Mobile-Kube: Mobility-aware and Energy-efficient Service Orchestration on Kubernetes Edge Servers

Saeid Ghafouri, Alireza Karami, Danial Bidekani Bakhtiarvand, Aliakbar Saleh Bigdeli, Sukhpal Singh Gill and Joseph Doyle

Introduction

Energy Proportional Computing

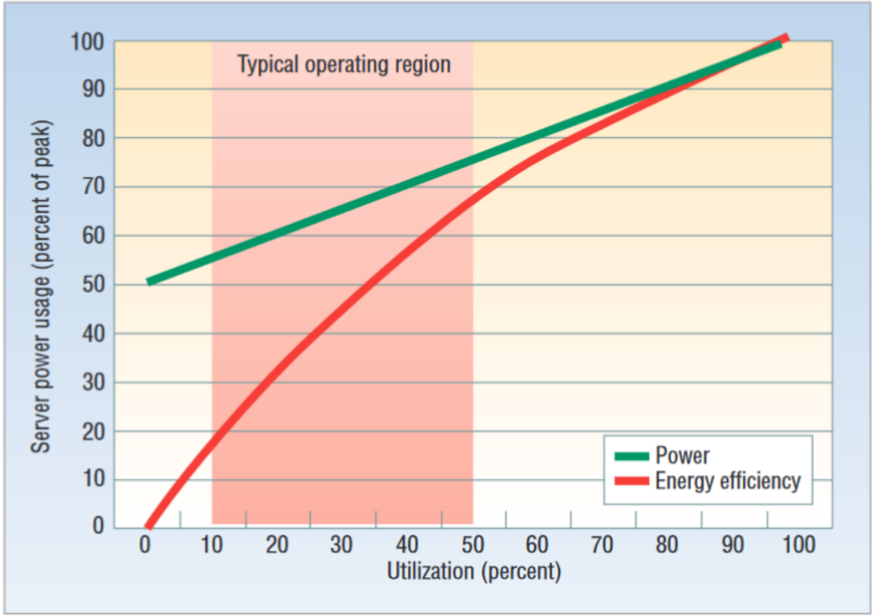


Figure 2. Server power usage and energy efficiency at varying utilization levels, from idle to peak performance. Even an energy-efficient server still consumes about half its full power when doing virtually no work.

Barroso, L.A. and Hölzle, U., 2007. The case for energy-proportional computing. *Computer*, *40*(12), pp.33-37.

Recommendations for improving consolidation

- Recommended utilisation levels of 75% for Hyperscale server users and 45% for smaller users in the US
  - A. Shehabi et al., "United state data center energy usage report", 2016.
- In the UK, HMG Sustainable Technology Advice and Reporting (STAR) have identified a “consolidation programmes to maximise use of capacity” as best practice for achieving this goal.
  - Depart for Environment Food & Rural Affairs, "Sustainable Technology Annual report 2018 to 2019," October 2019.

Commitments by Cloud Providers

- Microsoft committed to carbon neutrality by 2030
- Amazon committed to carbon neutrality by 2040
- What about Edge devices?
- In some ways better as they do not need supporting infrastructure for cooling
    - Ahvar, E., Orgerie, A.C. and Lebre, A., 2019. Estimating energy consumption of cloud, fog and edge computing infrastructures. *IEEE Transactions on Sustainable Computing*.
- However, mostly focused on latency reduction only.
- This can lead to low utilisation of resources
- Mobile-Kube attempts to balance the latency reduction and consolidation objectives for containerised services on the edge
- Also consider user mobility and how this affects the objectives

Container Orchestration Frameworks

- Containerized softwares
- Google Borg
- Docker Swarm
- Kubernetes!

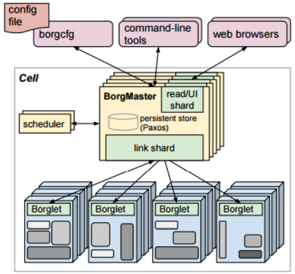
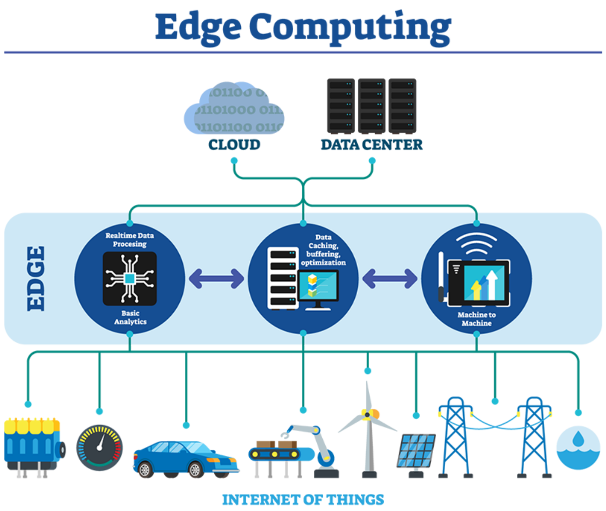
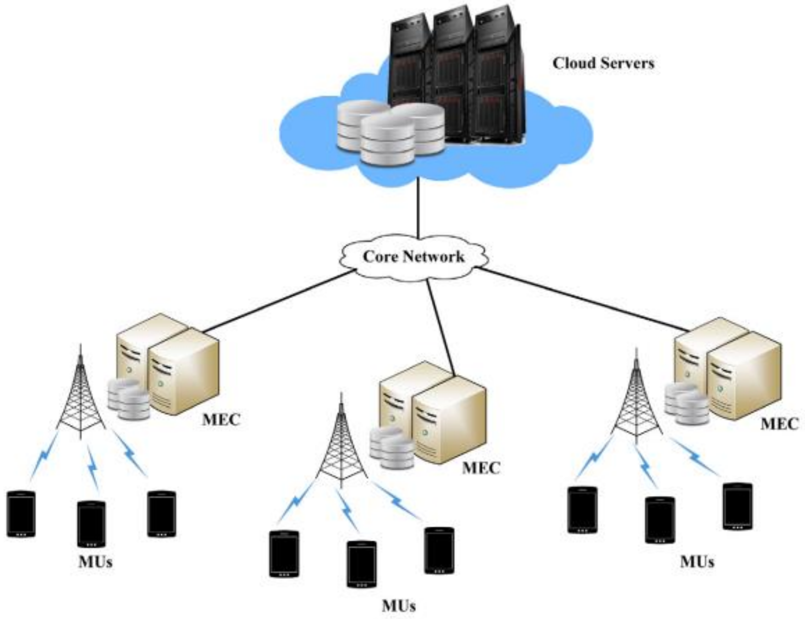


