The IEEE International Conference on Cluster Computing (CLUSTER 2025)


SoCL: Scalable and Latency-Optimized Microservices in Serverless Edge Computing

Shuaibing Lu, **Bojin Xiang**, Ziyu You
College of Computer Science
Beijing University of Technology
Beijing, China

Jie Wu
China Telecom Cloud Computing Research Institute
Center for Networked Computing
Temple University
Philadelphia, USA

Wentong Cai
College of Computing and Data Science
Nanyang Technological University
Singapore

1 Introduction

2 Model and Formulation

3 Algorithm Design

4 Experiment and Results

Emerging serverless edge computing

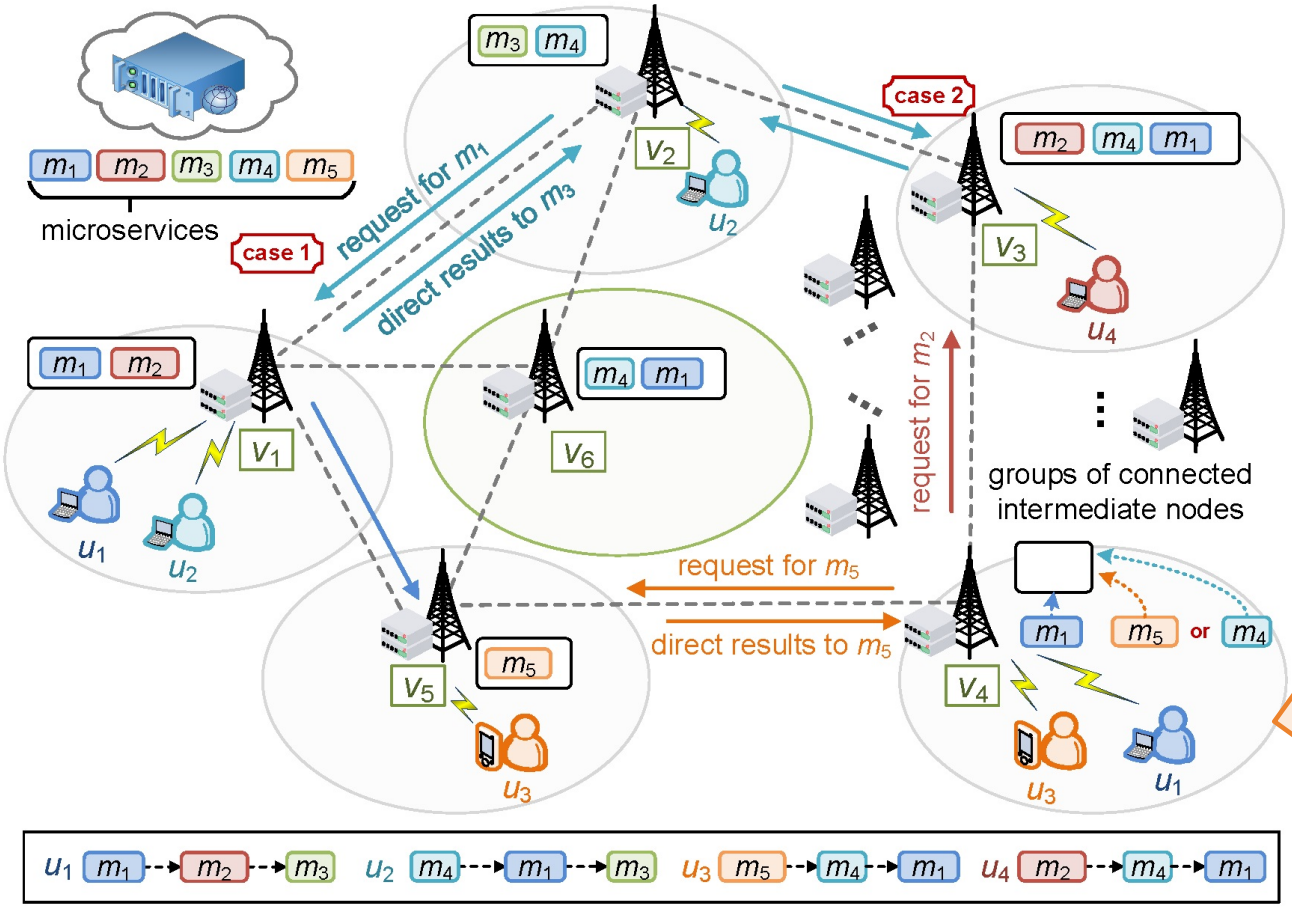
- Serverless edge computing deploys lightweight functions to the network edge.
- User mobility causes frequent and unpredictable shifts in request patterns.
- Random request dependency leads to path conflicts and network contention.

Drone Control [dji.com]

Automated Driving [xiaomiev.com]

Mobile Applications [huawei.com]

An example - scenario



Edge: constrained resources

- deployed near base station;
- accommodate limited microservices;

User: diverse requests

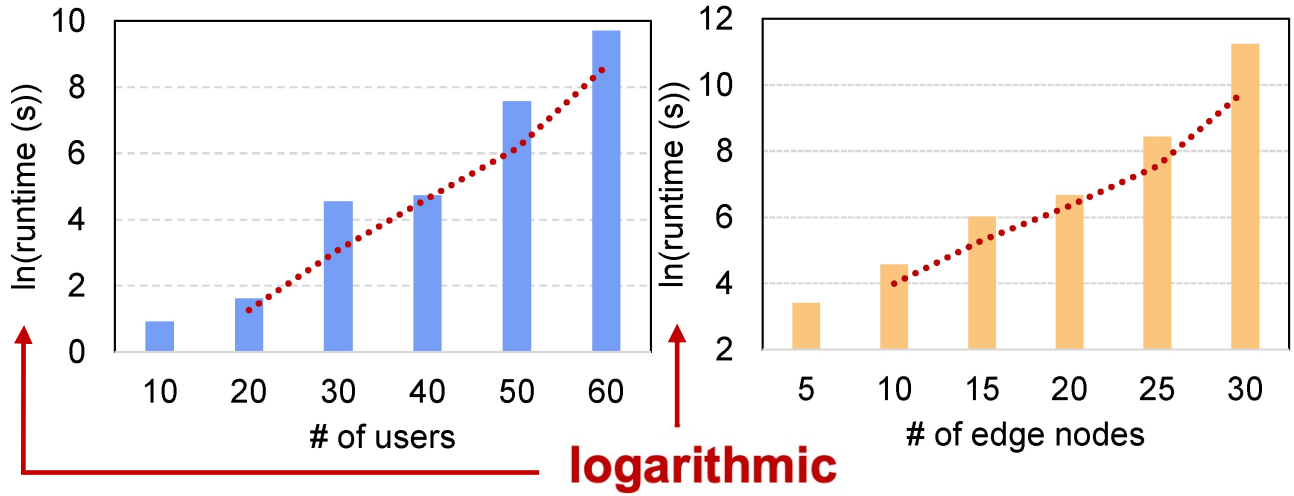
- high mobility;
- random request dependency;

