USENIX ATC '24

Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu

Qingyuan Liu, Yanning Yang, Dong Du,

Yubin Xia, Ping Zhang, Jia Feng, James Larus, Haibo Chen

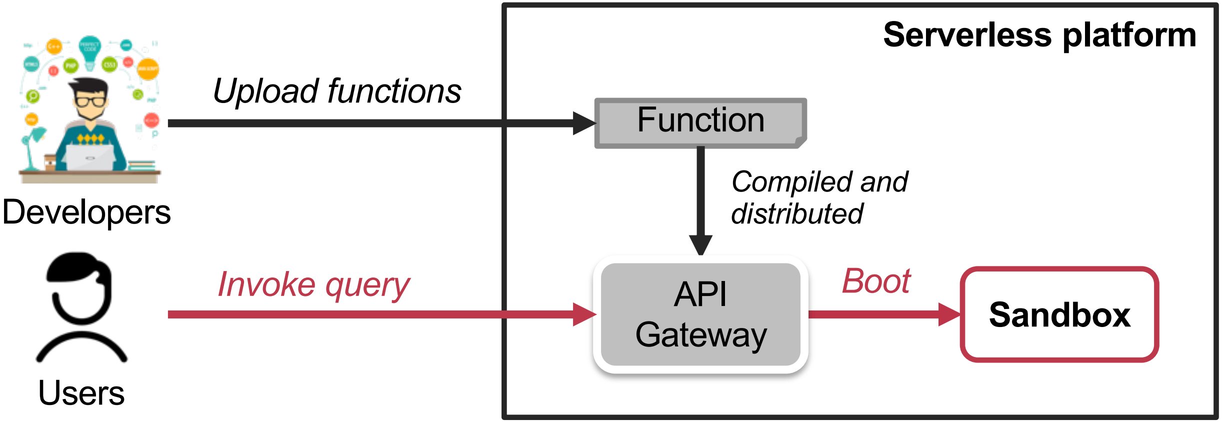
IPADS, SJTU; Huawei Cloud; EPFL

饮 水 思 源 • 爱 国 荣 校

BACKGROUND & MOTIVATION

Serverless Computing is Popular

- **Popular cloud paradigm**
  - Users upload the code and platforms are responsible for dev/ops.



