alloc}[d_j] \leftarrow \texttt{alloc}[d_j] \setminus m_l$ ▷ De-allocate m_l
14: $c_j \leftarrow c_j + \texttt{req}(m_l, \texttt{data}_{l'l})$ ▷ Free device capacity
15: $\mathcal{S} \leftarrow \mathcal{S} \cup m_l$ ▷ Return m_l to unscheduled microservice set $\mathcal{S}$
16: **end if**
17: **if** $c_j \approx 0$ **then** ▷ **State c): Full-utilization**
18: $m_l \leftarrow \text{LAST}(\texttt{alloc}[d_j])$ ▷ Select m_l with highest transmission
19: **for all** $m_s \in \texttt{MPL}[d_j] \wedge \texttt{MPL}[d_j].\text{IDX}(m_l) < \texttt{MPL}[d_j].\text{IDX}(m_s)$ **do**
20: $\texttt{MPL}[d_j] \leftarrow \texttt{MPL}[d_j] \setminus m_s$ ▷ Remove higher transmission m_s
21: $\texttt{DPL}[m_s] \leftarrow \texttt{DPL}[m_s] \setminus d_j$ ▷ and corresponding device
22: $c_j \leftarrow c_j + \texttt{req}(m_s, \texttt{data}_{s's})$ ▷ Free device capacity
23: **if** $\texttt{DPL}[m_s] = \emptyset$ **then** ▷ Remove m_s if no preferences
24: $\mathcal{S} \leftarrow \mathcal{S} \setminus m_s$
25: **end if**
26: **end for**
27: **end if**
28: **end if**
29: **end for**
30: **return** (sched);
31: **end function**

in descending order based on the data transmission time T_c.

3) Capacity-aware matching: presented in Algorithm 4 matches the microservices of a dependency level on the continuum devices based on their mutual preference lists computed in Algorithms 2 and 3. The goal is to identify a schedule that maximizes the aggregate microservice-side utility of the data processing workflow and device-side utility of the resource provider, as modeled in Section III-G. After initializing an empty microservice allocation list for each device (line 2), the algorithm iterates over the microservice set $\mathcal{S}$ in a dependency level (line 3), identifies the highest ranked device for each

microservice, and retrieves its resources (lines 4–5). Then, the algorithm continues in one of the following three states based on the capacity c_j of a device d_j (line 5), according to the utilization model presented in Section III-B.

a) Under-utilization: (lines 7–9) temporarily matches the microservice to the preferred device (line 7), allocates its required resources (line 8), and updates the capacity c_j (line 9).

b) Over-utilization: (lines 10–15) occurs if a microservice m_i exceeds the capacity c_j of the device d_j (line 10). In this case, the algorithm frees resources for the microservice m_i by selecting the microservice m_l with the highest rank and transmission time in the allocation list $\texttt{alloc}\,[d_j]$ of the device d_j (line 11). Afterward, it rejects its existing match, removes it from the allocation list (lines 12–13), increases the capacity c_j (line 14), and returns it to the microservice set (line 15).

c) Full-utilization: (lines 17–27) occurs if a device d_j reaches its capacity c_j. The algorithm identifies the microservice m_l with the highest transmission time in the allocation list $\texttt{alloc}\,[d_j]$ of the device d_j (line 18). Afterward, d_j removes the microservices m_s with a higher transmission time than the temporarily-matched microservice m_l (line 19) from its preference list $\texttt{MPL}\,[d_j]$ (line 20). Similarly, the microservice m_s removes d_j from its device preference list $\texttt{DPL}[m_s]$ (line 21), which avoids scheduling high processing, queuing time, and data transmission time microservices on d_j. Furthermore, it allows scheduling higher-ranked microservices in $\texttt{MPL}\,[d_j]$ on the device d_j, gradually approaching a stable schedule by fixing the temporary matches (lines 19–22). If the microservice m_s has no devices in its preference list $\texttt{DPL}\,[m_s]$, the algorithm removes it from the microservice set $\mathcal{S}$ (lines 23–25).

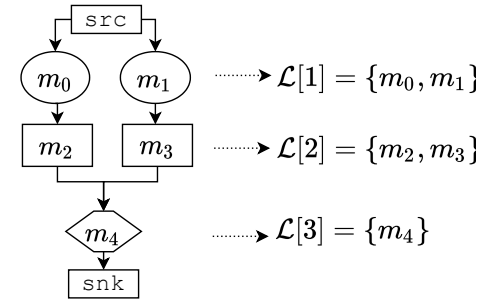
VI. C^3-Match Scheduling Example

Figure 2 illustrates a workflow with a set of five microservices $\mathcal{M} = \{m_0, m_1, m_2, m_3, m_4\}$ scheduled on four devices $\mathcal{D} = \{d_0, d_1, d_2, d_3\}$. Algorithm 1 creates three dependency levels in lines 3–6. Level $\mathcal{L}[1] = \{m_0, m_1\}$ includes the microservices closest to `src`, and level $\mathcal{L}[2] = \{m_2, m_3\}$ the microservice dependent on the first level and $\mathcal{L}[3] = \{m_4\}$ consists of the microservice closest to `snk`. Afterward, the algorithm iterates the three dependency levels (lines 7–15 and creates the device and microservice preference lists (displayed in brackets in Figure 2) by calling the Algorithms 2 and 3. Finally, it schedules the microservices by calling Algorithm 4.

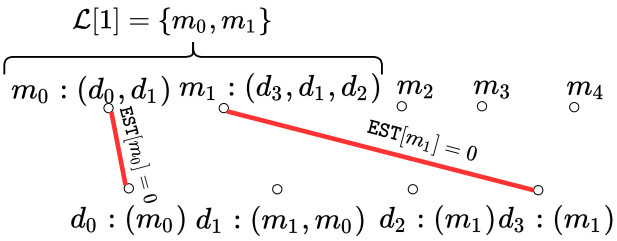
1) Level $\mathcal{L}[1] = \{m_0, m_1\}$: Figure 2b shows that d_0 and d_3 match the microservices m_0 and m_1 with the lowest data transmission times in their preference lists $\texttt{MPL}\,[d_0]$ and $\texttt{MPL}\,[d_3]$. Afterward, the microservices m_0 and m_1 start processing their first data element at time $\texttt{EST}\,[m_0] = \texttt{EST}\,[m_1] = 0$.