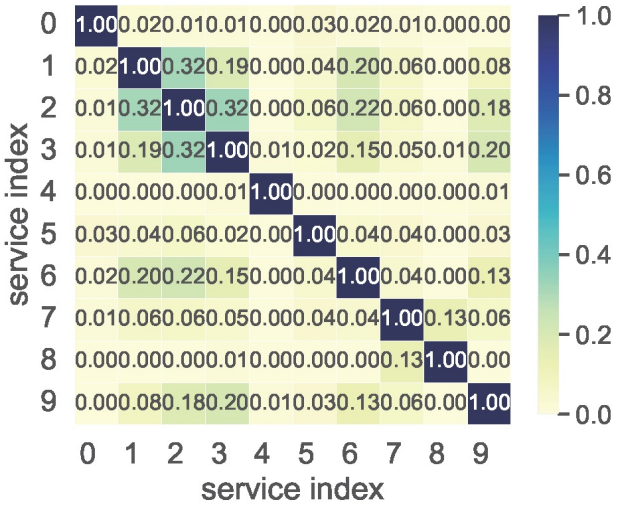
back-and-forth data transmission

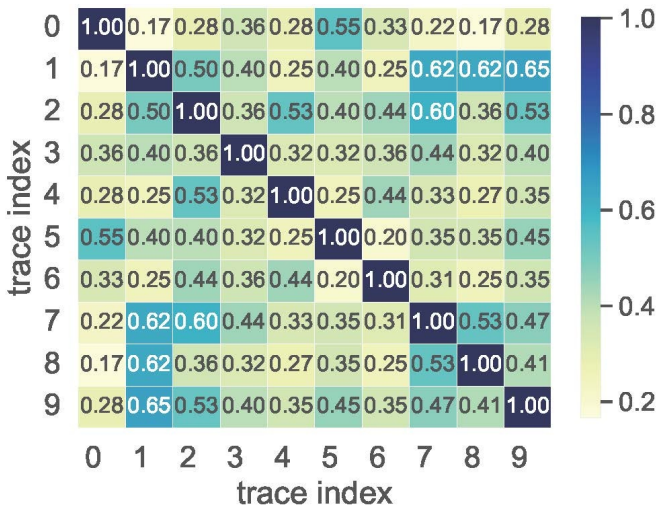
Challenges

High complexity: high dimensional
a Gurobi-based decisioning prototype.

Runtime increased exponentially as the number of users / of edge nodes grew.







Dynamic environment: time-varying
analysis of Alibaba Cluster traces
select top-10 frequently recorded services.

Similarity varies across different services and dependency chains, indicating a highly dynamic and heterogeneous service landscape.

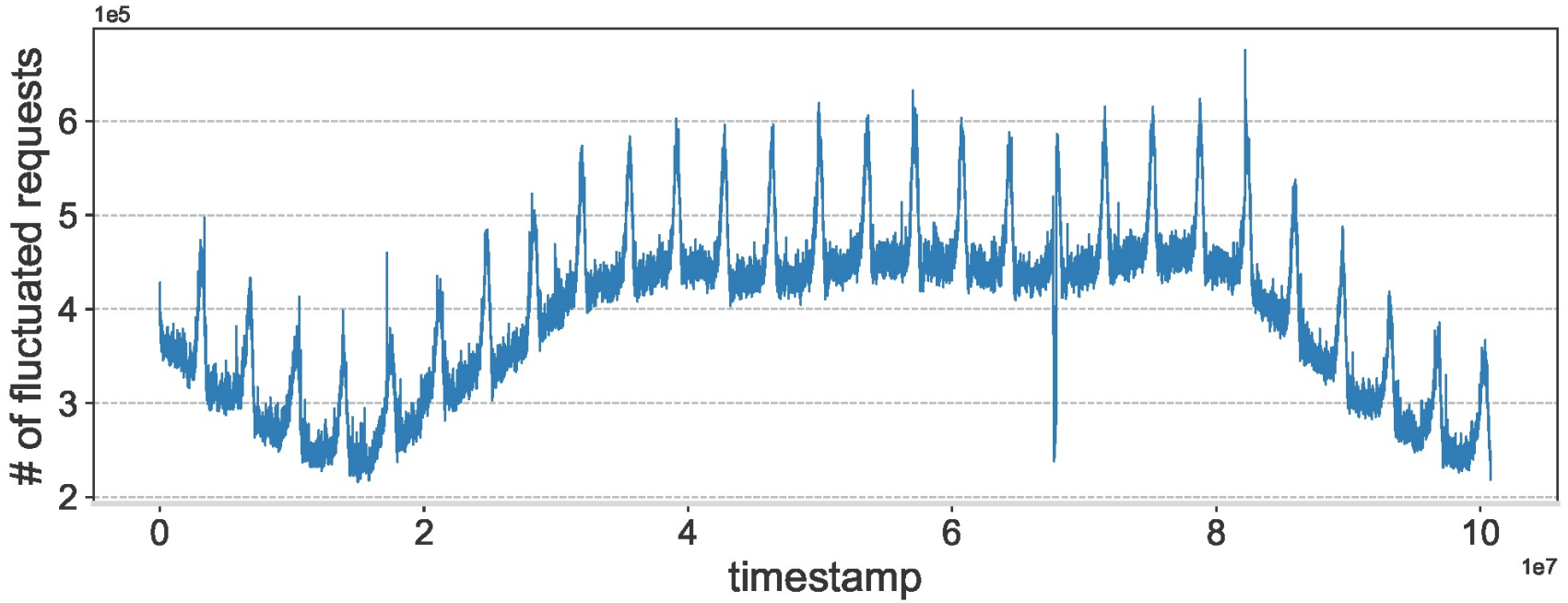
Challenges

Unpredictable workload: fluctuate requests
a 10-hour request trace across time intervals.

request rates—significant temporal fluctuations and recurring peaks
time-varying workload—make optimization complicate

Limited resource: stringent memory
edge servers from Alibaba—8GB to 128GB

Difficult in arranging microservice instances demanding a lightweight algorithm to perform in a memory limited edge.