Serverless platform

Developers

Upload functions

Function

Compiled and distributed

Users

Invoke query

API Gateway

Boot

Sandbox

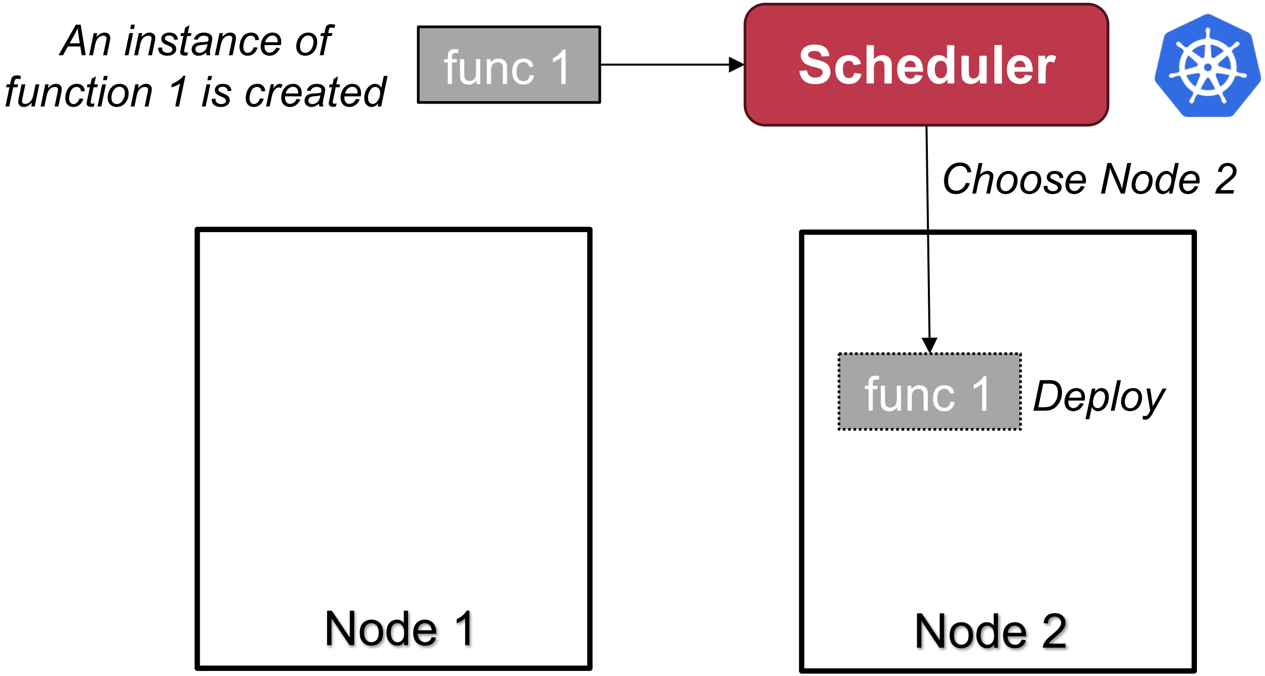
AWS Lambda

Microsoft Azure

Huawei FunctionGraph

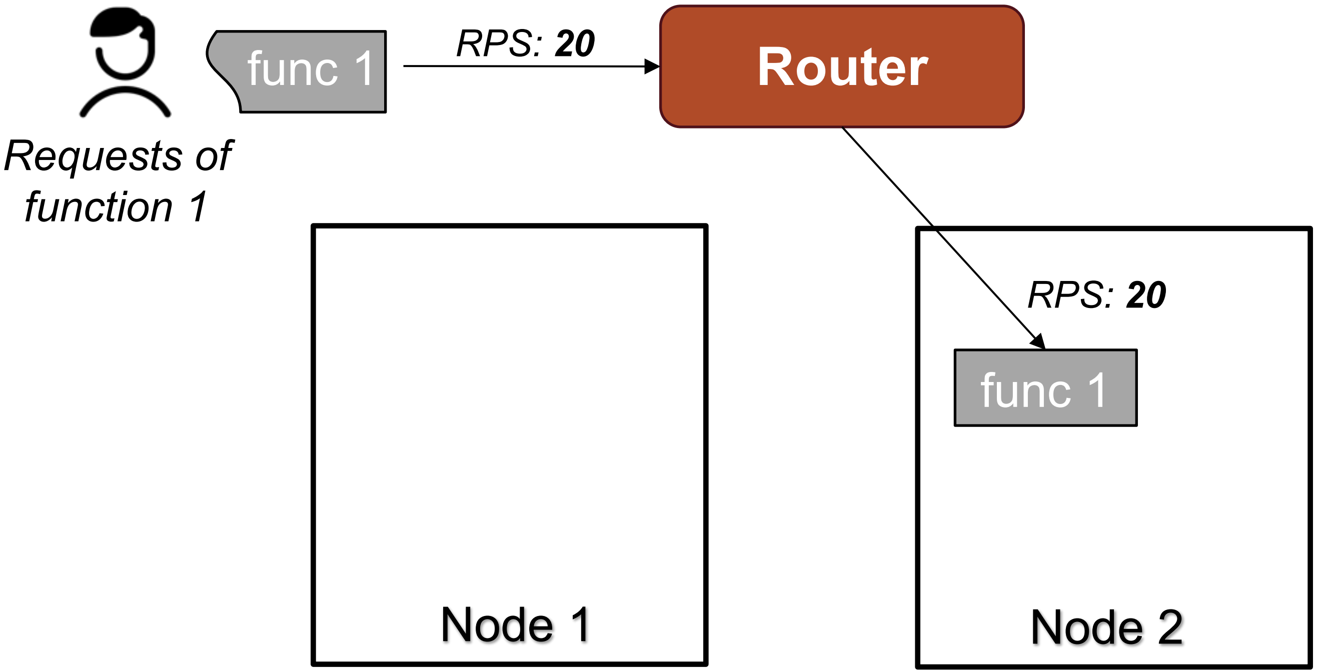
Knative