2) Level $\mathcal{L}[2] = \{m_2, m_3\}$: Figure 2c shows that the under-utilized devices d_0 and d_3 match their preferred microservices m_2, respectively m_3. Since the microservices m_2 and m_3 depend on the microservices m_0 and m_1, their earliest start times are $\texttt{EST}\,[m_2] = 2\text{s}$, respectively $\texttt{EST}\,[m_3] = 2.5\text{s}$.

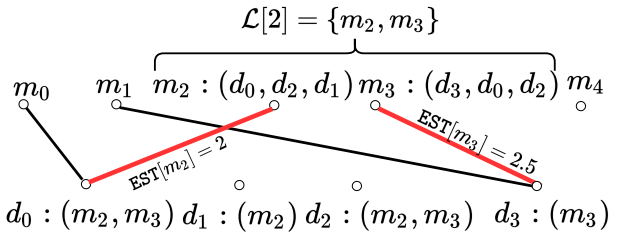
3) Level $\mathcal{L}[3] = \{m_4\}$: Figure 2d shows that d_3 provides the lowest data processing and queuing time for the microservice $m_4 \in \mathcal{L}[3]$. As the device d_3 is now in the over-utilized



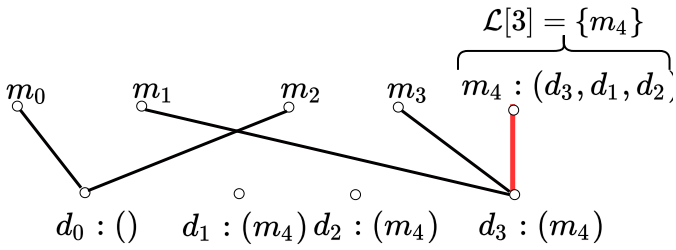
(a) Workflow with three dependency levels.



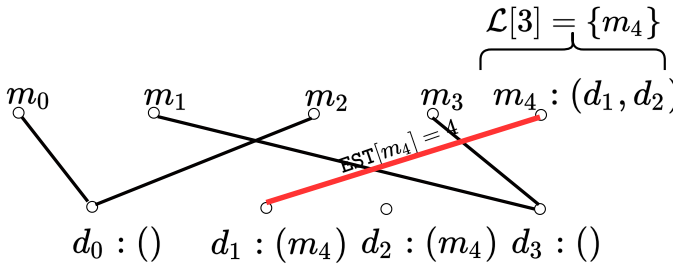
(b) m_0 matches to d_0; m_1 matches to d_3.



(c) m_2 matches to d_0; m_3 matches to d_3.



(d) m_4 matches to d_3.



(e) Over-utilized d_3 rejects m_4; m_4 matches to d_1.

Fig. 2: C^3-Match algorithm trace example.

state, it rejects the matching to m_4 and removes it from its preference list $\texttt{MPL}[d_3]$. Similarly, the microservice m_4 removes d_3 from its preference list $\texttt{DPL}[m_4]$. Subsequently, m_4 selects the device d_1 with the lowest data processing and queuing time from its preference list, as displayed Figure 2e. The device d_1 is now in the under-utilized state and schedules m_4 as the last microservice in the data processing workflow. Finally, since the microservice m_4 depends on the both microservices m_2 and m_3, its earliest start time is $\texttt{EST}[m_4] = 4\text{s}$.

VII. Implementation and Experimental Setup

We implemented the C^3-MATCH scheduling algorithm in `Python 3.9` using the matching library [26]. We integrated our customized scheduler in the `Kubernetes 1.21` orchestration tool using the `Python` client library 17.17 [27], published in the GitHub code repository [28]. We employed the `Prometheus Operator v0.45.0` to monitor the Kubernetes deployments, microservices, containers, and devices [17]. We used the `KubeMQ 2.2.10` message queue platform to implement the asynchronous data exchange between microservices [29]. The experimental validation reproducibility artifact, including the open-source code implementation, is available in the GitHub repository [30].

A. Carinthian computing continuum (C^3) testbed

We run experiments on the C^3 testbed [31], displayed in Table I, consisting of eleven heterogeneous instances types from two providers, distributed at four geographical locations across the Cloud, Fog, and Edge layers.

1) Cloud and Fog layers: comprise three on-demand instances from the Exoscale provider, hosted in the data center of the $A1$ network operator in Sofia (as the Cloud data center), Frankfurt, and Vienna (as Fog data centers) (blind review):

- `large` with four virtual cores and $8\,\text{GB}$ of memory;
- `medium` with two virtual cores and $4\,\text{GB}$ of memory;
- `small` with two virtual cores and $2\,\text{GB}$ of memory.

2) Edge layer: comprises private virtual instances and small devices at the University of Klagenfurt site interconnected by an HP Aruba layer-3 switch with 48 $1\,\text{Gbit}\,\text{s}^{-1}$ ports, $3.8\,\mu\text{s}$ latency and aggregate throughput of $104\,\text{Gbit}\,\text{s}^{-1}$:

- `large` with a twelve-core processor and $32\,\text{GB}$ memory;
- `medium` with an eight-core processor and $16\,\text{GB}$ memory;
- NVIDIA Jetson Nano (NJN) running `Linux for Tegra (L4T)` operating system;
- Raspberry Pi (RPi): 10 RPi3B+ and 30 RPi4 running `Raspberry Pi OS`.

We set an availability threshold $\theta = 33\%$ to indicate fully utilized resources (see Section III-B), which keeps one-third of free capacity for consolidation and management tasks [32].

B. Evaluation metrics

We evaluated the performance of C^3-MATCH for scheduling two data processing workflows on the simulated and real-world infrastructure testbed using four metrics.

1) Completion time: $\texttt{CT}(\mathcal{W}, \mathcal{D})$ defined in Section III-F3.

2) Data processing and queuing time: $T_{qp}(m_i, \texttt{data}_{ui})$ defined in Section III-G2.

3) Data transmission time: $T_c(m_u, \texttt{data}_{ui}, m_i)$ defined in Section III-G4.