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

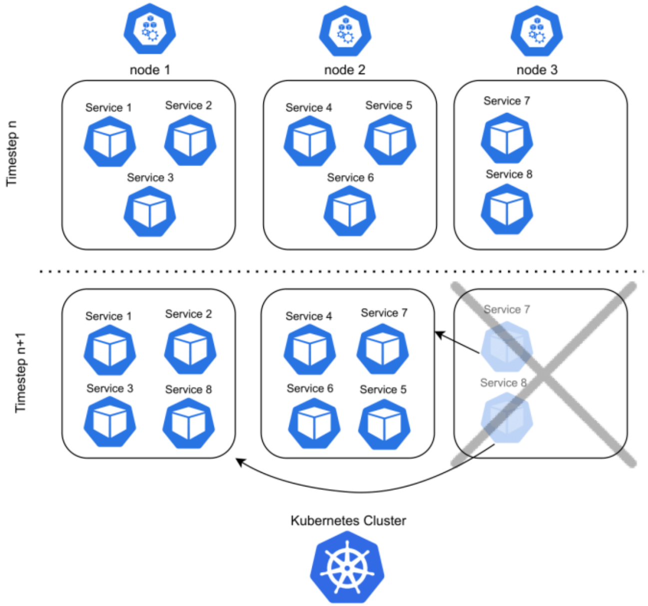
6

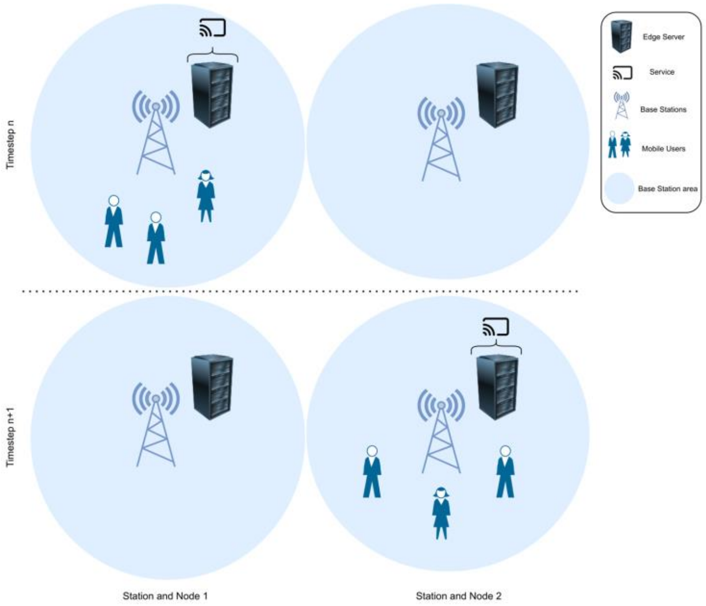
Edge Computing



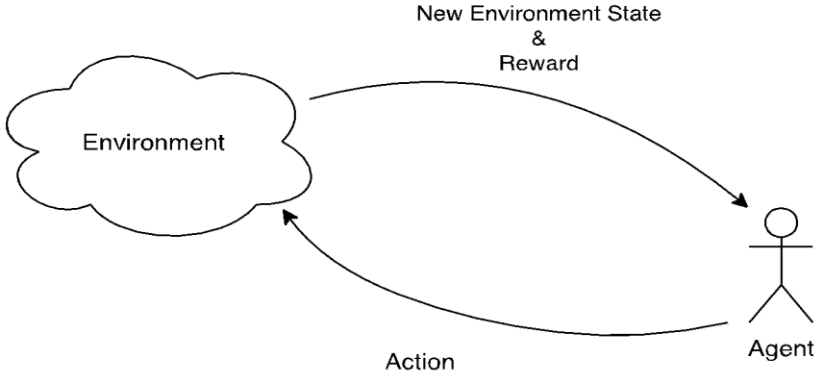


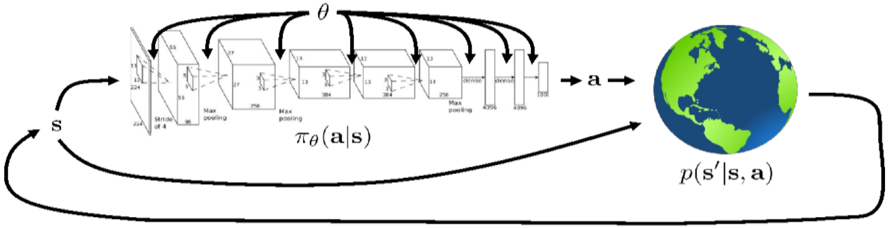
Problem statement





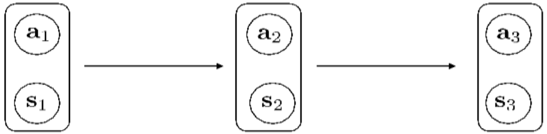
Reinforcement Learning





$$\underbrace{p_\theta(\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1)\prod_{t=1}^{T}\underbrace{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}_{\text{Markov chain on }(\mathbf{s},\mathbf{a})}$$

$$p((\mathbf{s}_{t+1},\mathbf{a}_{t+1})|(\mathbf{s}_t,\mathbf{a}_t)) =$$
$$p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)\pi_\theta(\mathbf{a}_{t+1}|\mathbf{s}_{t+1})$$