Key Components of Serverless Systems: Scheduler



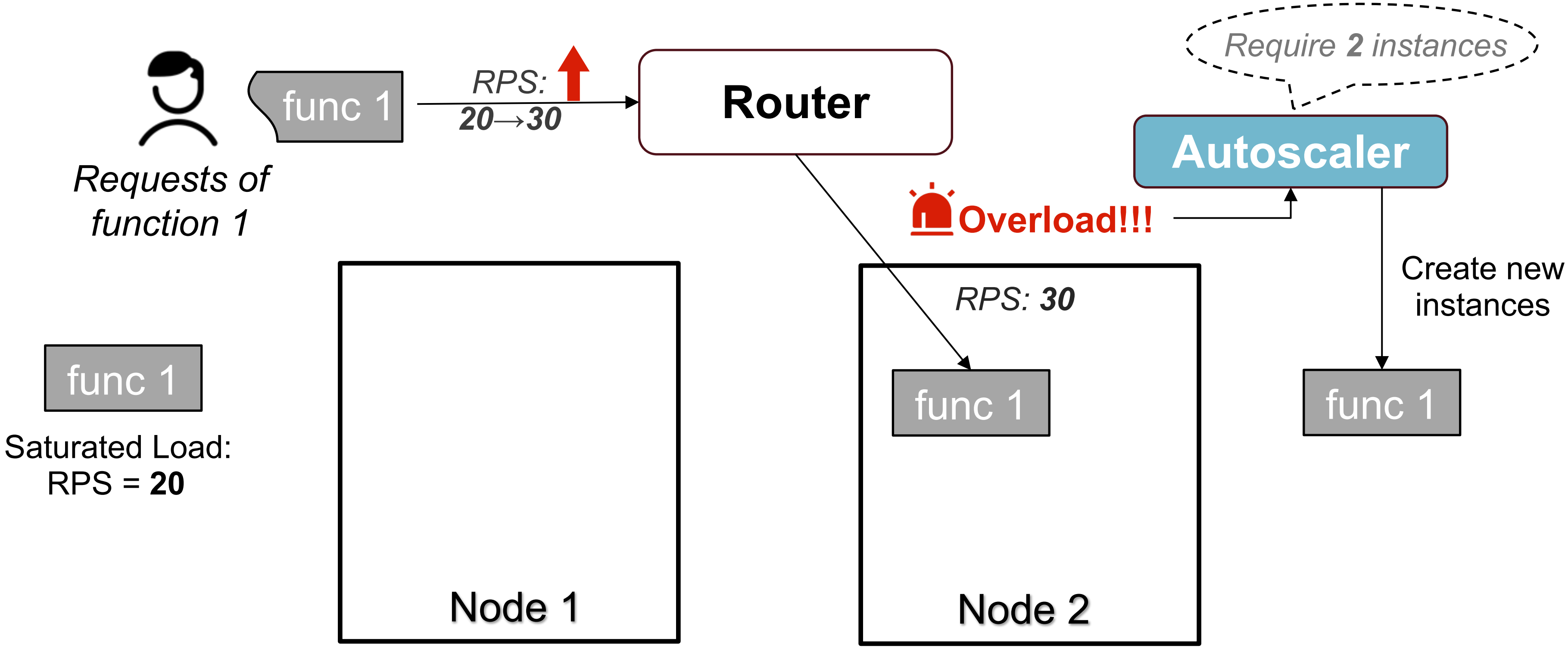
Scheduler: assigning each instance to a right server

Key Components of Serverless Systems: Router



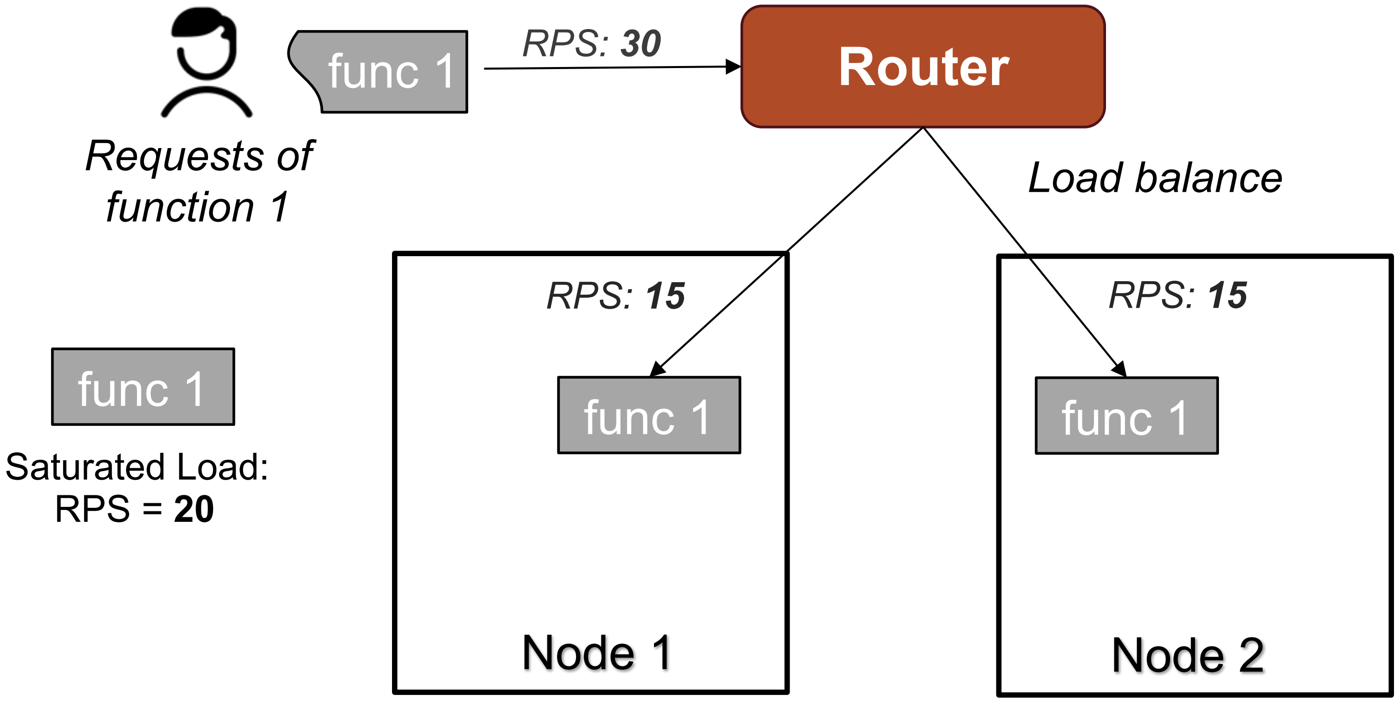
Router: distribute requests to specific instances