4) Element queuing time: $t_q(m_u, m_i)$ in a queue $\texttt{qu}_{ui}$ is the ratio between the average queue size and element arrival rate λ_{ui} following the Poisson distribution [22]:

$$t_q(m_u, m_i) = \frac{\texttt{quSize}_{ui}}{\lambda_{ui}}; \quad \texttt{quSize}_{ui} = \frac{\lambda_{ui}^2}{\tau_{ui} \cdot (\tau_{ui} - \lambda_{ui})},$$

assuming that the queue size and the number of data elements processed by m_i reach a steady state, following the $M/M/1$ and the Little's queuing model [23].

C. Related work comparison

We selected three state-of-the-art methods tailored for supporting microservice placement or microservice scheduling on the computing continuum. We compared C^3-MATCH against each related method $\mathcal{R}$ using its *relative improvement* on the metrics $\mathcal{S}$ defined in Section VII-B: $\frac{\mathcal{S}_{\mathcal{R}} - \mathcal{S}_{C^3\text{-MATCH}}}{\mathcal{S}_{\mathcal{R}}}$.

1) CODA*:* [9] applies matching theory to minimize the completion time of workflows, alongside the residual bandwidth over the network channels. CODA assigns microservices to devices with fixed capacity, ignoring the earliest start and finish times along with synchronization phases.

2) SEA-LEAP: [15] implements a heuristic algorithm to perform workflow and data movement as a placement control mechanism. SEA-LEAP minimizes the data transmission time by identifying the Fog instances with sufficient compute capacity closest to the data producers.

3) KCSS: [10] is the Kubernetes container scheduler of microservices on Cloud or Fog data centers using the Topsis algorithm [33] based on multiple criteria such as workflow completion time, the number and size of Docker container images, and the number of running containers on each device.

D. Case study workflows

1) Road sign inspection: is a traffic sign recognition workflow with seven microservices following road safety inspection concerns, depicted in Figure 3. The application recognizes defects (e.g., low reflection or high inclination) and damages (e.g., broken surfaces) on road signs from multiple camera data flows in moving vehicles.

a) Encode: the raw video in multiple bitrate and resolution pairs near the vehicles equipped with multi-view cameras using `ffmpeg` [34], [35] with the `H.264` video codec;

b) Frame: the encoded video using `OpenCV` to produce still images for traffic road signs recognition;

c) Train: a multi-class model using $50\,000$ still frames extracted from the input video for classification of traffic signs with an accuracy of at least 70% [36].

d) Inference: with the trained model for sign recognition;

e) Package and delivery: of detected signs in a format required for validation by the traffic management authorities;

f) Dataset storage: of the records of the road signs for inventory purposes.

TABLE I: Carinthian computing continuum (C^3) testbed.

Layer	*Cloud*			*Fog*			*Edge*				
Location	Sofia			Frankfurt, Vienna			Klagenfurt				
Instance type	Exoscale `large`	Exoscale `medium`	Exoscale `small`	Exoscale `large`	Exoscale `medium`	Exoscale `small`	Klagenfurt `large`	Klagenfurt `medium`	*NJN*	*RPi4*	*RPi3*
Number	20	20	20	40	40	40	40	40	5	30	10
`CPU` cores	4	2	2	4	2	2	12	8	4	4	4
`CPU` (MIPS)	14 000	7200	7100	14 000	7200	7100	58 000	21 700	4080	5100	3500
`MEM` (GB)	8	4	2	8	4	2	32	16	4	4	1
`STOR` (GB)	10	10	10	10	10	10	32	32	16	16	16
`BW` (Mbit s^{-1})	70	70	70	100–220	100–220	100–220	940	940	840	800	328
`LAT` (ms)	30	30	30	7–18	7–18	7–18	1–2	1–2	1–2	1–2	1–2

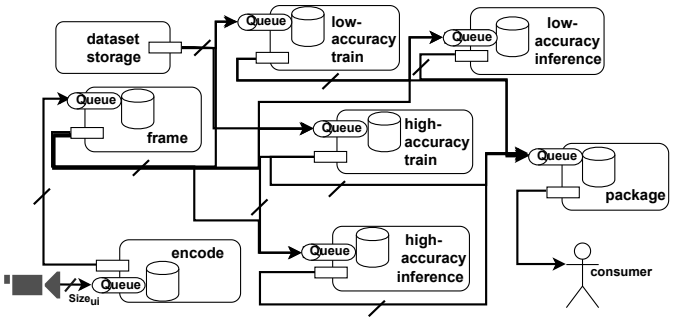


Fig. 3: Road sign inspection.

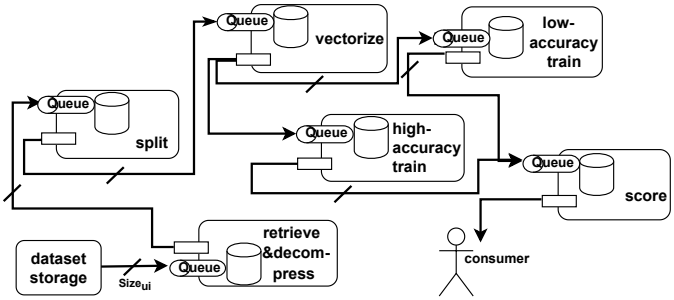


Fig. 4: Sentiment analysis of Amazon reviews.

2) Sentiment analysis: is a workflow with six microservices depicted in Figure 4, classifying Amazon customer reviews into positive or negative classes. The workflow comprises data transformation, feature extraction (word tokenizing, vectorizing), and ML model training operations [37], [38]. We used the `scikit-learn` framework and extended the large-scale text classification to support sentiment analysis.

a) Retrieve and decompress: 3.2 million Amazon customer reviews of 1.6 GB in size;

b) Split: the data in sequences of $500 - 10^5$ elements (reviews) to fit into devices' memory;

c) Vectorize: the raw data to a matrix with classified features by a vocabulary-based vectorizer;

d) Logistic regression training: of an ML-based model on Amazon reviews with the accuracy range of $69 - 87\%$;

e) Score: the test data and calculate the probability of its alignment with the set of annotated reviews.

We identified the resource requirements of each microservice based on the average resources and network utilization in terms of `CPU` $(m_i, \texttt{data}_{ui})$ (in MI), `MEM` $(m_i, \texttt{data}_{ui})$ (in MB), `STOR` $(m_i, \texttt{data}_{ui})$ (in GB), $\texttt{data}_{ui}$ (in MB), and arrival rate (in s^{-1}), presented in Table II.

TABLE II: Workflow resource requirements.