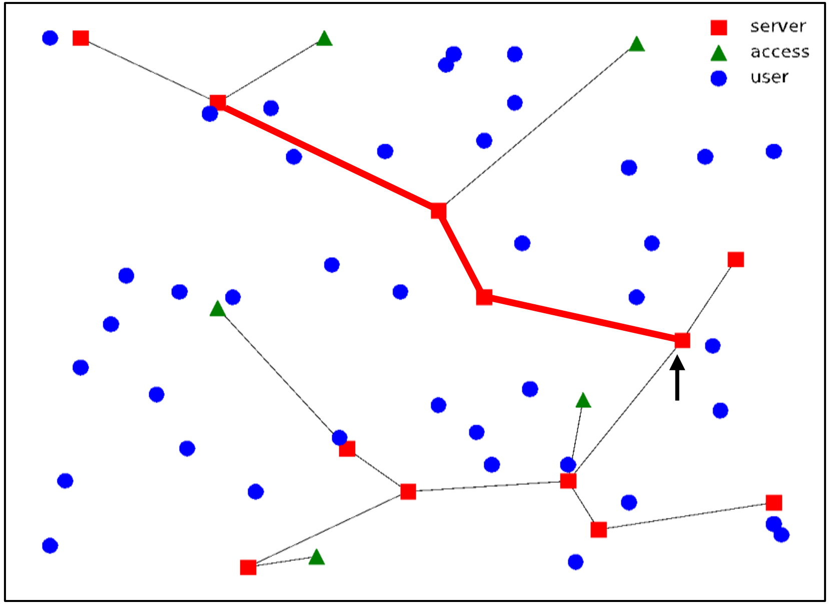


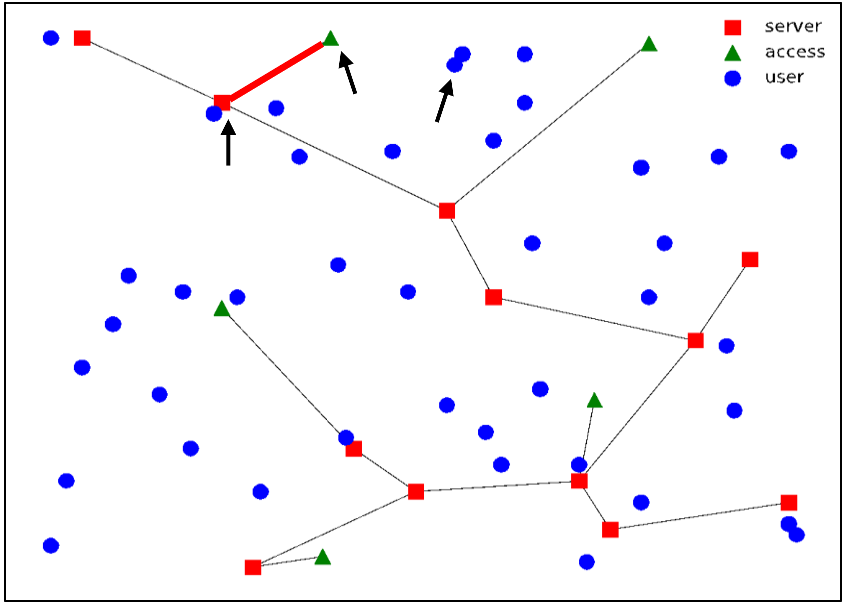
Contribution of this work

- A new design for reducing the **latency** and **energy consumption** on Kubernetes-driven edge nodes.
- Use of RL for achieving a trade-off between maintaining reasonable energy consumption and latency. Proposed the use of a distributed RL method named **IMPALA**.
- To test the efficiency of our method we have implemented of a simulation framework for training and a real-world emulator on top of **real-world Kubernetes.**
- The RL based method is able to achieve similar energy efficiency of the heuristic methods while reducing the latency by 43%.