Key Components of Serverless Systems: Autoscaler

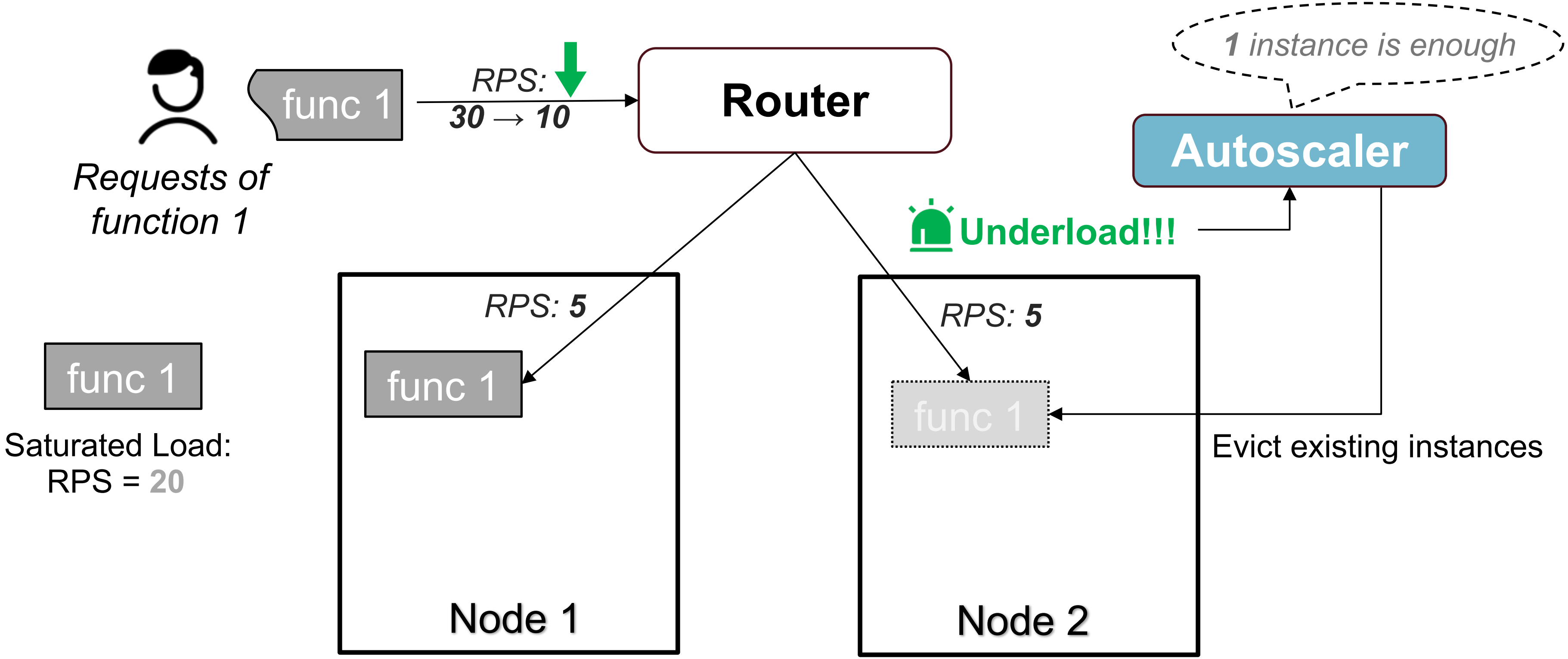


Autoscaler: scaling instances according to user loads

Key Components of Serverless Systems



Key Components of Serverless Systems: Autoscaler

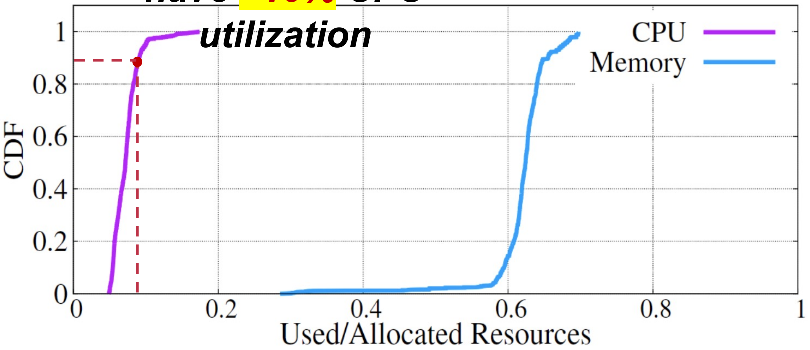


Autoscaler: scaling instances according to user loads

Low Resource Utilizations for Serverless Computing

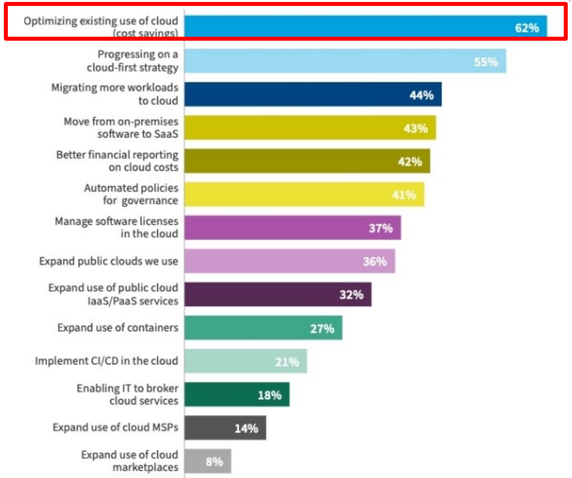
Resources are **under-utilized** in serverless computing

***~90%* nodes have *<10%* CPU utilization**



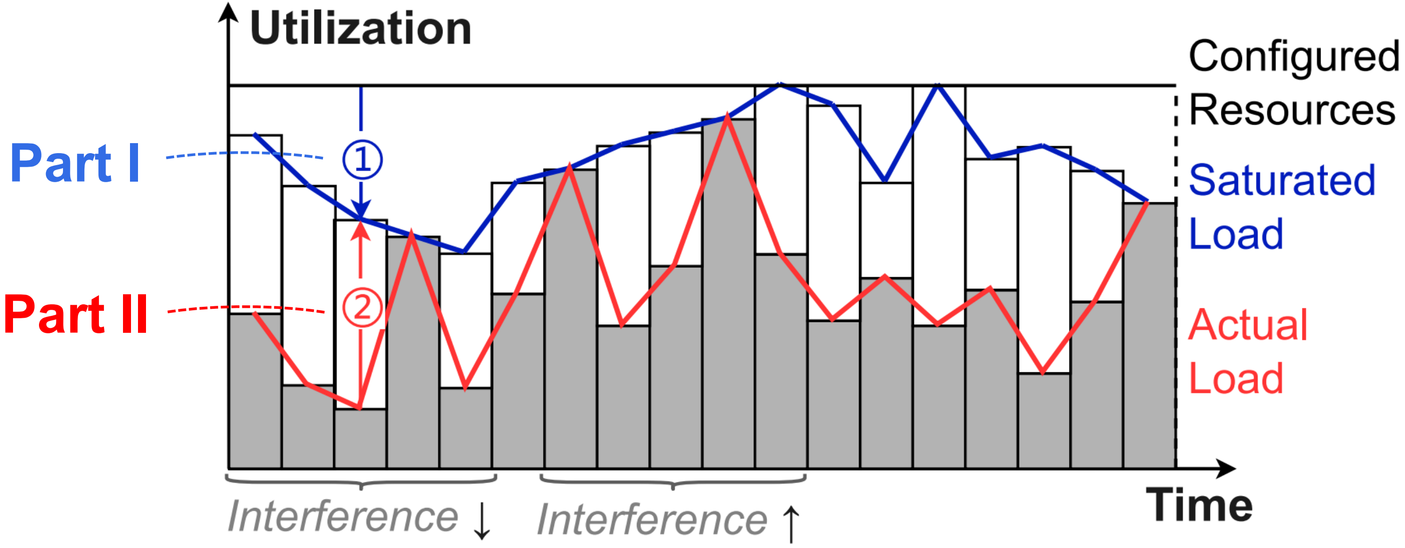
Cost saving is the mostly concerned issue for most organizations (62%)

Which of the following initiatives are you planning to make progress on in the next year?