Enabling features

- **One-shot decision-making:** reacts promptly; continuously responds to real-time environment
- **Dependency-based filtering:** figure out intrinsic dependency pattern; avoid detoured routing
- **Optimized for routing:** heuristic grouping; ensure each edge group has at least one instance
- **Flexible storage planning:** storage and provisioning trade-off; allow more warm instances to stay nearby

1 Introduction

2 Model and Formulation

3 Algorithm Design

4 Experiment and Results

Model and Formulation

System Model

entity	notation
Graph: server & links	$G = \{V, L\}, V = \{v_k\}, L = \{l_{k,k'}\}$
microservice	$M = \{m_i\}$
user request	$U = \{u_h\}, u_h = \{M_h, E_h\}$
requested services & dependencies	$M_h = \{m_i\}, E_h = \{e_{m_i \to m_j}\}$

indication	notation
path from v_a to v_b	$\pi(v_a, v_b), l_{k,k'} \in \pi$
users under server v_k	$U_k, u_h \in U_k, f(u_h) = k$
u_h choose v_a to handle $m_i \in M_h$	$v_a \leftarrow \mathrm{loc}^h(m_i)$
indication function	$\mathbb{1}_{[v_k \neq v_s]} = 1$ if $v_k \neq v_s$

Cost Model

deploy cost on server v_k:

$$\mathcal{K}_k = \sum_{m_i \in M} \kappa(m_i) \cdot x(i,k)$$

decision variable

deploy cost of m_i

where

$$d_{in}^h = \mathbb{1}_{[v_k \neq v_s]} \cdot \sum_{l_{i,j} \in \pi(v_k, v_s)} \frac{r_{in}^h}{b(l_{i,j})}$$

bandwidth

Completion Time Model

completion time for a user request u_h:

$$\mathcal{D}_h = d_{in}^h + \sum_{m_i \in M_h} d_c^h(m_i) + \sum_{e_{m_i \to m_j} \in E_h} d_l^h(e_{m_i \to m_j}) + d_{out}^h$$

$$d_c^h(m_i) = \frac{q(m_i)}{c(v_k)}$$

computation capacity

$$d_l^h\left(e_{m_i \to m_j}\right) = \sum_{l_{i,j} \in \pi(v_a, v_b)} \frac{r_{m_i \to m_j}^h}{b(l_{i,j})}$$

Model and Formulation

Formulation

objective function

$$\text{minimize } \lambda \sum_{v_k \in V} \mathcal{K}_k + (1-\lambda) \sum_{u_h \in U} \mathcal{D}_h \quad (1)$$

$$\text{s.t.} \quad \mathcal{D}_h \le \mathcal{D}_h^{max}, \forall u_h \in U \quad (2)$$

$$\sum_{v_k \in V} \mathcal{K}_k \le \mathcal{K}^{max} \quad (3)$$

$$\sum_{m_i \in M} x(i,k) \cdot \phi(m_i) \le \Phi(v_k), x \in \{0,1\}, \forall i, \forall k \quad (4)$$

QoS maintain constraints

memory capacity constraint

Objective: find a provisioning strategy for microservices to minimize cost and delay under the QoS and memory constraints.

Model and Formulation

❑ Reformulation

- Gurobi optimizer needs a structured decision variable.
- Method: illustrate the routing decision by service decision variable $y(\mathrm{h}, i, k)$
  - $y(\mathrm{h}, i, k) \in \{0,1\}$ denotes whether m_i serves request u_h on v_k
  - Regard $d^\mathrm{h}(m_i)$ as a transmission-computation cycle: $d^\mathrm{h}(m_i) = d_c^\mathrm{h}(m_i) + d_l^\mathrm{h}(e_{p \to m_i})$

Then the completion time was reformulated as:

$$\mathfrak{D}_\mathrm{h} = \sum_{m_i \in M_\mathrm{h}} \sum_{v_k \in V_i} y(\mathrm{h}, i, k)\left(d^\mathrm{h}(m_i) + d_{out}^\mathrm{h}\right)$$

where the variable $y(\mathrm{h}, i, k)$ must follow the constraints:

- path uniqueness: $\sum_{k \in V_i} y(\mathrm{h}, i, k) = 1, \forall \mathrm{h}, \forall m_i \in M_\mathrm{h}$
- feasibility (the service exist): $y(\mathrm{h}, i, k) \le x(i, k), \forall \mathrm{h}, \forall i, \forall k$
- binarity: $y(\mathrm{h}, i, k) \in \{0,1\}, \forall \mathrm{h}, \forall i, \forall k$