	`CPU` [MI] $\cdot 10^3$	`MEM` [GB]	`STOR` [GB]	$\texttt{data}_{ui}[x]$ [MB]	λ_{ui} [s^{-1}]
Road sign inspection	$1 - 3500$	$0.1 - 2$	$0.1 - 1$	$0.4 - 3.5$	$0.1 - 1$
Sentiment analysis	$50 - 5000$	$0.1 - 2$	$0.1 - 2$	$3 - 5$	$0.1 - 1$

TABLE III: Simulation experimental design.

Experiment	*Producer* (`src`)	*Microservice*	$\texttt{Size}_{ui}$	$\lambda_{ui}[s^{-1}]$
Processing load	35 096, 58 660, 87 239, 107 790	228 124, 381 290, 567 054, 700 635	10	0.1 – 1
Data flow load	35 096	195 376	10, 20 100, 200	0.1 – 1

VIII. Simulation-based Evaluation

We evaluated the C^3-MATCH results by employing `YAFS` simulator [39], running Ubuntu 18.04 LTS on an `Intel® Core`$^{(TM)}$ `i7-8650U` processor, and 16 GB of memory.

A. Infrastructure simulation

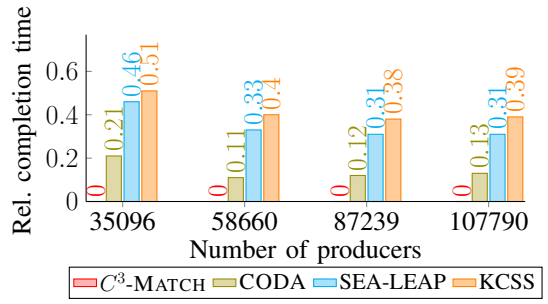
We simulated the C^3 testbed described in Section VII-A as a bidirectional graph using `NetworkX Python` package with 300 Cloud, Fog and Edge instances. We configured the network bandwidth and latency by measuring maximum achievable throughput with `iPerf3` tool and round-trip time with `ICMP` echo request and reply (see Table I).

B. Workflow simulation

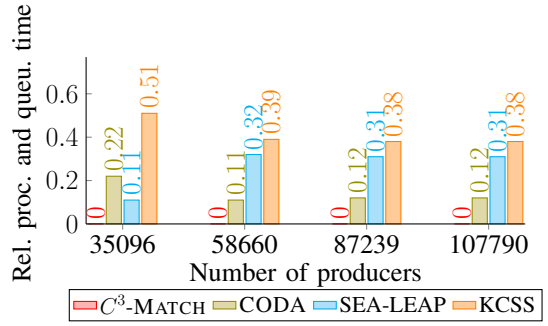
We simulated the road sign inspection and sentiment analysis workflows using the `Gn_graph` library of the `NetworkX` package. We set the simulated microservice requirements based on the ranges presented in Table II and generated data elements between microservices with different inter-arrival times, following uniform distribution [40].

C. Processing load analysis

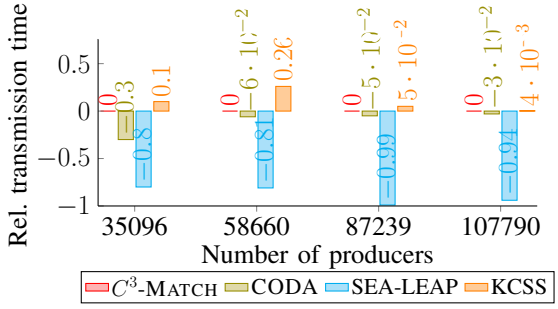
1) Simulation experiment: We simulated a different number of producers (half for each case study), generating data processed by a separate workflow replica (see Table III). We dynamically generated the number of producers based on a random element arrival rate selected for each replica in the $0.1 - 1s^{-1}$ range within 2000 s simulation time [40]. We fixed



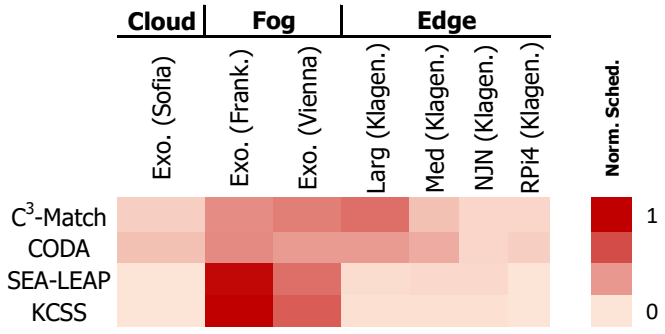
(a) Relative workflow completion time for different producers.



(b) Relative processing and queuing time for different producers.



(c) Relative data transmission time for different producers.



(d) Normalized number of microservices on continuum instances.

Fig. 5: Processing load simulation results.

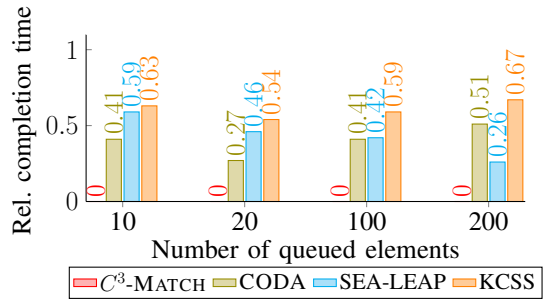
the data flow size to ten elements, the smallest data flow size supported by the simulated communication protocol [41].

2) Results: Figure 5a shows that C^3-MATCH reduces the completion time by $11-21\%$, $31-46\%$, respectively $38-51\%$ compared to CODA, SEA-LEAP, and KCSS by selecting underutilized Edge and Fog instances with lower data processing and transmission times. Further analysis in Figure 5b shows that C^3-MATCH reduces the average data processing and queuing time by selecting a range of instances distributed across the Cloud, Fog, and Edge layers. Although C^3-MATCH's advantage decreases with the number of producers, it stays above 30% in processing and queuing time. Figure 5c shows that C^3-MATCH fails to improve the data transmission time compared to SEA-LEAP, which collocates the microservices. This disadvantage is from the order of $10\,\mathrm{ms}$ and is negligible compared to the processing and queuing times. As the number of producers increases, the data flow gradually saturates the network channels, and the transmission time decreases for all methods. Figure 5d shows that C^3-MATCH schedules the microservices mainly on the Fog and Edge instances. The traffic-aware CODA balances the load across the continuum instances, while the locality-aware SEA-LEAP and KCSS select specific clusters of Fog or Edge devices.