“+1% resource util, Billons of $ saved”

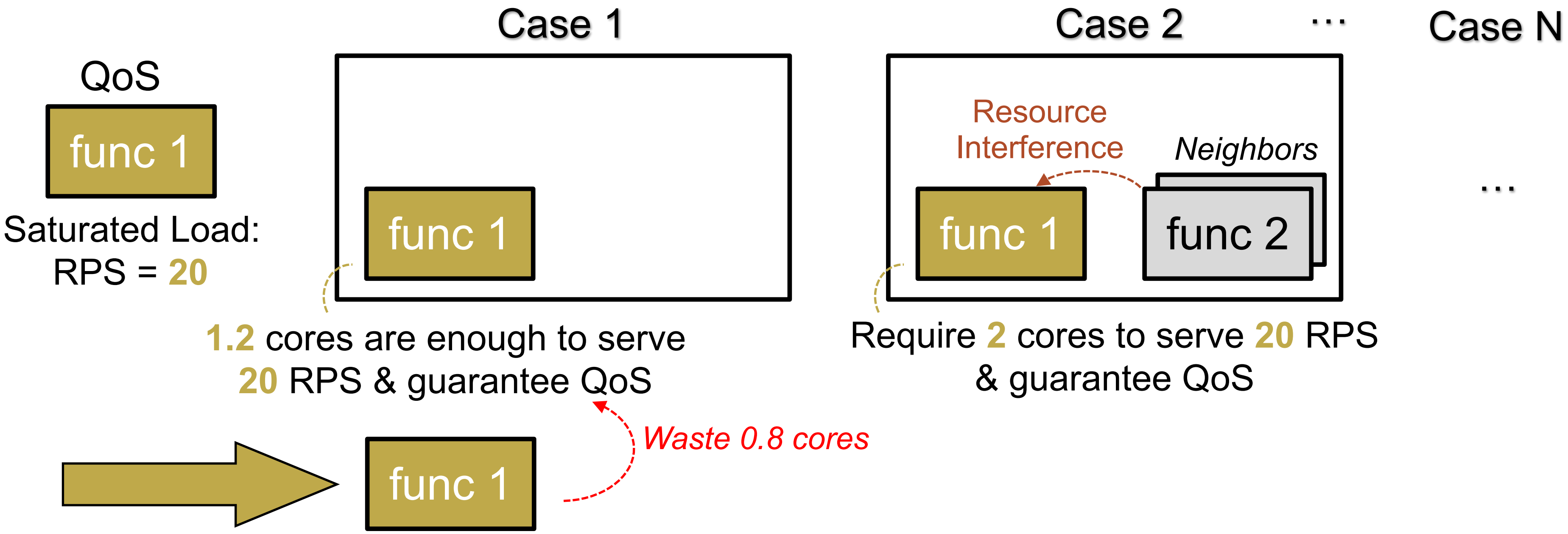
Identify Two Causes of Resource Wastage



Part I: caused by **resource overprovisioning**

Part II: caused by **load overestimation**

Wastage Part I: Resource Overprovisioning



QoS

func 1

Saturated Load:
RPS = 20

Case 1

func 1

1.2 cores are enough to serve
20 RPS & guarantee QoS

Case 2

Resource
Interference

Neighbors

func 1

func 2

Require 2 cores to serve 20 RPS
& guarantee QoS

…

Case N

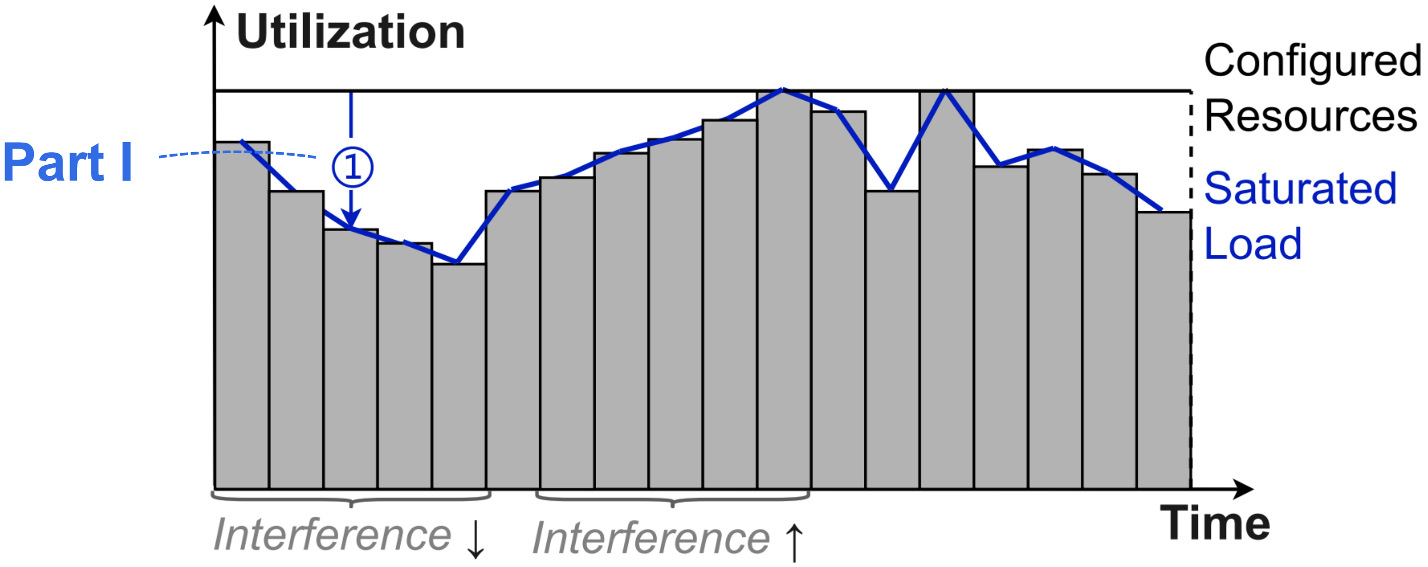
…

func 1

Waste 0.8 cores

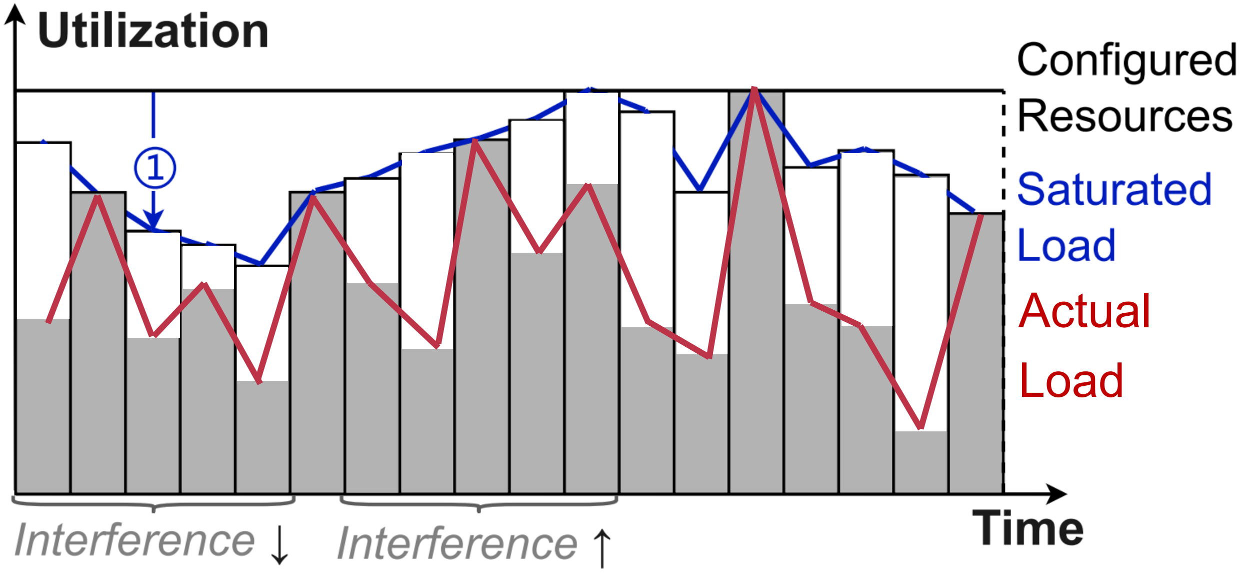
Conservative user configuration to
guarantee QoS for **both Case 1&2**:
Allocate 2 cores

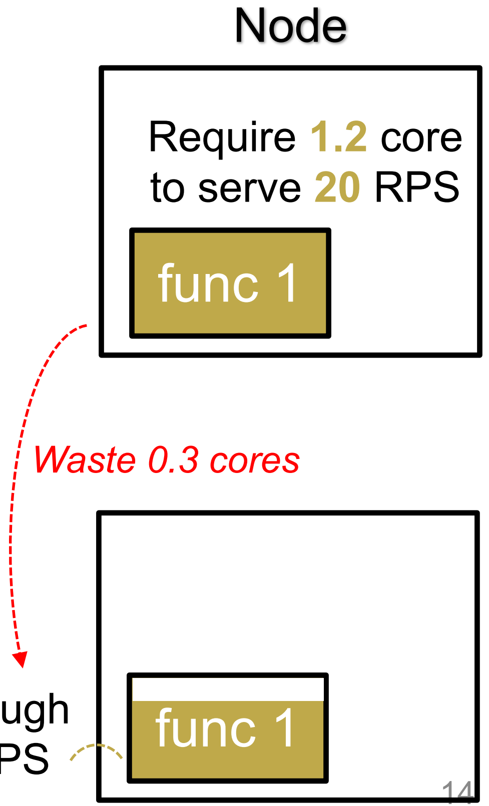
Wastage Part I: Resource Overprovisioning



Part I: resources are **overprovisioned** even for *saturated* instances

Wastage Part II: Load Overestimation



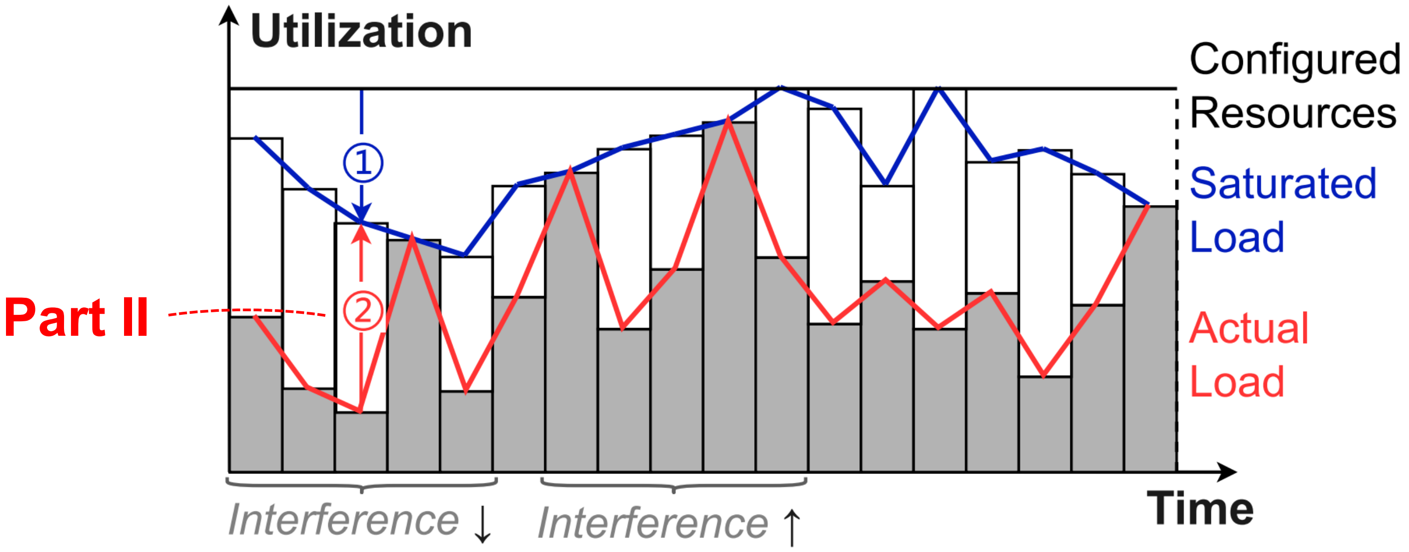


Node
Require 1.2 core to serve 20 RPS
func 1
Waste 0.3 cores
0.9 core is enough to serve 15 RPS
func 1