1 Introduction

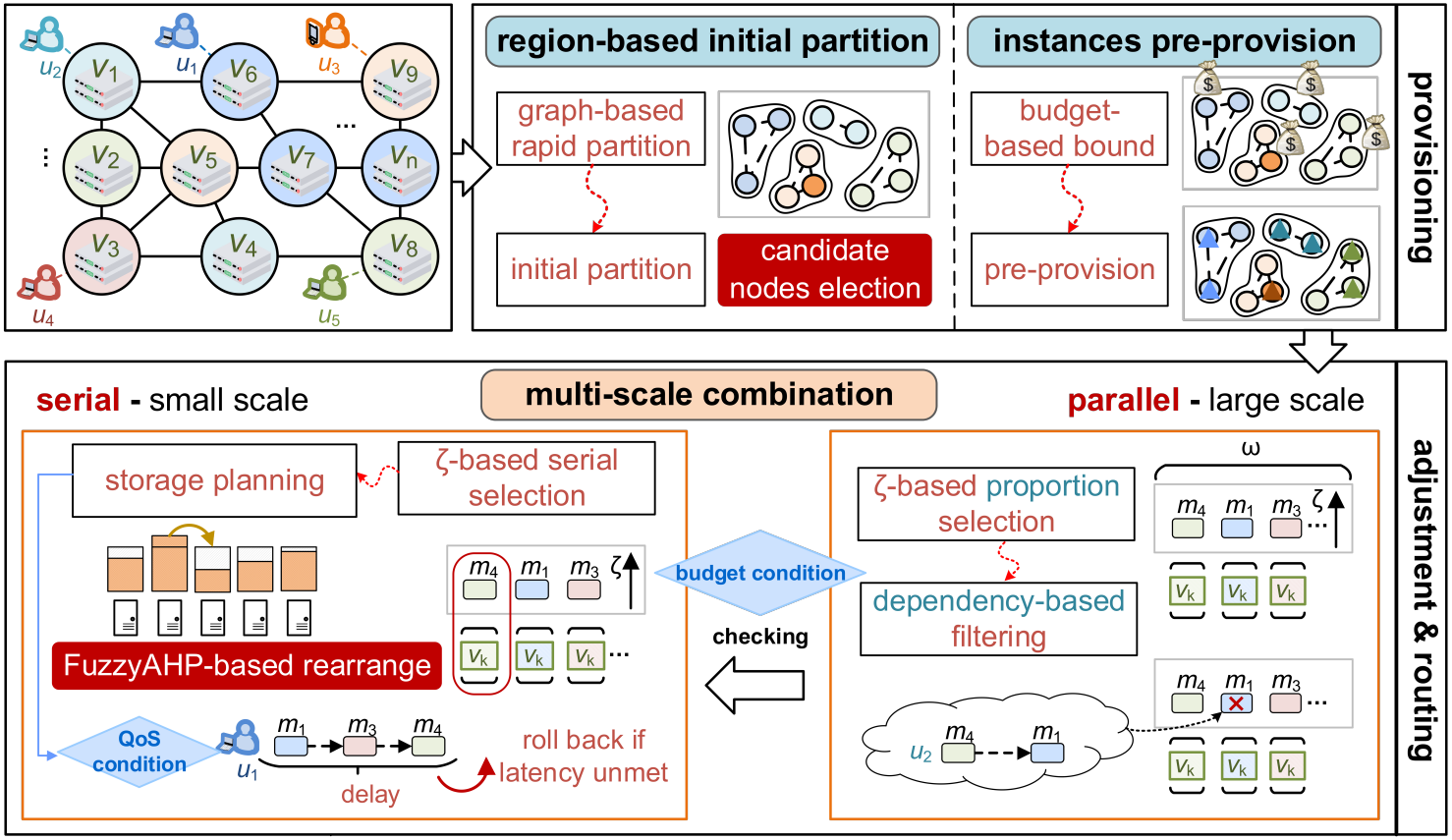
2 Background

3 Algorithm Design

4 Experiment and Results

SoCL framework overview

- Scalable optimization framework with Cost-efficiency and Latency reduction



region-based initial partition

- grouping nodes for $\forall m_i \in M$
- candidate node election

instance pre-provision

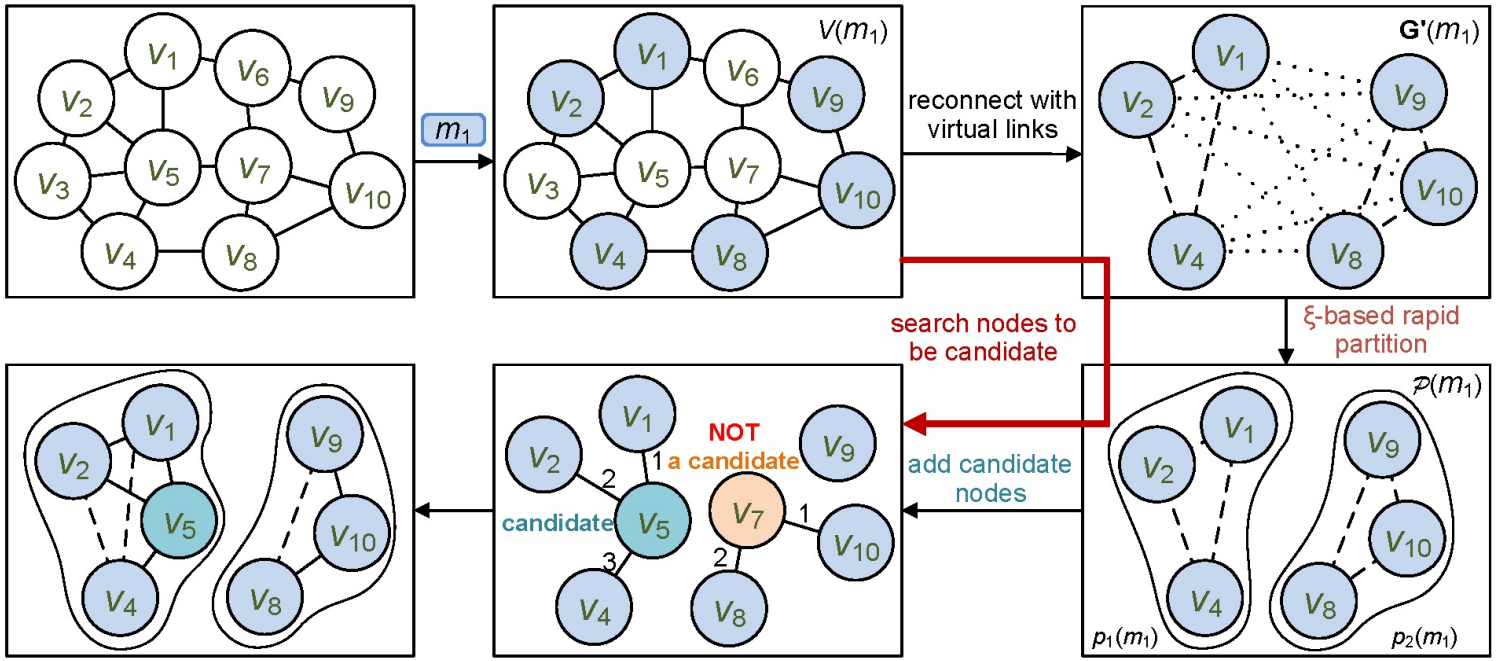
- budget bound determination
- node group initial provision

multi-scale combination

- dependency-based filtering
- fuzzy-based rearrange

Region-based initial partition

- provisioning perspective $V(m_i)$: for an individual microservice m_i, eg. m_1
- reconnect the graph with full-connected virtual links $G'(m_i)$
- partitioning into groups by thresholding virtual link bandwidth $\mathbb{P}(m_i) = \{p_s(m_i)\}$
- elect candidate node into groups to enhance connectivity



Region-based initial partition

Algorithm 1 Region-Based Initial Partitioning

Input: G, U, M;
Output: initial partition $\mathcal{P}$;

for each $m_i \in M$ **do**
  for each $v_k \in V$ **do**
    Construct the node set $V(m_i)$ of $G(m_i)$;
    Reconnect $G'(m_i)$ based on virtual links;
  for each $l'_{k,q} \in L'(m_i)$ **do**
    Add $l'_{k,q}$ to $G'(m_i)$ with $\mathbb{B}(l'_{k,q}) > \xi$;
  Generate the initial partitions $\mathcal{P}(m_i)$ by $G(m_i)$;
  for each partition $p_s(m_i) \in \mathcal{P}(m_i)$ **do**
    for each $v_k \in V \setminus V(m_i)$ **do**
      Choose v_k with $\mathcal{H}(v_k) > 2$;
      for each $v_a \in p_s(m_i)$ **do**
        Reorder $p_s(m_i)$ by $\arg\min\{\chi_{v_a}\}$;
      Check Δ^k with $v_a \in p_s(m_i)$;
      Add v_k to set $p_s(m_i)$ verified $\Delta^k < 0$;
Return initial partition $\mathcal{P}$;

Definition1(proactive factor): the completion time deviation between provisioning the instance only on candidate v_η and only on any v_a belonging to this group:

$$\Delta^\eta = \left[\sum_{v_i \in p_s(m_i)} \frac{r_i}{\mathbb{B}(l'_{i,\eta})}\right] \mathrm{I}_{m_i \to v_\eta} - \left[\sum_{v_i \in p_s(m_i) \setminus \{v_a\}} \frac{r_i}{\mathbb{B}(l'_{i,a})}\right] \mathrm{I}_{m_i \to v_a}$$

where r_i is the number of user requests for m_i inside the server group $p_s(m_i)$.

a candidate node must satisfy:

- proactive factor $\Delta^\eta < 0$
- number of direct link $\mathcal{H}(v_\eta^{(m_i)}) > 2$

communication intensity χ_{v_k} : accelerate the candidate validation.