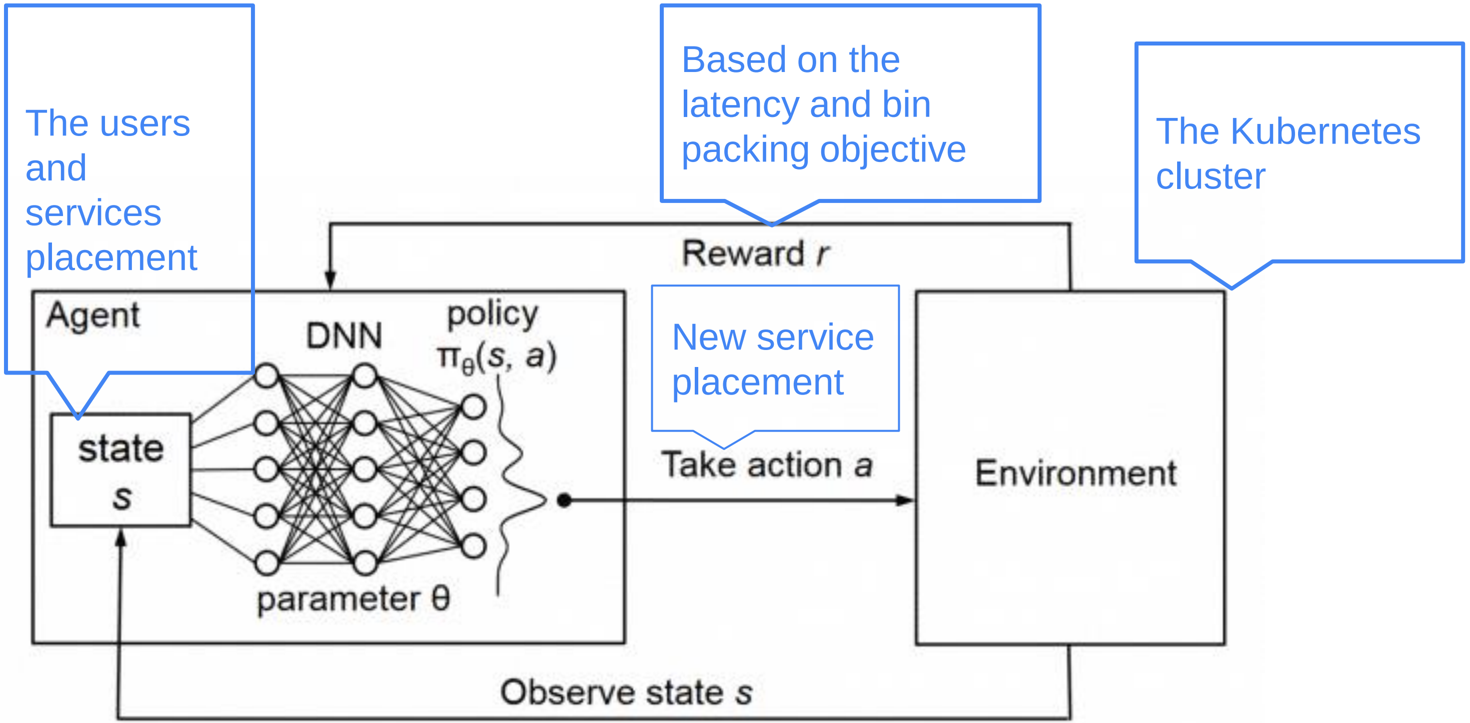
Proposed RL Solution

Latency Reduction objective





Proposed RL solution (Overview)



Proposed RL solution (cont)

- States: concatenation of two arrays
  - An array containing the service placements
  - An array containing the users closest station
- Actions: next placement of the services in the nodes
- Rewards

$$R = w_1 R_b + w_2 R_l$$

  - The latency objective is computed from the inverse of the total distances of the the users from their service
  - The binpacking objective is simply the number of empty servers
- Policy network: A 64*64 fully connected neural network

Used RL Methods

- We used three algorithms for our experiments:
- **Vanilla Policy Gradient**: The basis of all policy gradient methods used before in system research for bitrate adaptation
- **PPO**: A more advanced version of the policy gradient which tries to minimize the variance by clipping the objective function
- **IMPALA**: One of the newest widely used distributed RL algorithm with fast convergence and low variance
- Also use heurstic methods (greedy and binpacking) for comparison

System Design

Kubernetes Internal Structure

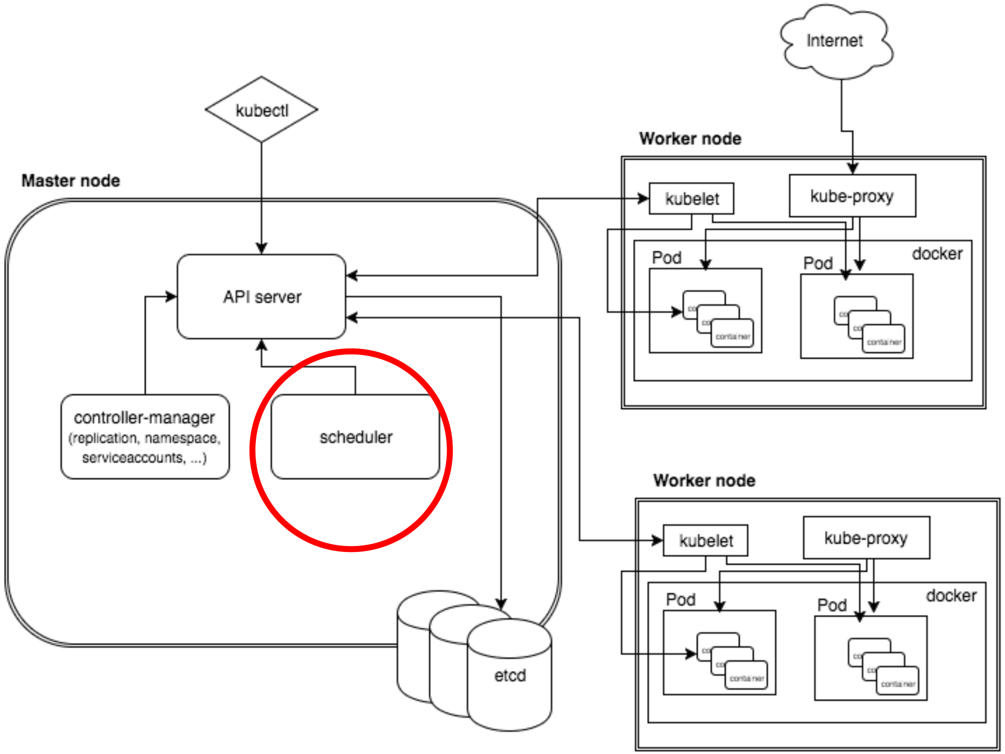


image from: https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-complete-guide/