Unpredictable load fluctuation causes

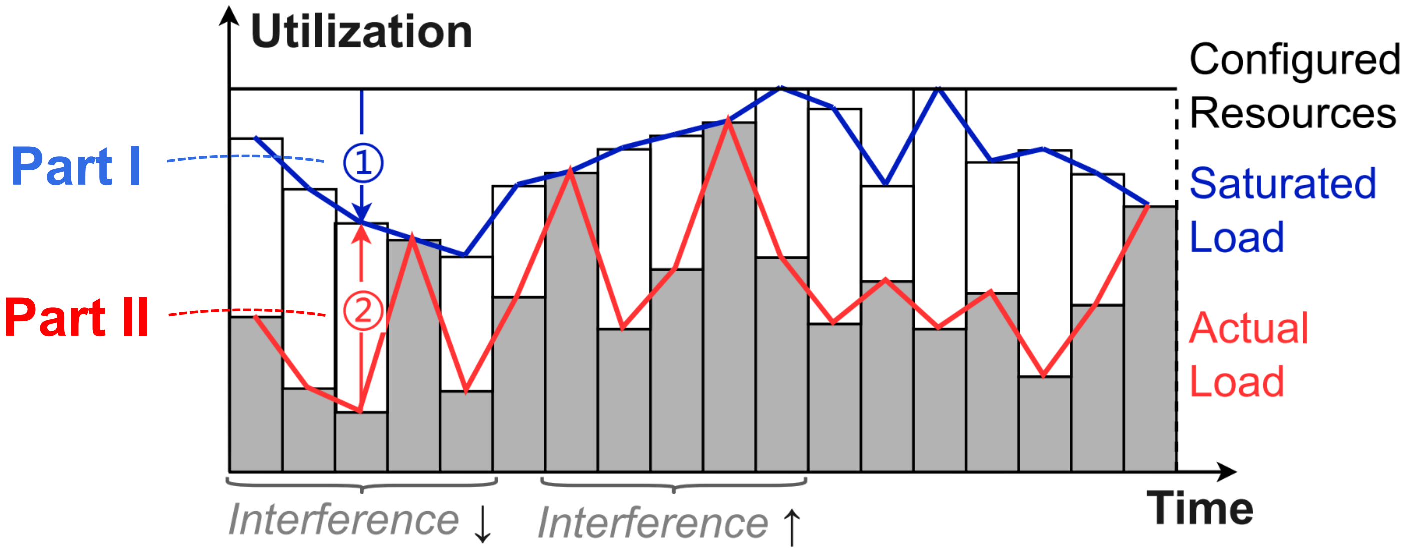
load overestimation

Wastage Part II: Load Overestimation



Part II: resources are **overestimated** due to load fluctuation

Challenges to Mitigate the Two Parts of Wastage

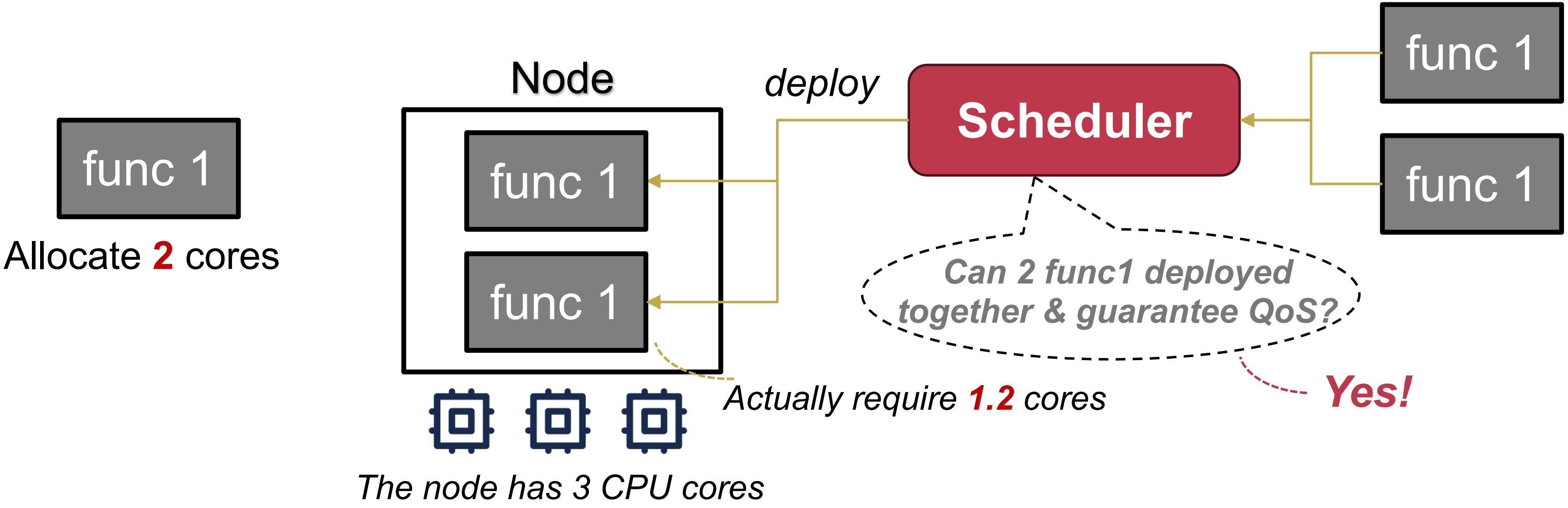


Challenges for prior methods:

Tradeoffs between high effectiveness & low cost

Mitigate Wastage Part I: Overcommitment

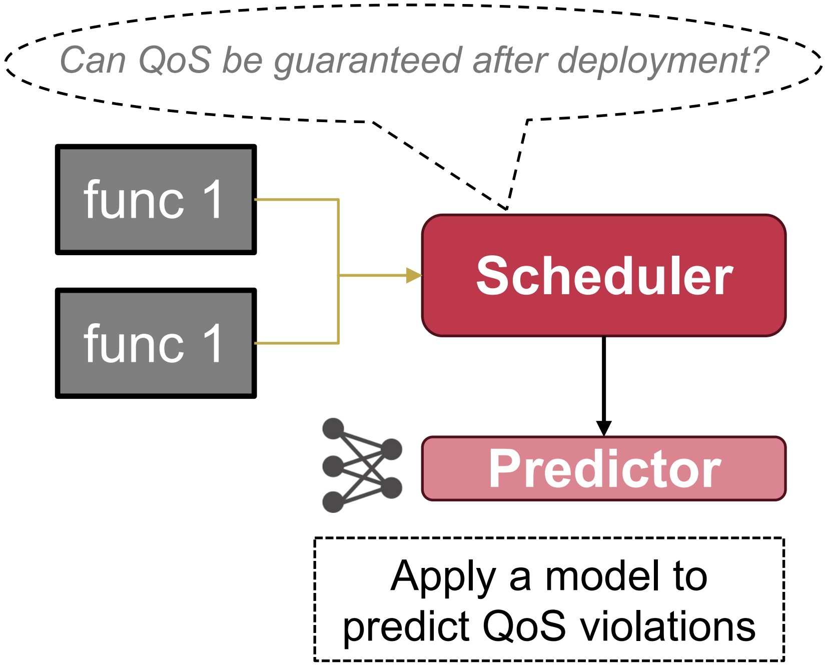
Overcommitment: increase deployment density



Requirement:

the scheduler should **accurately predict QoS violations**

Challenges of Overcommitment



Can QoS be guaranteed after deployment?