Theorem1: a node $v_\eta^{(m_i)}$ must satisfy $\mathcal{H}(v_\eta^{(m_i)}) > 2$ to provide sufficient connectivity as a candidate.

Instance pre-provision

- **Maximum remaining budget:** subtracting the unit costs of all other microservices

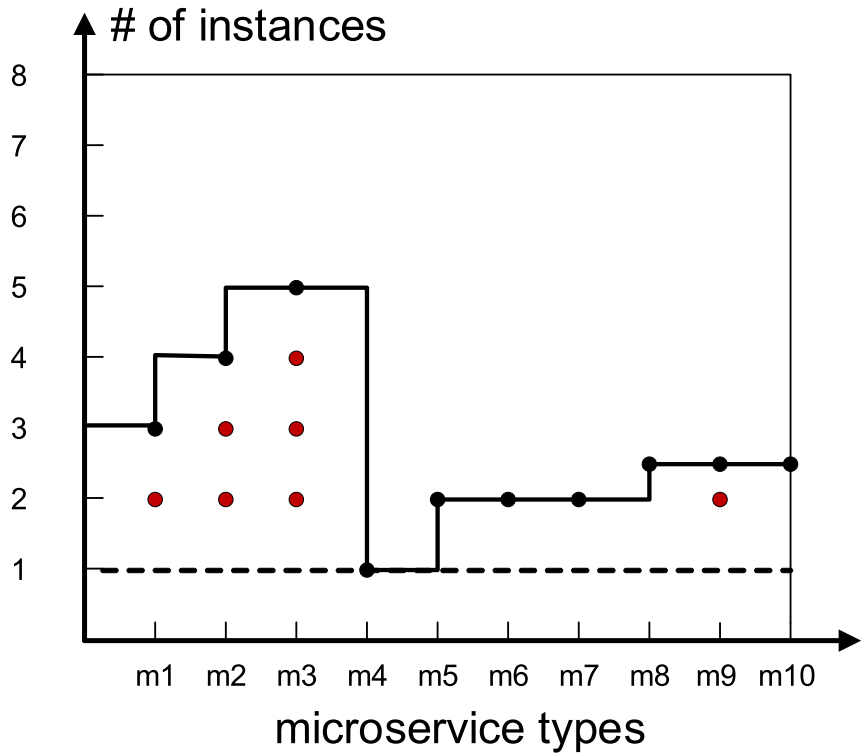
$$\mathcal{K}^u(m_i) = \mathcal{K}^{max} - \sum_{m_j \in M \setminus \{m_i\}} \kappa(m_j)$$

- **Maximum tolerant instance number:** take the floor of the remaining budget divided by the microservice unit cost

$$\mathcal{N}^u(m_i) = \lfloor \mathcal{K}^u(m_i) / \kappa(m_i) \rfloor$$

- **The upper bound:** the upper bound should not exceed the number of servers hosting requests for m_i, that is

$$\overline{\mathcal{N}}(m_i) = \min\{|V(m_i)|, \mathcal{N}^u(m_i)\}$$



Instance pre-provision

Algorithm 2 Instance Pre-provisioning

Input: initial partition $\mathcal{P}$;
Output: pre-provisioning $\mathcal{P}^t$;

1: **for** each $v_k \in V$ **do**
2: **for** each $m_i \in M$ **do**
3: Calculate $\left|\mathbb{U}_{v_k}^{m_i}\right|$;
4: **for** each $\mathcal{P}(m_i) \in \mathcal{P}$ **do**
5: **for** each $p_s(m_i) \in \mathcal{P}(m_i)$ **do**
6: Update $\left|\mathbb{U}_{p_s(m_i)}\right|$;
7: Calculate $\varepsilon_s(m_i)$;
8: **if** $\varepsilon_s(m_i) \cdot \overline{\mathcal{N}}(m_i) \geq |p_s(m_i)|$ **then**
9: Pre-provision $p_s^t(m_i) \leftarrow m_i$;
10: **else**
11: **for** each $v_k \in p_s(m_i)$ **do**
12: Update $\mathbb{D}_{p_s(m_i)}(v_k)$;
13: **while** $|p_s^t(m_i)| < \varepsilon_s(m_i) \cdot \overline{\mathcal{N}}(m_i)$ **do**
14: Pre-provision with $\arg\min\{\mathbb{D}_{p_s(m_i)}(v_k)\}$;
15: **Return** pre-provisioning $\mathcal{P}^t$;

- $\mathbb{U}_{v_k}^{m_i}$: the users requesting for m_i ($m_i \in M_h$) on server v_k ($u_h \in U_k$)
- $\mathbb{U}_{p_s(m_i)}$: the users requesting for m_i ($m_i \in M_h$) in partition $p_s(m_i)$
- $\varepsilon_s(m_i) = \frac{\left|\mathbb{U}_{p_s(m_i)}\right|}{\sum_{p_s(m_i) \in \mathbb{P}(m_i)} \left|\mathbb{U}_{p_s(m_i)}\right|}$: the demand ratio

the allocated quota is sufficient

Definition2(instance contribution): the estimated overall completion time for the group $p_s(m_i)$ if v_k is the only server hosting an instance of m_i :

$$\mathbb{D}_{p_s(m_i)}(v_k) = \sum_{f(u_h) \in p_s(m_i) \setminus \{v_k\}} \left(\frac{r_i}{\mathbb{B}\left(l'_{f(u_h),k}\right)} + \frac{q(m_i)}{c(v_k)}\right)$$

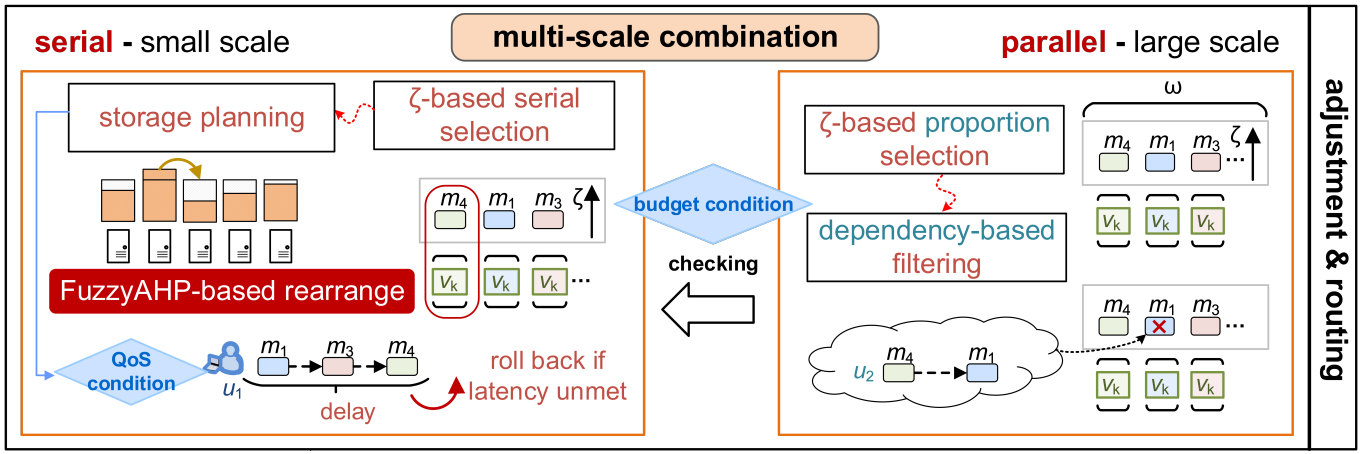
where $f(u_h)$ is the other nodes within $p_s(m_i)$ that host requests of m_i