Kubernetes Resource Model

- Request: reserved amount

  of resource for a container

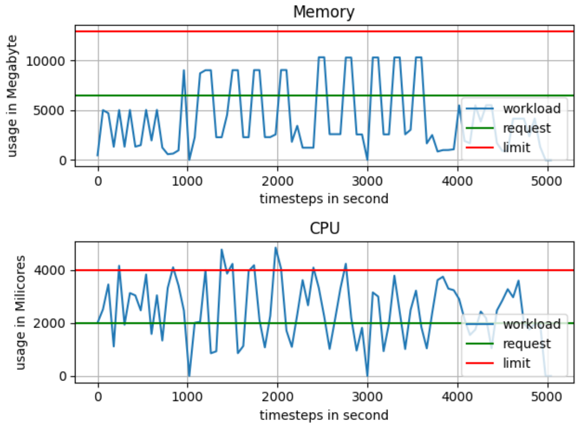
- Limit: Maximum amount of

  Of resource for a container

- Exceeding limit: OOM error for memory

  and throttling for CPU

- We used resource request for scheduling



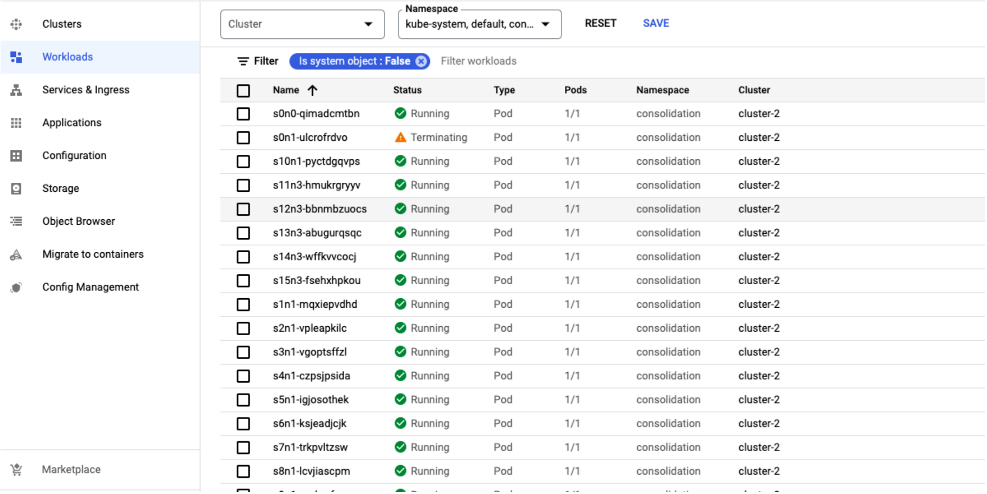

```yaml
spec:
  containers:
  - name: change
    image: k8s.gcr.io/ubuntu-slim:0.1
    resources:
      limits:
        cpu: 351m
        memory: 150Mi
      requests:
        cpu: 200m
        memory: 50Mi
    command: ["/bin/sh"]
    args: ["-c", "while true; do timeout 0.5s yes >/dev/null; sleep 0.5s; done"]
```

Kubernetes Default Scheduler

- Pods the smallest scheduling unit in kubernetes
- Currently the scoring is done based-on the rules defined by Kubernetes user and also heuristic algorithms
- Nodes available resources
- Requested resources
- A two step process
    - **Filtering**: Filtering out suitable nodes
    - **Scoring**: Ranks the nodes based-on a sets of criteria to find the most suitable node
- Assign the pod to the node with the highest rank

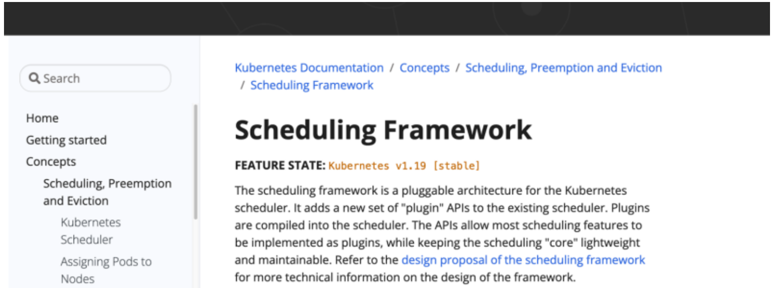
Limitations of the default scheduler

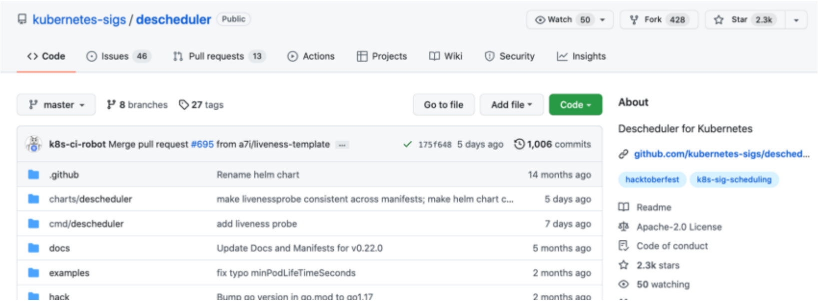
- Using the Kubernetes builtin custom scheduler feature was not feasible
- No migration of the pods based on external metrics
- We implemented this feature as deleting on pods in one place and restarting it in the destination node outside the cluster using the Python client API