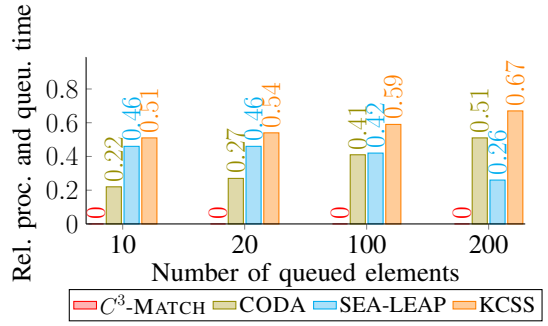
D. Data flow analysis

1) Simulation experiment: We evaluated different sequences of $\texttt{Size}_{ui}$ element arrival rates λ_{ui} with a fixed number of producers (fewest workflow replicas), displayed in Table III. We set the element arrival rate to $1\,\mathrm{s}^{-1}$, which overloads the computing instances and creates the element queue within the average workflow completion. We bound the data flow size to 200 elements, as more elements overload each computing instance, increase the workflow completion time, and require a replica deployment for each microservice.

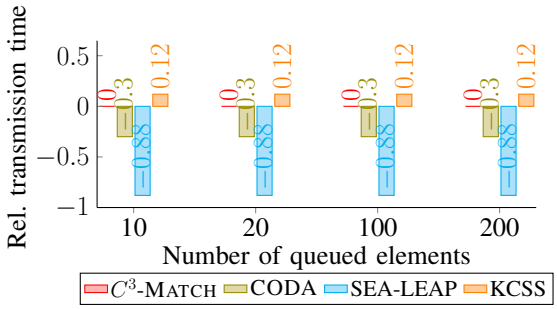
2) Results: Figure 6a shows that C^3-MATCH reduces the completion time by $27-51\%$, $26-59\%$, respectively $54-67\%$ compared to CODA, SEA-LEAP, and KCSS by scheduling the microservices on multiple Fog or Edge instances such as Klagenfurt-`medium` or Exoscale in Vienna. The processing and queuing time show similar improvements (see Figure 6b). Again, SEA-LEAP collocates the microservices on Exoscale instances that reduces the data transmission (see Figure 6c). CODA selects the Klagenfurt-`large` and Exoscale-`large` instances in Vienna for the road sign inspection, and the Klagenfurt-`medium` along with Exoscale-`medium` in Frankfurt for sentiment analysis workflow. CODA improves the processing and queuing time up to $\texttt{Size}_{ui} = 100$ data flow elements compared to SEA-LEAP and KCSS (see Figure 6b), but fails for the largest data flow size of 200. Figure 6c shows that C^3-MATCH reduces the data transmission time by 12% by selecting the Edge and Exoscale instances in Vienna and Frankfurt. KCSS chooses Fog instances without collocating microservices on the Edge to reduce the data traffic. While C^3-MATCH does not improve the data transmission time compared to the traffic- and latency-aware CODA and SEA-LEAP methods, the increasing data flow does not saturate the network channels and does not affect the transmission time. Figure 6d shows that C^3-MATCH reduces the element queuing time for the last (200^{th}) data flow element by considering the arrival rate along with the devices' processing rate. The training microservices of both workflows have the highest element queuing times because of the lower processing rate compared to the arrival rate, enlarging the element queue.



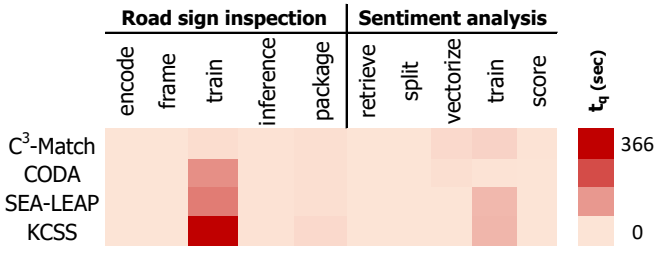
(a) Relative completion time for different queued elements.



(b) Relative processing and queuing time for different queued elements.



(c) Relative data transmission time for different queued elements.



(d) 200$^{\text{th}}$ element queuing time by different microservices.

Fig. 6: Data flow simulation results.

IX. Real Testbed Evaluation

We perform a real-world evaluation of the C^3-MATCH method using the two case study workflows presented in Section VII-D. The experiments aim is to validate the simulation scenarios presented in Sections VIII-C and VIII-D through real executions on the C^3 testbed introduced in Section VII-A.

TABLE IV: Real-world testbed experimental design.

Experiment	*Producer* (`src`)	*Microservice*	$\texttt{Size}_{ui}$	$\lambda_{ui}[\text{s}^{-1}]$
Processing load	10, 20, 30, 40	130, 260, 390, 520	10	$0.1-1$
Data flow load	2	13	10, 20, 100, 200	$0.1-1$

A. Processing load analysis

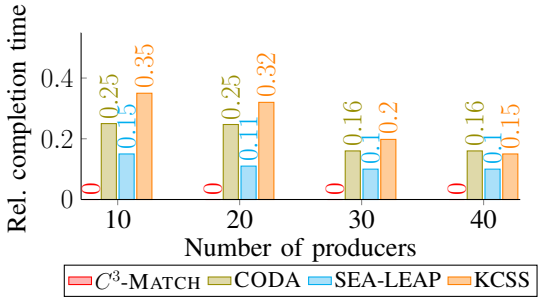
1) Experimental design: We evaluated various producers requiring various data processing workflows, following the uniform distribution for the inter-arrival time between two subsequent data elements. Table IV shows the number of microservices deployed for all producers.