Multi-scale combination

Notes

$U_{i,k}$: the users relying on instance m_i at v_k

$f(u_h)$: a server hosting u_h's request for m_i



- Estimated overall completion time: $\psi\left(\mathcal{P}^t(m_i)\right)$
- Completion time before remove v_k: $\psi\left(\mathcal{P}'^t(m_i)\right) = \sum_{u_h \in U_{i,k}} \left(\frac{r_i^h}{b\left(l_{f(u_h),k}\right)} + \frac{q(m_i)}{c(v_k)}\right)$
- After removing v_k: each influenced user $u_h \in U_{i,k}$ should find new reliance server v_q
- Completion time after remove v_k: $\psi\left(\mathcal{P}''^t(m_i)\right) = \sum_{u_h \in U_{i,k}} \left(\frac{r_i^h}{\max\limits_{v_q \in ps^t(m_i)} b\left(l_{f(u_h),q}\right)} + \frac{q(m_i)}{c(v_q)}\right)$

Definition3(latency loss): the estimated completion time increase after the removal of v_k:

$$\zeta_{i,k} = \psi\left(\mathcal{P}''^t(m_i)\right) - \psi\left(\mathcal{P}'^t(m_i)\right)$$

Multi-scale combination

Algorithm 3 Multi-scale Combination

Input: initial partition $\mathcal{P}$, pre-provisioning $\mathcal{P}^t$;
Output: provisioning $\mathcal{X}$;

1: **while** $\sum_{k=1}^{|V|} \mathcal{K}_k \geq \mathcal{K}^{\max}$ **do** ▷ parallel: large-scale
2: Update $\zeta = \{\zeta_{i,k}\}$ in Algorithm 4;
3: Update $\Omega = \{\Omega_{i,k}\}$ with ω;
4: Filter Ω by service dependency E_h;
5: Parallel combine $\Omega_{i,k} \in \Omega$ from $\mathcal{P}^t$;
6: **while** $\delta > 0$ **do** ▷ serial: small-scale
7: Calculate Q' of $\mathcal{P}^t$;
8: Choose $\Omega_{i,k} \leftarrow \arg\min_{\zeta_{i,k}} \mathcal{P}^t$;
9: Calculate Q'' of $\mathcal{P}^t \setminus \{v_k\}$ and update δ;
10: Serial combine $\Omega_{i,k}$ from $\mathcal{P}^t$;
11: Storage planning with Algorithm 5;
12: Constraint checking on latency with Equation (4);
13: **if** $\mathcal{D}_h > \mathcal{D}_h^{\max}$ **then**
14: Roll-back: add v_k back to $\mathcal{P}^t$;
15: **Continue** the while loop;
16: **Return** provisioning $\mathcal{X}$;

Parallel combine: large-scale gradient descent

Line2: update latency loss list $\zeta = \{\zeta_{i,k}\}$ from Alg.4

- at least one instance to avoid fallback to cloud

Line3: regulate the combine number with ω% minimal

- only combine ω% lower $\zeta_{i,k}$ to limit combine speed

Line4: filter the "dependency conflicted" instances

- prevent significant increase in completion time

Algorithm 4 UPDATE_INSTANCE_SET Function

Input: pre-provisioning $\mathcal{P}^t$;
Output: latency loss list ζ;

1: **for** each $m_i \in M$ **do**
2: **if** $\sum_{p_s^t(m_i) \in \mathcal{P}^t(m_i)} |p_s^t(m_i)| = 1$ **then**
3: **Continue for**: $M \leftarrow M/\{m_i\}$;
4: **for** each $v_k \in \mathcal{P}^t(m_i)$ **do**
5: Calculate $\zeta_{i,k}$;
6: Update $\Omega_{i,k} \leftarrow \{m_i | \arg\min \zeta_{i,k}\}$;
7: **Return** latency loss list ζ;

Multi-scale combination

Algorithm 3 Multi-scale Combination

Input: initial partition $\mathcal{P}$, pre-provisioning $\mathcal{P}^t$;
Output: provisioning $\mathcal{X}$;

while $\sum_{k=1}^{|V|} \mathcal{K}_k \geq \mathcal{K}^{\max}$ **do** ▷ parallel: large-scale
  Update $\zeta = \{\zeta_{i,k}\}$ in Algorithm 4;
  Update $\Omega = \{\Omega_{i,k}\}$ with ω;
  Filter Ω by service dependency E_h;
  Parallel combine $\Omega_{i,k} \in \Omega$ from $\mathcal{P}^t$;
while $\delta > 0$ **do** ▷ serial: small-scale
  Calculate Q' of $\mathcal{P}^t$;
  Choose $\Omega_{i,k} \leftarrow \arg\min_{\zeta_{i,k}} \mathcal{P}^t$;
  Calculate Q'' of $\mathcal{P}^t \setminus \{v_k\}$ and update δ;
  Serial combine $\Omega_{i,k}$ from $\mathcal{P}^t$;
  Storage planning with Algorithm 5;
  Constraint checking on latency with Equation (4);
  if $\mathcal{D}_h > \mathcal{D}_h^{\max}$ **then**
    Roll-back: add v_k back to $\mathcal{P}^t$;
    Continue the while loop;
Return provisioning $\mathcal{X}$;

estimated objective before combination

estimated objective after combination

- small-scale gradient $\delta = Q' - Q'' + \Theta$

disturbance factor

- regard $\delta < 0$ as the sign of the objective rise back
- disturbance factor Θ: avoid early rise back

Multi-scale combination

Algorithm 5 Storage Planning

Input: pre-provisioning $\mathcal{P}^t$;
Output: intermediate provisioning $\mathcal{X}'$;