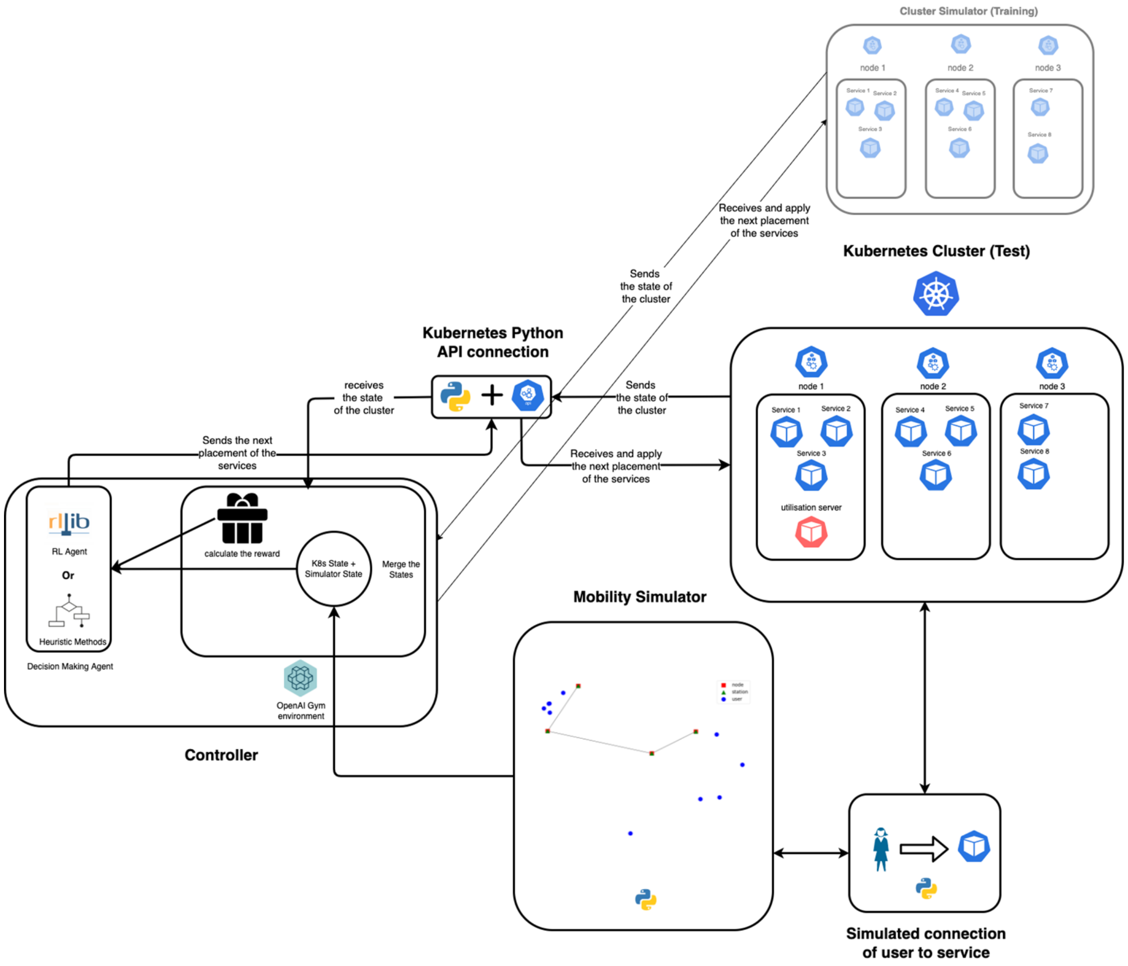
Our design for changing the Scheduler

- Using the Python client API of Kubernetes
- For the emulation setting we discard the built in scheduler decision and used our own scheduler which resides outside the K8S cluster
- A better design for this should be fully integrated into the K8S
- The user mobility side is simulation





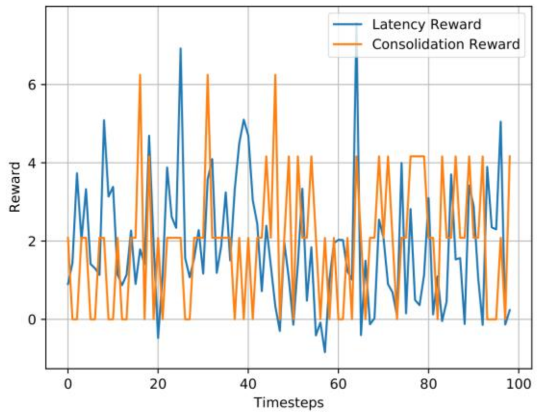
System design



Experimental Setup and Results

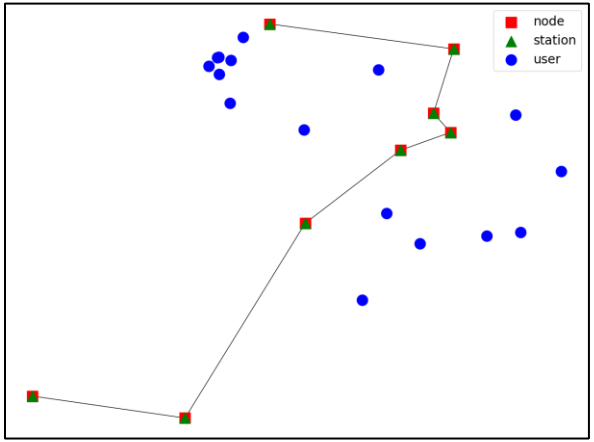
System setting and datasets

- Cabspotting dataset: The Cabspotting dataset contains GPS traces of taxi cabs in San Francisco (USA), collected in May 2008.
- http://www.antennasearch.com/ for the location of cell towers
- Python simulator for user mobility
- 8 kubernetes GKE nodes and 16 stateless services
- Reward scaling