2) Results: Figure 7a shows that C^3-MATCH improves the completion time by $10-35\%$ compared to the three related works by concurrently scheduling the microservices corresponding to the same dependency levels of all the workflows on the continuum. Figure 7b shows that KCSS performs worst, as it deploys the microservices on the Fog instances in the same network cluster and ignores the Klagenfurt-`large` or Klagenfurt-`medium` instances. While CODA schedules each workflow separately, the dependency-aware C^3-MATCH considers independent microservices from multiple workflows and their deployment influence on each other, which further reduces their average completion time. Lastly, SEA-LEAP fails to properly analyze the data flow dependencies and collocates multiple microservices on a single Fog or Edge instance, increasing the processing and queuing time. Although C^3-MATCH distributes the execution of microservices across the continuum, it does not improve the data transmission time compared to the traffic-aware CODA and latency-aware SEA-LEAP (see Figure 7c). While CODA collocates the workflow microservices, SEA-LEAP selects a specific Fog or Edge instance for each workflow, and KCSS a single cluster of Cloud or Fog instances. However, C^3-MATCH distributes workflow over the Cloud, Fog, and Edge layers and selects a wide range of continuum instances. Moreover, increasing the number of producers does not impact data transmission. Figure 7d shows that C^3-MATCH primary schedules the microservices on the Fog and Edge instances and rarely deploys the workflows on the Cloud. While CODA selects a variety of Cloud, Fog, and Edge instances, SEA-LEAP schedules the workflows on the Fog and Edge instances to localize the execution of the data-dependent analytics. Finally, KCSS selects the Fog instances inside one Kubernetes cluster.

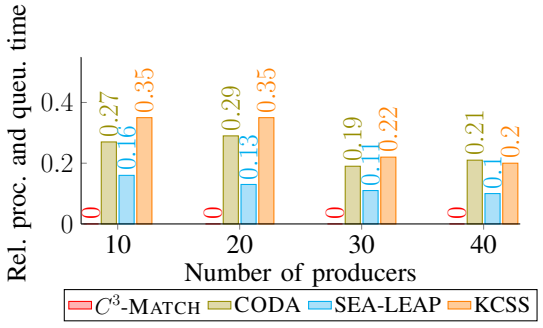
B. Data flow analysis

1) Experimental design: evaluates the scheduling results for both workflows by modifying the element arrival rates $10^{-1}-1\text{s}^{-1}$ [42], leading to different element queuing times. Each producer generates $10-200$ elements (see Table IV).

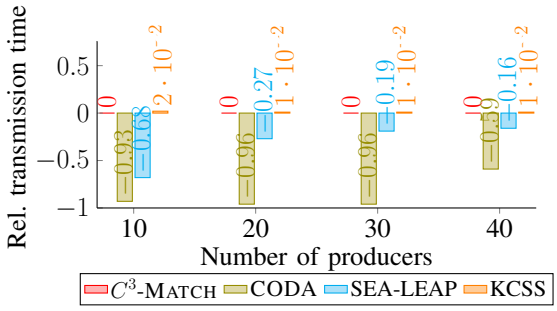
a) Results: Figure 8a shows that the average completion time increases with the queuing of the 200 elements generated by one producer. C^3-MATCH reduces the workflow completion time by $1-18\%$, $4-20\%$, and $4-21\%$ compared to CODA,



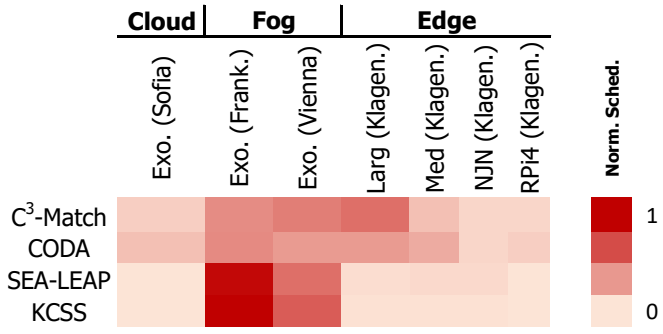
(a) Relative completion time for different producers.



(b) Relative processing and queuing time for different producers.

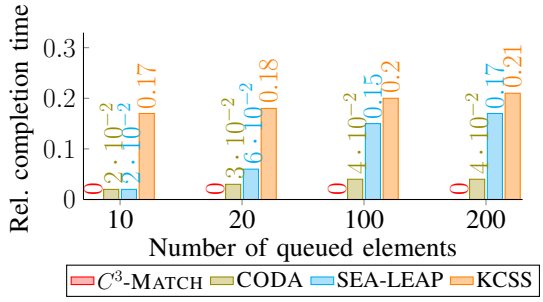


(c) Relative data transmission time for different producers.

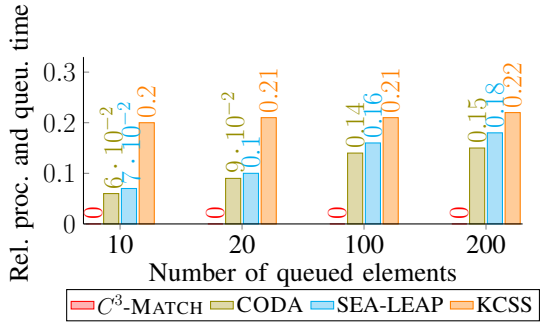


(d) Normalized number of microservices on continuum instances.

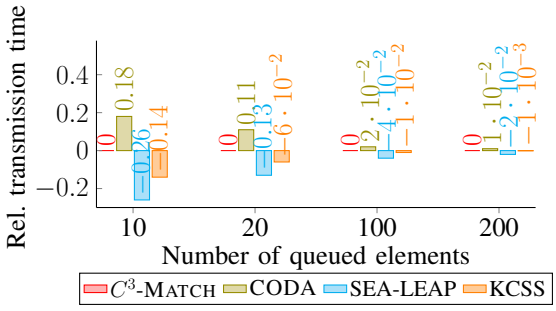
Fig. 7: Processing load results in a real testbed.



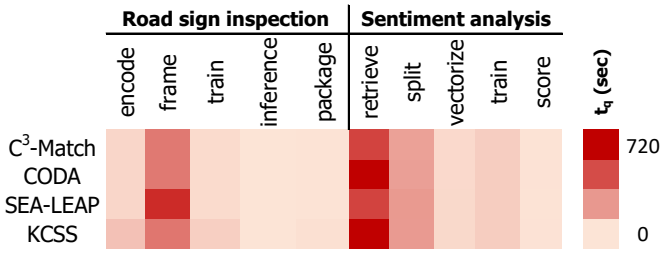
(a) Relative completion time for different queued elements.



(b) Relative processing and queuing time for different queued elements.