1: **if** $\sum_{v_k \in V} \Phi(v_k) \geq \sum_{m_i \in M} |\mathcal{P}^t(m_i)| \cdot \phi(m_i)$ **then**
2: **for** each $m_i \in M$ **do** ▷ FuzzyAHP progress
3: Record $\kappa(m_i), \phi(m_i)$;
4: **for** each $v_k \in V(m_i)$ **do**
5: **for** each $u_h \in U_k$ **do**
6: Calculate and record $\left|\mathbb{U}_{v_k}^{m_i}\right|, \mathbb{R}_{v_k}^{m_i}$;
7: Calculate local demand factor ρ;
8: **for** each $v_k \in V$ **do**
9: **while** $\sum_{m_i \in M} x(i,k) \cdot \phi(m_i) > \Phi(v_k)$ **do**
10: Choose m_j with $\arg\min \rho_k$;
11: Reorder $v_q \in V \setminus \{v_k\}$ by $\arg\max b(l_{k,q})$
12: **if** $x(j,q) = 0$ and $\phi(m_j) \leq \overline{\Phi}(v_k)$ **then**
13: Let $x(j,k) = 0$ and $x(j,q) = 1$;
14: **Continue** the while loop;
15: Record current placement $x(i,k)$ as $\mathcal{X}'$;
16: **else**
17: **Continue** the while loop in Algorithm 3;
18: **Return** intermediate provisioning $\mathcal{X}'$;

Definition4(local demand factor): the local demand factor $\rho = \{\rho_k\}$ reflects the importance of providing instance m_i on server v_k, where $\rho_k = \{\rho_{v_k}^{m_i}\}$ denotes the instance priority list of a particular server v_k.

FuzzyAHP method:

- deployment cost $\kappa(m_i)$
- storage requirement $\phi(m_i)$
- number of requesting users $\left|\mathbb{U}_{v_k}^{m_i}\right|$
- order factor $\mathbb{R}_{v_k}^{m_i} = (3u_{f}{}_{v_k}^{m_i} + 2u_{l}{}_{v_k}^{m_i} + u_{m}{}_{v_k}^{m_i}) / \left|\mathbb{U}_{v_k}^{m_i}\right|$

(where $u_{f}{}_{v_k}^{m_i}$, $u_{l}{}_{v_k}^{m_i}$, and $u_{m}{}_{v_k}^{m_i}$ denote the number of users for whom m_i is at first, last, and intermediate dependency chain)

1 Introduction

2 Model and Formulation

3 Algorithm Design

4 Experiment and Results

Experiment and Results

Basic Setting

- Hardware: Windows 10 with an Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz, NVIDIA RTX5000 GPU, and 128GB RAM.
- Dataset: Microservices (Version 1.0)[1] .
- Project: **eShopOnContainers**

Comparison

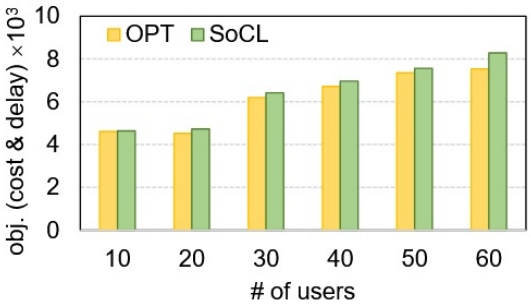
- Random Provisioning (RP).
- Joint Deployment and Routing (JDR)[2].
- Greedy Combine with Objective Gradient (GC-OG)

[1] Imranur, M., Panichella, S., & Taibi, D. (2019). A curated dataset of microservices-based systems. Joint Proceedings of the Summer School on Software Maintenance and Evolution (CEUR-WS), Tampere, Finland.
[2] Peng, K., Wang, L., He, J., Cai, C., & Hu, M. (2024). Joint optimization of service deployment and request routing for microservices in mobile edge computing. IEEE Transactions on Services Computing.

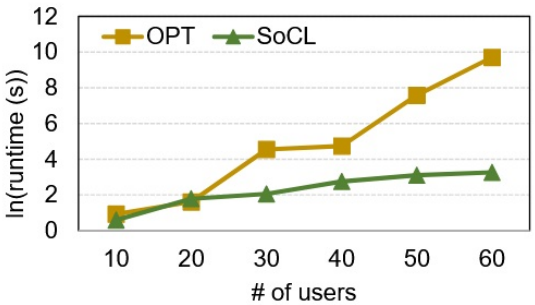
Experiment and Results

Comparison with Gurobi Optimizer

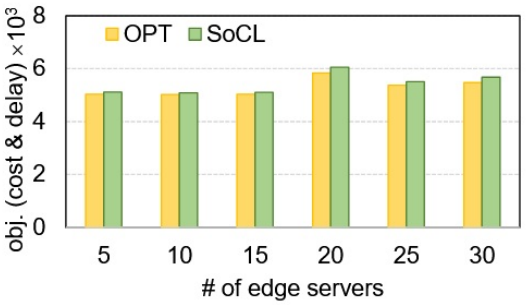
- Gurobi achieves slightly better objective values, but the differences are minimal.
- SoCL outperforms Gurobi in runtime as the user scale and server scale increases.
- SoCL can deliver near-optimal solutions with significantly higher efficiency.



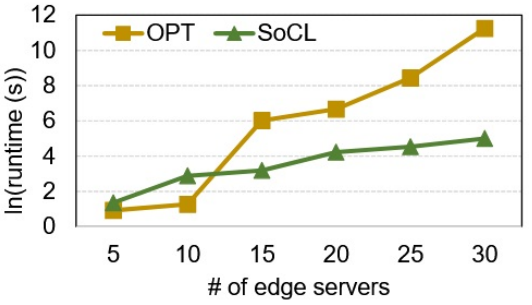
(a) # of users (obj).



(b) # of users (runtime).



(c) # of edge servers (obj).

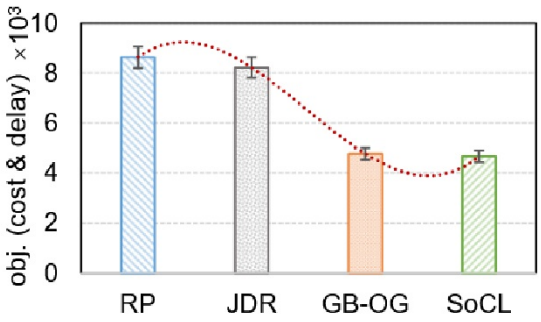


(d) # of edge servers (runtime).