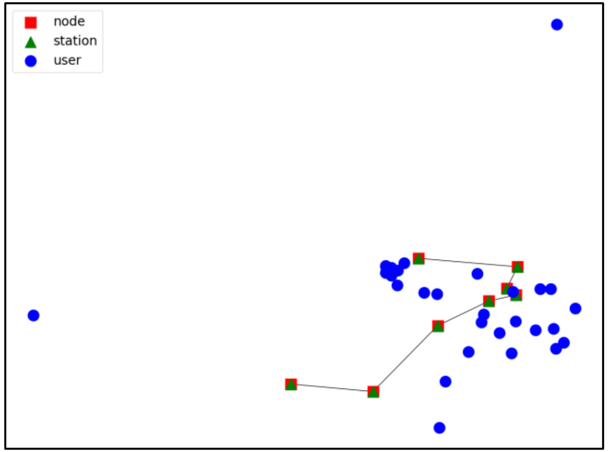


Picture of the network

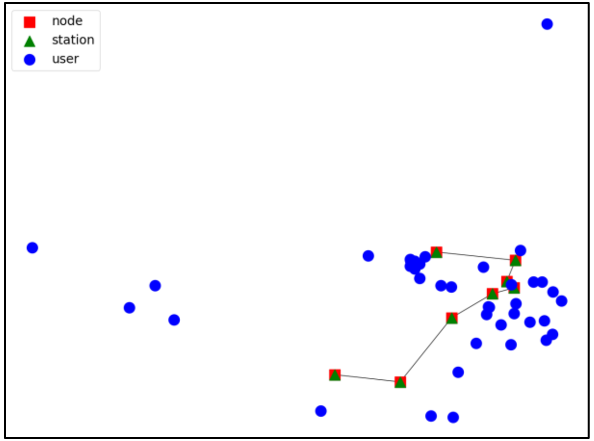
- Co-located stations and servers
- 5 minute interval mobility



16 users



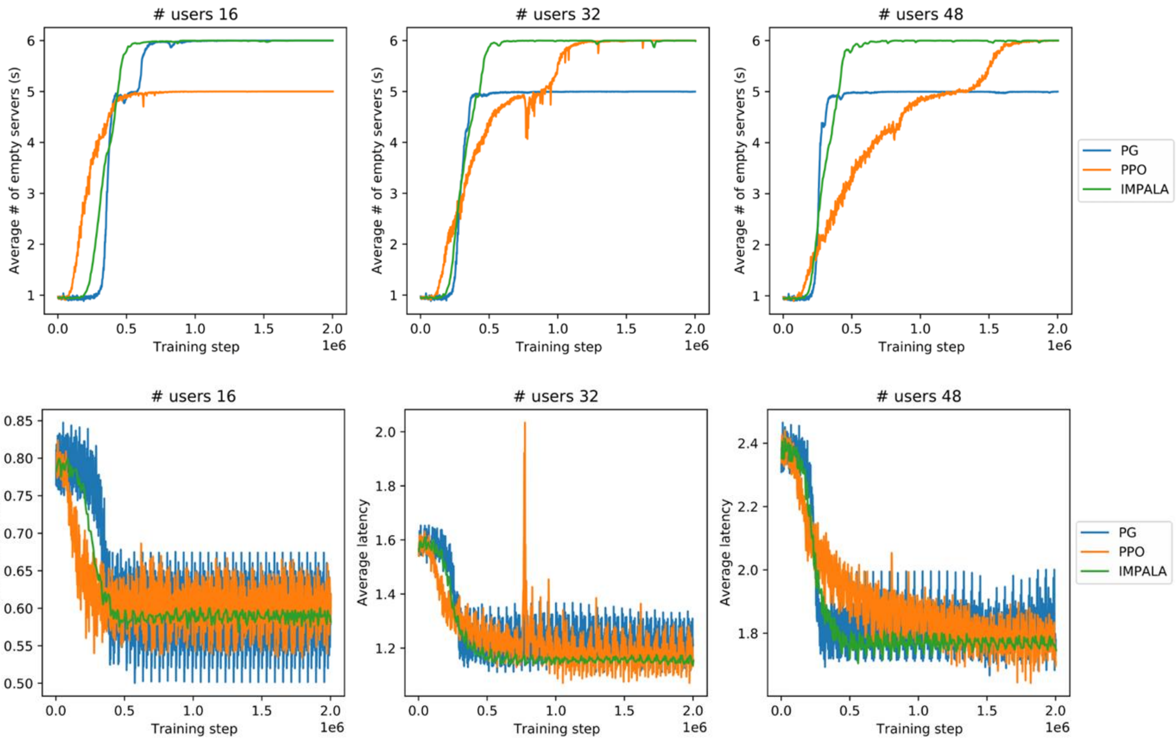
32 users



48 users

Results - Training

- For training we generated a dataset based on the user locations around the servers
- A simulator that used the real-world K8S for training