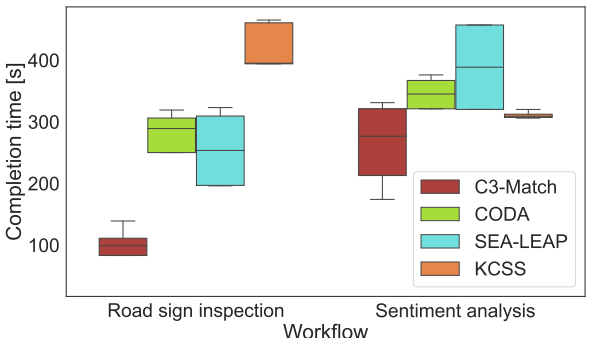
(c) Relative data transmission time for different queued elements.



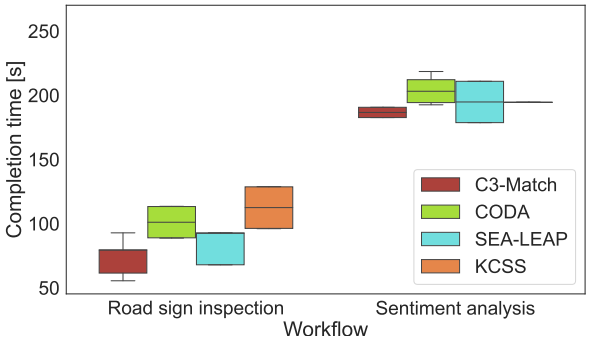
(d) 200$^{\text{th}}$ element queuing time by different microservices.

Fig. 8: Data flow results in a real testbed.

SEA-LEAP, and KCSS for the different numbers of data elements in the queue. Unlike the related methods, C^3-MATCH considers the element queuing time, which avoids selecting devices with limited network bandwidth and processing capacities. Figure 8b confirms that C^3-MATCH reduces the average processing and queuing time using asynchronous communication among the microservices. In contrast to SEA-LEAP, C^3-MATCH does not impose a higher processing load by the microservices' collocation, especially for a large sequence of 200 data elements (see Figures 8b). Moreover, Figure 8c further shows that C^3-MATCH improves the data transmission time compared to KCSS, which selects the farther instances of a single network cluster. However, unlike SEA-LEAP, it does not reduce the data transmission time as SEA-LEAP localizes the data flow-dependent microservices on the same instances (see Figure 8c). This disadvantage is from the order of 1000 ms



(a) Simulated testbed.



(b) Real testbed.

Fig. 9: Completion time in processing load experiments.

and is negligible compared to the processing and queuing times. Although C^3-MATCH does not reduce the data transmission time, it decreases the completion time by not imposing a higher processing load with the microservices' collocation, especially for an extended sequence of 200 data elements (see Figure 8a). C^3-MATCH reduces the queuing time compared to other methods by considering the element queuing time in selecting the computing instances with appropriate network bandwidth and processing capacities (see Figure 8d).

C. Result validation

We discuss in this section the alignment of the large-scale simulation comprising 107 790 workflows with the real smaller-scale testbed running 40 workflows.

1) Completion time skewness: Figure 9 demonstrates that C^3-MATCH has a lower median completion time compared to the other methods in both simulation and real testbed experiments. The completion time of the road sign inspection workflow is lower than the sentiment analysis due to the different microservices processing and transmission requirements. Finally, the workflow completion time has a higher deviation in the simulated experiments compared to the real testbed because of the higher number of simulated replicas.

2) Experimental result correlation: We use the Pearson's correlation coefficient to measure the relative variation between the simulated and real workflow completion times. The C^3-MATCH, CODA, and SEA-LEAP reached a high correlation between both types of experiments up to 80%. The KCSS's correlation of 60% is because of the identical schedules and small completion time ranges.

X. CONCLUSION AND FUTURE WORK

We introduced C^3-MATCH, a matching theory-based capacity-aware algorithm that considers asynchronous data processing, queuing, and transmission times for scheduling microservices of the workflows on the computing continuum. C^3-MATCH identifies parallelism by grouping independent microservices in dependency levels and applies an iterative matching game algorithm to each level by involving two sets of players. Firstly, the microservices of every level rank the devices based on their data processing and queuing times. Secondly, the devices rank the microservices of the level based on their data transmission times. Afterward, C^3-MATCH analyzes the preference lists and schedules each microservice on one of its ranked devices based on its availability, aiming to maximize the aggregate utility of the workflow and resource providers. Finally, it estimates each microservice's earliest start time and finish time, which helps to reserve and release the assigned devices. We evaluated C^3-MATCH on the road sign inspection and sentiment analysis case studies using scalable simulation experiments varying the processing and data flow loads. The evaluation results showed that C^3-MATCH achieved $1 - 67\%$ lower workflow completion time compared to three related methods thanks to its asynchronous communication and selection of the computing instances with appropriate network bandwidth and processing capacities. We validated the simulation using smaller-scale real testbed experiments showing a good correlation result.

The experimental validation reproducibility artifact, including the open-source code implementation, is available in the GitHub repository [30]. We plan to explore the data locality-based approach by re-routing the producer's data flow to the appropriate computing instance in future work.

ACKNOWLEDGEMENT

This work received funding from the European Commission's Horizon 2020 program (grant 101016835, DataCloud) and Austrian Research Promotion Agency (FFG) (grant 888098, Kärntner Fog).

REFERENCES

[1] Yue Zhou, Yue Yu, and Bo Ding. Towards mlops: A case study of ml pipeline platform. In *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, pages 494–500. IEEE, 2020.

[2] Sunil Singh Samant, Mohan Baruwal Chhetri, Quoc Bao Vo, Ryszard Kowalczyk, and Surya Nepal. Towards end-to-end qos and cost-aware resource scaling in cloud-based iot data processing pipelines. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 287–290. IEEE, 2018.

[3] Dragi Kimovski, Narges Mehran, Christopher Emanuel Kerth, and Radu Prodan. Mobility-aware iot applications placement in the cloud edge continuum. *IEEE Transactions on Services Computing*, 2021.

[4] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

[5] Ahmed Ali-Eldin, Bin Wang, and Prashant Shenoy. The hidden cost of the edge: a performance comparison of edge and cloud latencies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2021.

[6] Ali J Fahs and Guillaume Pierre. Tail-latency-aware fog application replica placement. In *International Conference on Service-Oriented Computing*, pages 508–524. Springer, 2020.