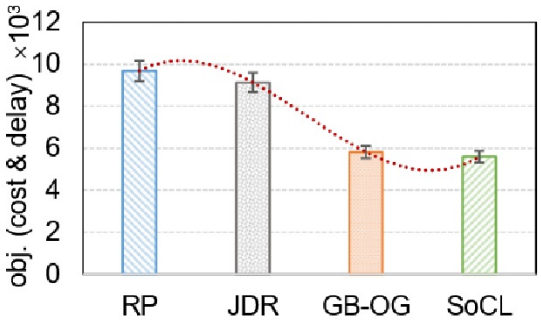
Experiment and Results

Comparison on Baselines

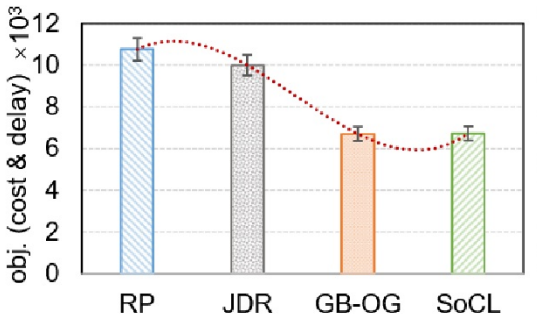
- SoCL achieved the lowest objective values across all user scales.
- The difference in performance became clearer as the user scale increased.
- SoCL's efficient search and provisioning strategy effectively avoids resource redundancy and search inefficiency.



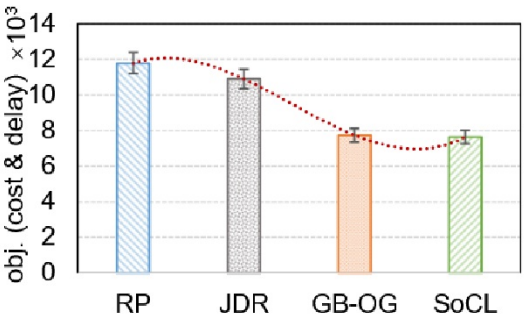
(a) # of users 80.



(b) # of users 120.

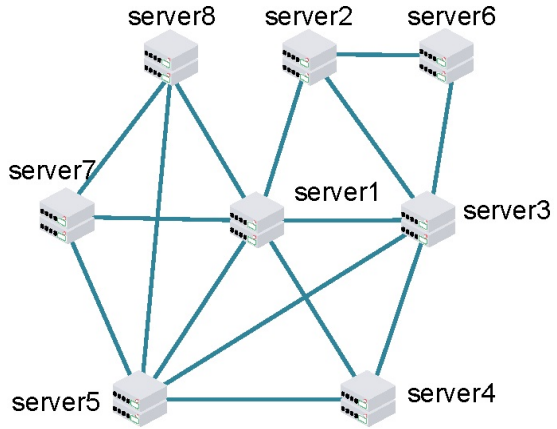


(c) # of users 160.



(d) # of users 200.

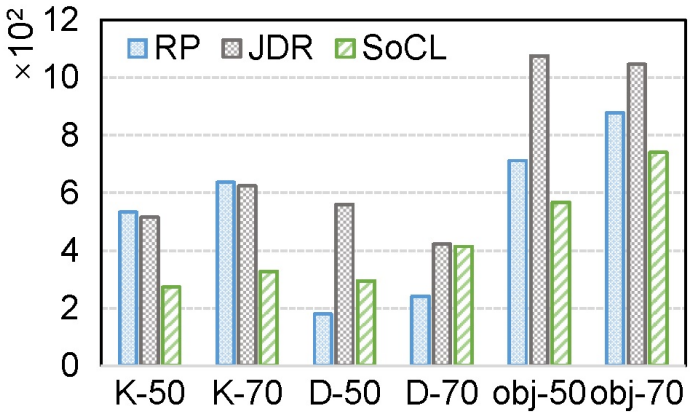
Experiment and Results



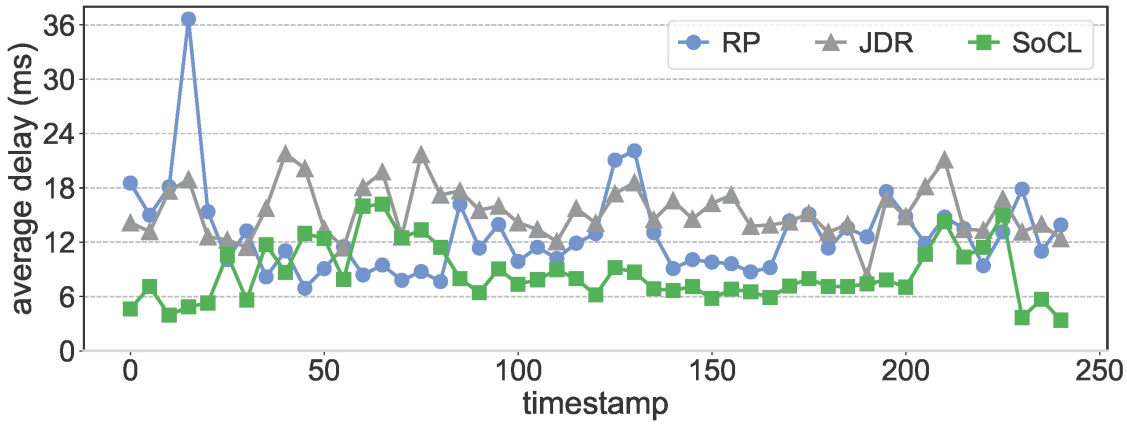
Evaluation on Kubernetes

Testbed Results

- SoCL can balance the deployment cost with the users' requirements and place the instances in a proper place.
- SoCL consistently achieved the lowest average delay during the latency trace.



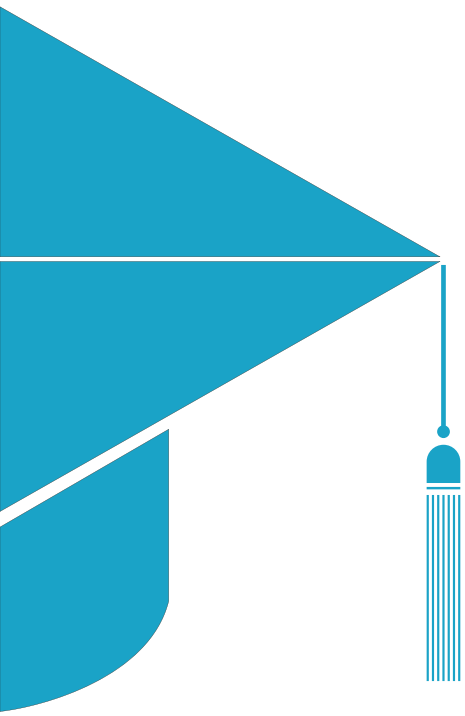
Objectives with 8 servers



Avg delay trace with 16 edge servers

Conclusion

- **SoCL**: a scalable framework for efficient microservice provisioning, which optimally positions microservice instances to balance cost and latency based on user demands.
- **Experiments:** SoCL consistently outperforms baselines, handling large-scale user requests and network expansions effectively.
- **Compared to Gurobi:** SoCL achieves significant reductions in cost and latency while delivering execution times up to one order of magnitude faster.
- **Future work:** incorporate user behavior modeling and preference integration to support context-aware resource management.



THANK YOU