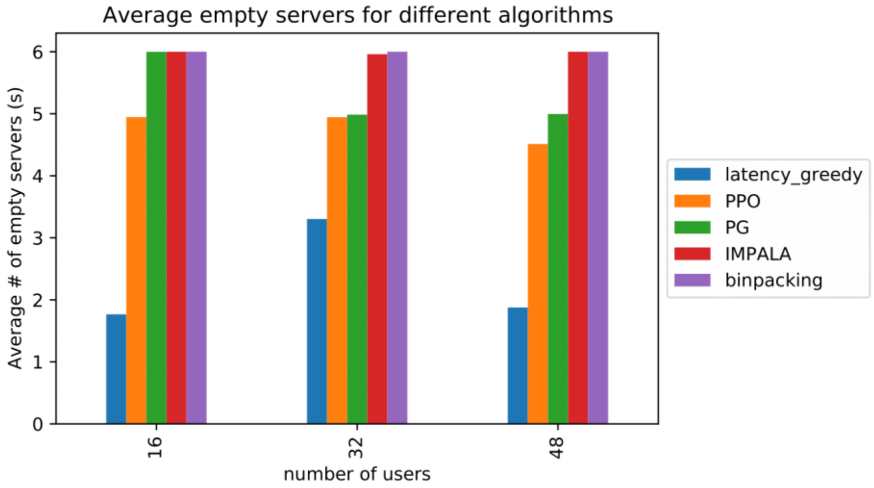


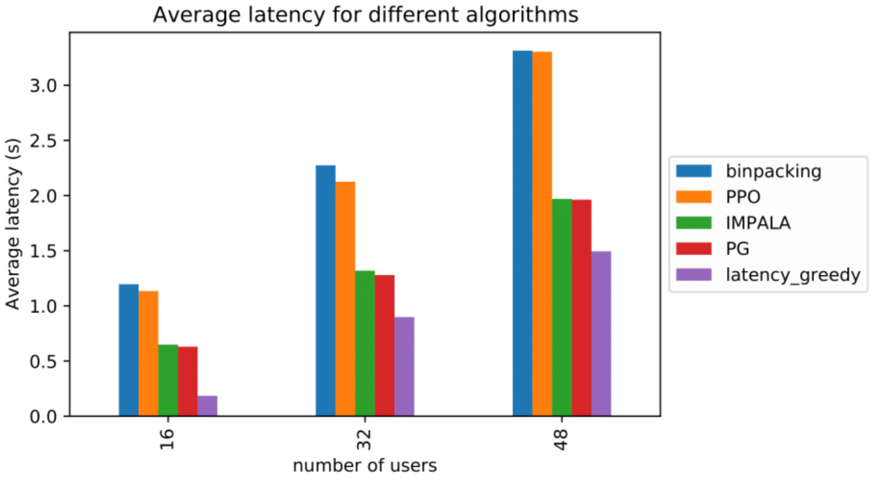
 # empty servers

Average network latency

Results - Test

- Average over 20 sample episode run
- On the cluster instead of the simulator



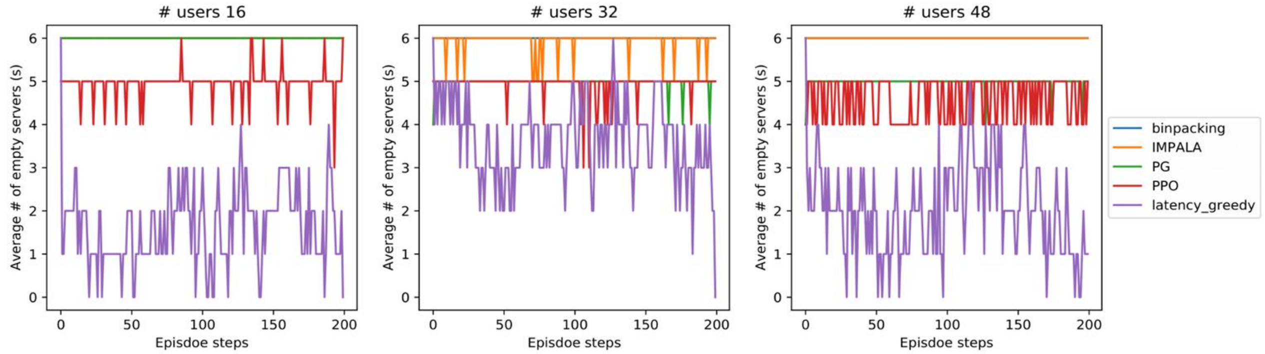


 # empty servers

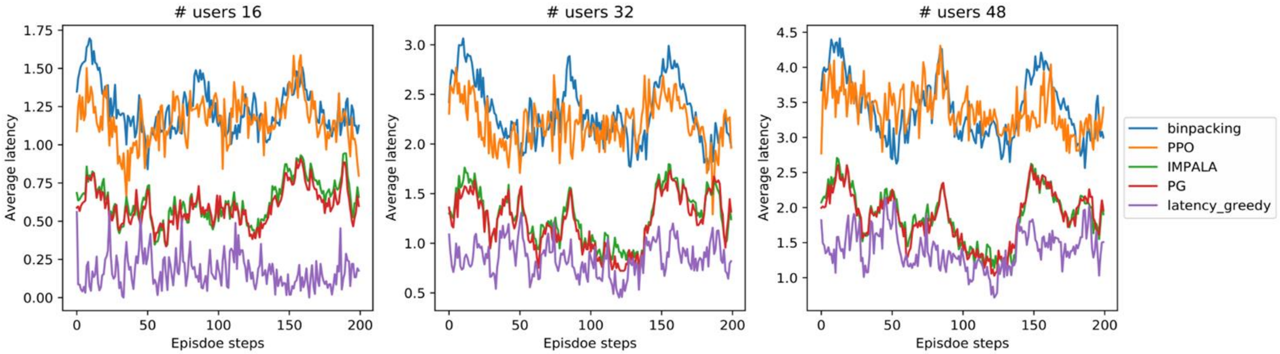
Average latency

Results - Example episode

- Single episode run per timestep



 # empty servers



Average network latency

Directions for future works

- Checkpointing of stateful services
- Kubernetes full implementation
- Hierarchical and multi-agent RL

Thank you for your attention!

- Source code available at: https://github.com/saeid93/mobile-kube.git
- Currently under review in Transactions of Service Computing
- Early version of work https://ieeexplore.ieee.org/abstract/document/9284153
- Email: j.doyle@qmul.ac.uk