[7] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[8] Mutaz Barika, Saurabh Garg, Albert Y Zomaya, Lizhe Wang, Aad Van Moorsel, and Rajiv Ranjan. Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

[9] Narges Mehran, Dragi Kimovski, and Radu Prodan. A two-sided matching model for data stream processing in the cloud – fog continuum. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 514–524. IEEE, 2021.

[10] Tarek Menouer. Kcss: Kubernetes container scheduling strategy. *The Journal of Supercomputing*, 77(5):4267–4293, 2021.

[11] Sijie Wu, Hanhua Chen, Yonghui Wang, and Hai Jin. Argus: Efficient job scheduling in rdma-assisted big data processing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 827–836, 2021.

[12] Sherif Abdelwahab, Sophia Zhang, Ashley Greenacre, Kai Ovesen, Kevin Bergman, and Bechir Hamdaoui. When clones flock near the fog. *IEEE Internet of Things Journal*, 5(3):1914–1923, 2018.

[13] Harshit Gupta, Tyler C Landle, and Umakishore Ramachandran. epulsar: Control plane for publish-subscribe systems on geo-distributed edge infrastructure. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–241. IEEE, 2021.

[14] Thomas Pusztai, Fabiana Rossi, and Schahram Dustdar. Pogonip: Scheduling asynchronous applications on the edge. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 660–670, 2021.

[15] Ivan Lujic, Vincenzo De Maio, Srikumar Venugopal, and Ivona Brandic. Sea-leap: Self-adaptive and locality-aware edge analytics placement. *IEEE Transactions on Services Computing*, 2021.

[16] Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. Model-based stream processing auto-scaling in geo-distributed environments. In *ICCCN 2021-30th International Conference on Computer Communications and Networks*, 2021.

[17] Mulugeta Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In *ICCCN 2021-30th International Conference on Computer Communications and Networks*, 2021.

[18] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[19] Nikolay Nikolov, Yared Dejene Dessalk, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H Payberah, and Dumitru Roman. Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers. *Internet of Things*, page 100440, 2021.

[20] Omar Aaziz, Jonathan Cook, and Hadi Sharifi. Push me pull you: Integrating opposing data transport modes for efficient hpc application monitoring. In *2015 IEEE International Conference on Cluster Computing*, pages 674–681, 2015.

[21] Alexandre da Silva Veith, Marcos Dias de Assunçao, and Laurent Lefèvre. Latency-aware placement of data stream analytics on edge computing. In *International Conference on Service-Oriented Computing*, pages 215–229. Springer, 2018.

[22] John DC Little et al. A proof for the queuing formula. *Operations Research*, 9(3):383–387, 1961.

[23] Gunter Bolch, Stefan Greiner, Hermann De Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[24] Sarder Fakhrul Abedin, Md Golam Rabiul Alam, SM Ahsan Kazmi, Nguyen H Tran, Dusit Niyato, and Choong Seon Hong. Resource allocation for ultra-reliable and enhanced mobile broadband iot applications in fog network. *IEEE Transactions on Communications*, 67(1):489–502, 2018.

[25] Nafiseh Sharghivand, Farnaz Derakhshan, Lena Mashayekhy, and Leyli Mohammad Khanli. An edge computing matching framework with guaranteed quality of service. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[26] Henry Wilde, Vincent Knight, and Jonathan Gillard. Matching: A python library for solving matching games. *Journal of Open Source Software*, 5(48):2169, 2020.

[27] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. Context-aware kubernetes scheduler for edge-native applications on 5g. *Journal of Communications Software and Systems*, 16(1):85–94, 2020.

[28] C^3-MATCH code repository. https://github.com/SiNa88/C3-Match. [Online; accessed July-2022].

[29] Yared Dejene Dessalk, Nikolay Nikolov, Mihhail Matskin, Ahmet Soylu, and Dumitru Roman. Scalable execution of big data workflows using software containers. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, pages 76–83, 2020.

[30] C^3-MATCH reproducibility artifact. https://github.com/SiNa88/C3-Match/blob/main/describeArtifact.pdf. [Online; accessed July-2022].

[31] Dragi Kimovski, Roland Mathá, Josef Hammer, Narges Mehran, Hermann Hellwagner, and Radu Prodan. Cloud, fog or edge: Where to compute? *IEEE Internet Computing*, 2021.

[32] Mahammad Shareef Mekala and Perumal Viswanathan. Energy-efficient virtual machine selection based on resource ranking and utilization factor approach in cloud computing for iot. *Computers & Electrical Engineering*, 73:227–244, 2019.

[33] James Fallon. Topsis python - gitlab. https://gitlab.com/jamesfallon/topsis-python/-/tree/master/. [Online; accessed July-2022].

[34] Anatoliy Zabrovskiy, Christian Feldmann, and Christian Timmerer. Multi-codec dash dataset. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 438–443. ACM, 2018.

[35] Roland Mathá, Dragi Kimovski, Anatoliy Zabrovskiy, Christian Timmerer, and Radu Prodan. Where to encode: A performance analysis of x86 and arm-based amazon ec2 instances. In *2021 IEEE 17th International Conference on eScience (eScience)*, pages 118–127. IEEE, 2021.

[36] Peng Liu, Bozhao Qi, and Suman Banerjee. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st international workshop on edge systems, analytics and networking*, pages 1–6, 2018.

[37] Tanjim Ul Haque, Nudrat Nawal Saber, and Faisal Muhammad Shah. Sentiment analysis on large scale amazon product reviews. In *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pages 1–6, 2018.

[38] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Markus Weimer, and Matteo Interlandi. From the edge to the cloud: Model serving in ml. net. *IEEE Data Eng. Bull.*, 41(4):46–53, 2018.

[39] I. Lera, C. Guerrero, and C. Juiz. Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, 7:91745–91758, 2019.

[40] Isaac Lera, Carlos Guerrero, and Carlos Juiz. Availability-aware service placement policy in fog computing based on graph partitions. *IEEE Internet of Things Journal*, 6(2):3641–3651, 2019.

[41] Zahra Najafabadi Samani, Nishant Saurabh, and Radu Prodan. Multilayer resource-aware partitioning for fog application placement. In *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, pages 9–18. IEEE, 2021.

[42] Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. Waiting game: optimally provisioning fixed resources for cloud-enabled schedulers. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.