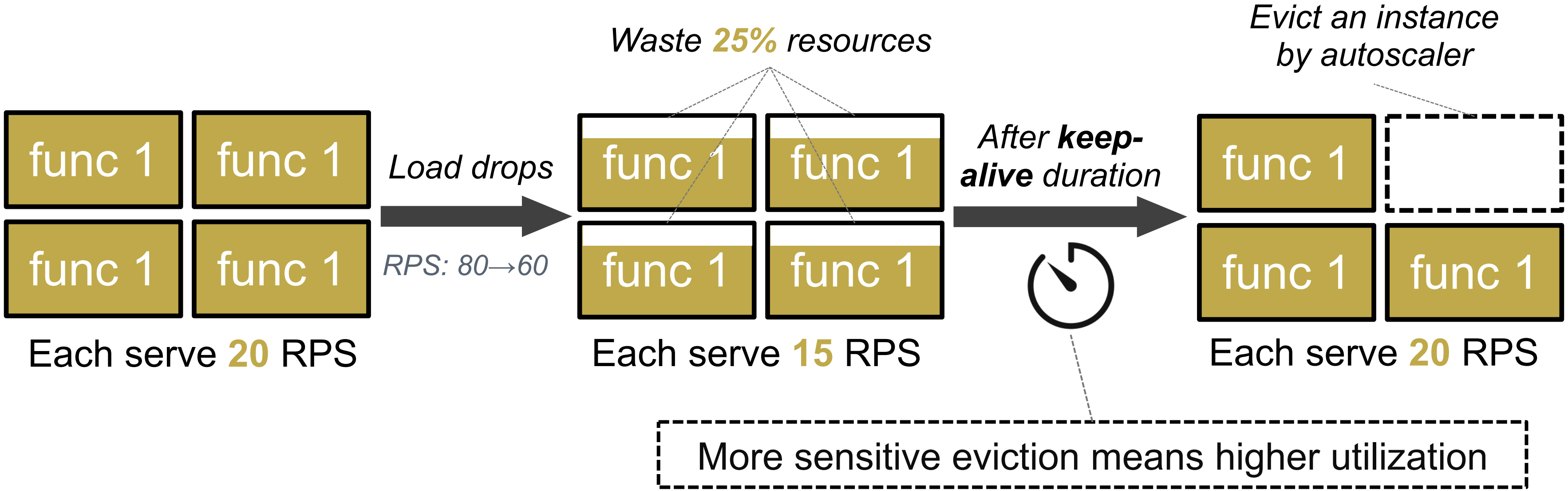
Apply a model to predict QoS violations

- Predict with <u>complex</u> models
  - Accurate prediction 👍
  - Costly (>tens of ms) 👎
- Predict with <u>heuristic</u> models or <u>historical information</u>
  - Inaccurate prediction or unscalable profilings 👎
  - Fast (~1ms) 👍

Challenge I:

Achieve <u>accurate prediction</u> & <u>practical cost (<10ms)</u> simultaneously

Mitigate Wastage Part II: Sensitive Autoscaling

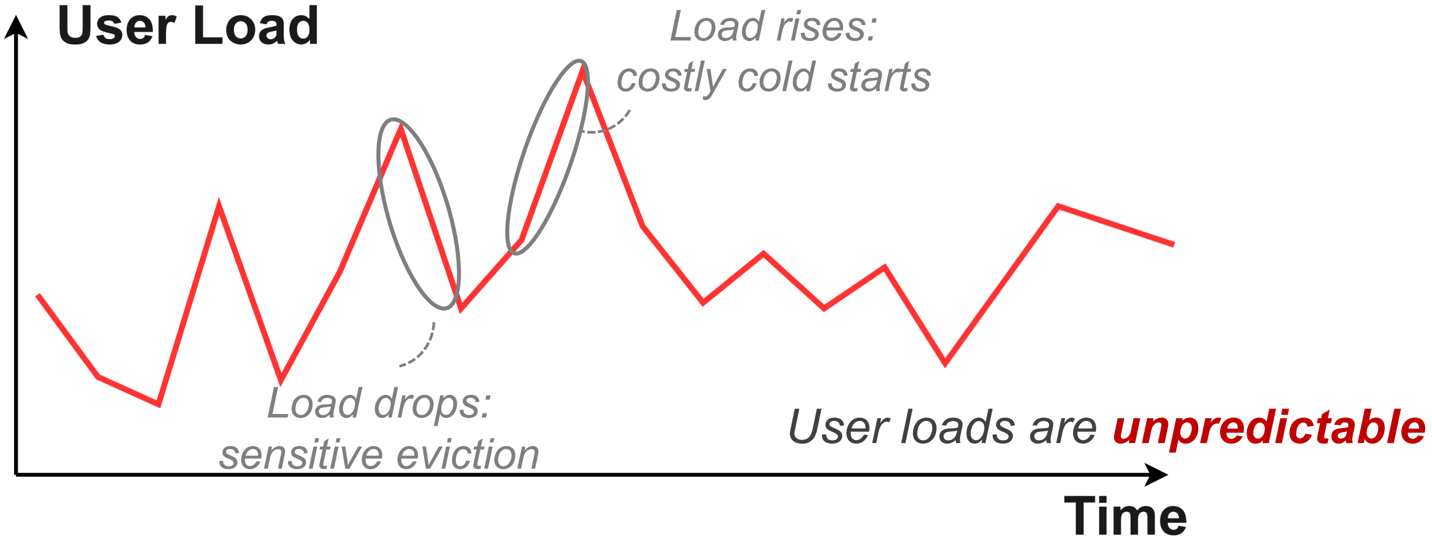


More sensitive eviction means higher utilization

Autoscaling: dynamically reclaim unused resources upon load drops

Challenges of Sensitive Autoscaling

Problem: more sensitive eviction could mean **more cold starts**

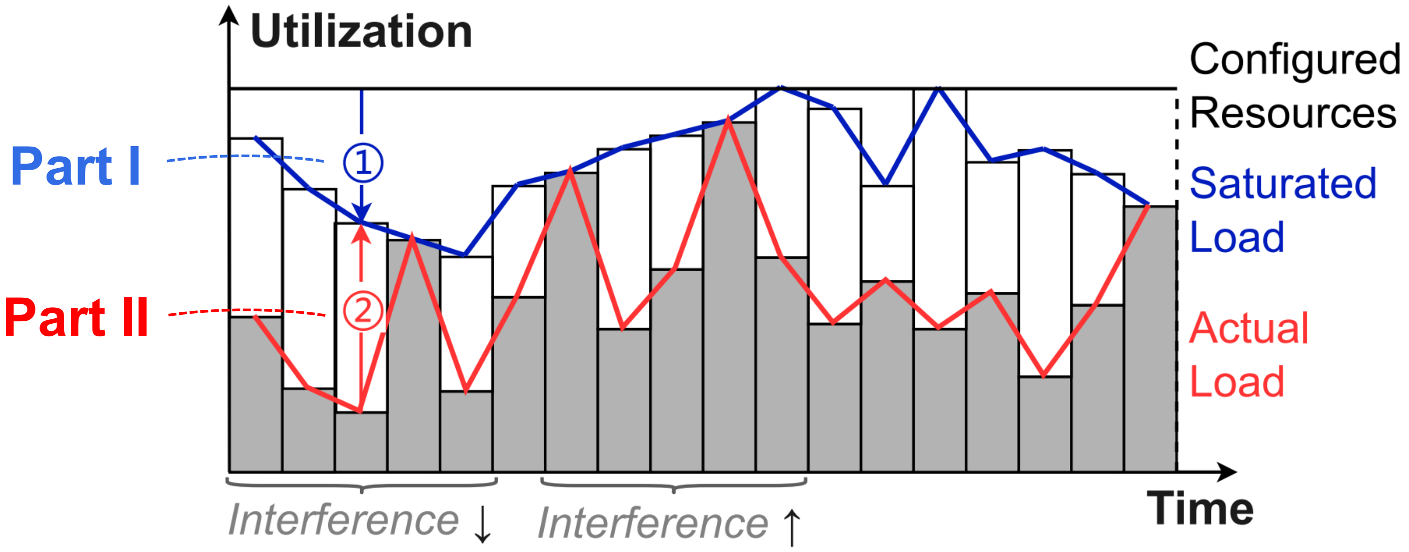


User loads are ***unpredictable***

Challenge II:

Achieve high utilization and low cold start costs simultaneously

Jiagu: Two Designs to Break the Tradeoffs



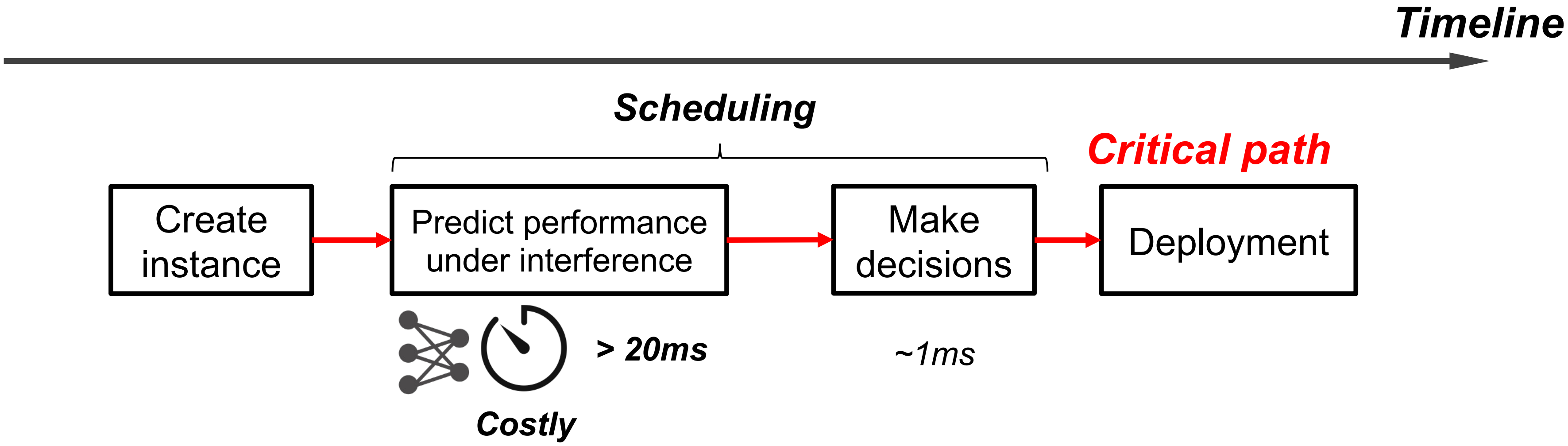
Design I for Part I: pre-decision scheduling

Design II for Part II: dual-staged scaling

1. Achieve both efficiency and performance for overcommitment

DESIGN I: PRE-DECISION SCHEDULING

Insight I: Decouple Prediction and Decision Making



Timeline

Scheduling

Critical path

Create instance

Predict performance under interference

Make decisions

Deployment

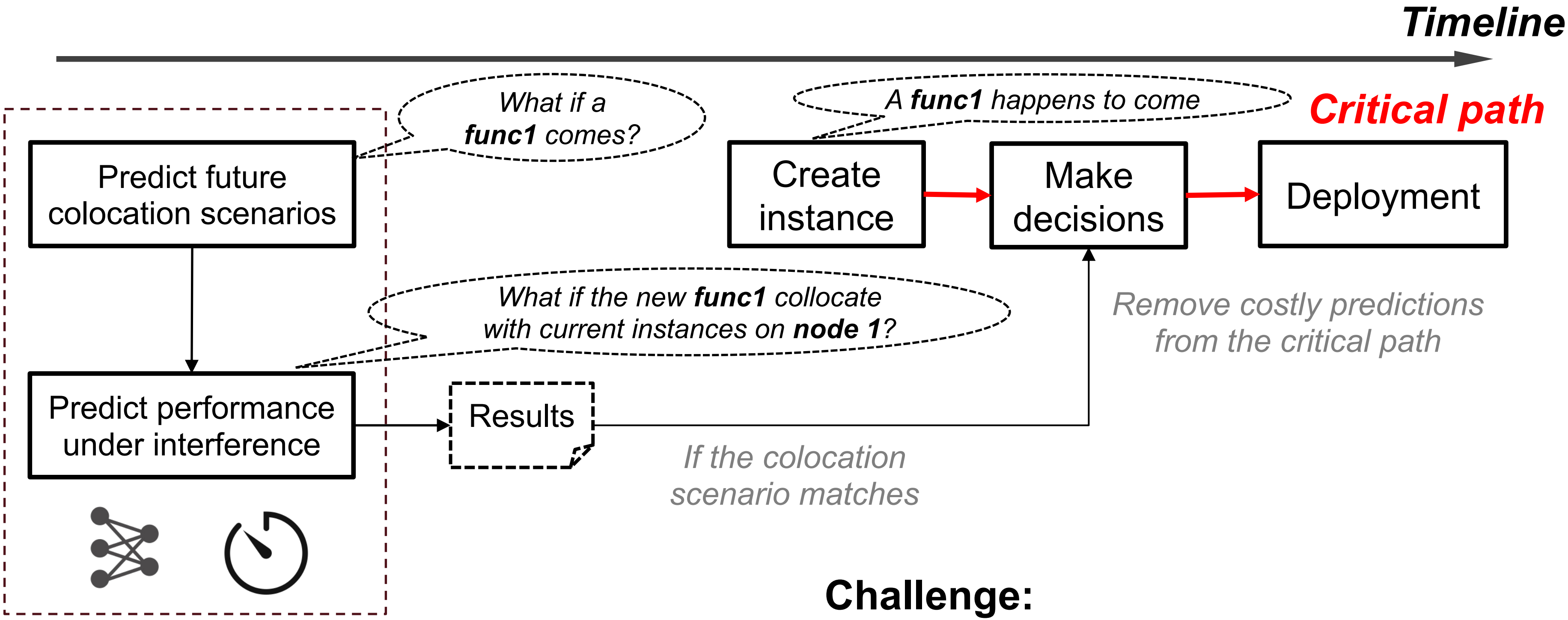
> 20ms

~1ms

Costly

Costly predictions are on the critical path

Insight I: Decouple Prediction and Decision Making



Timeline

Predict future colocation scenarios

*What if a **func1** comes?*

Predict performance under interference

*What if the new **func1** collocate with current instances on **node 1**?*

Results

If the colocation scenario matches

*A **func1** happens to come*

Create instance

Make decisions

Deployment

Critical path

Remove costly predictions from the critical path

Advance Prediction

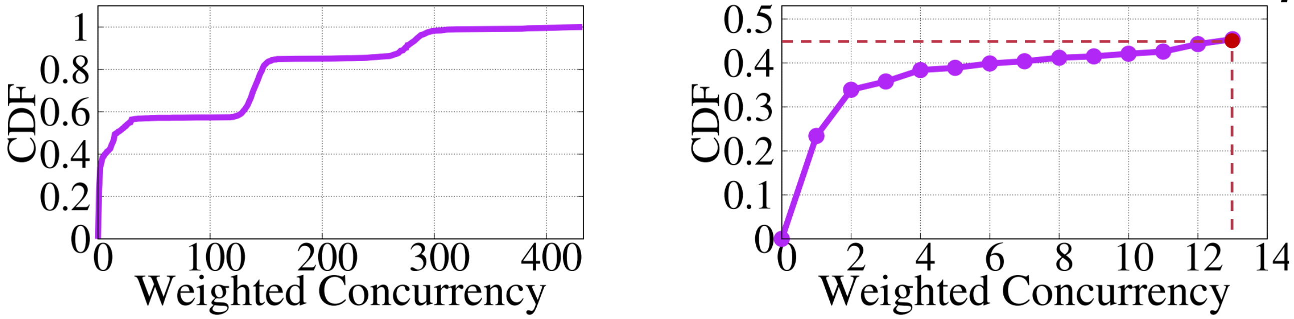
Challenge:

impossible to traverse all possible colocation environments

Serverless Highly-replicated Nature

Serverless instances are **highly replicated**

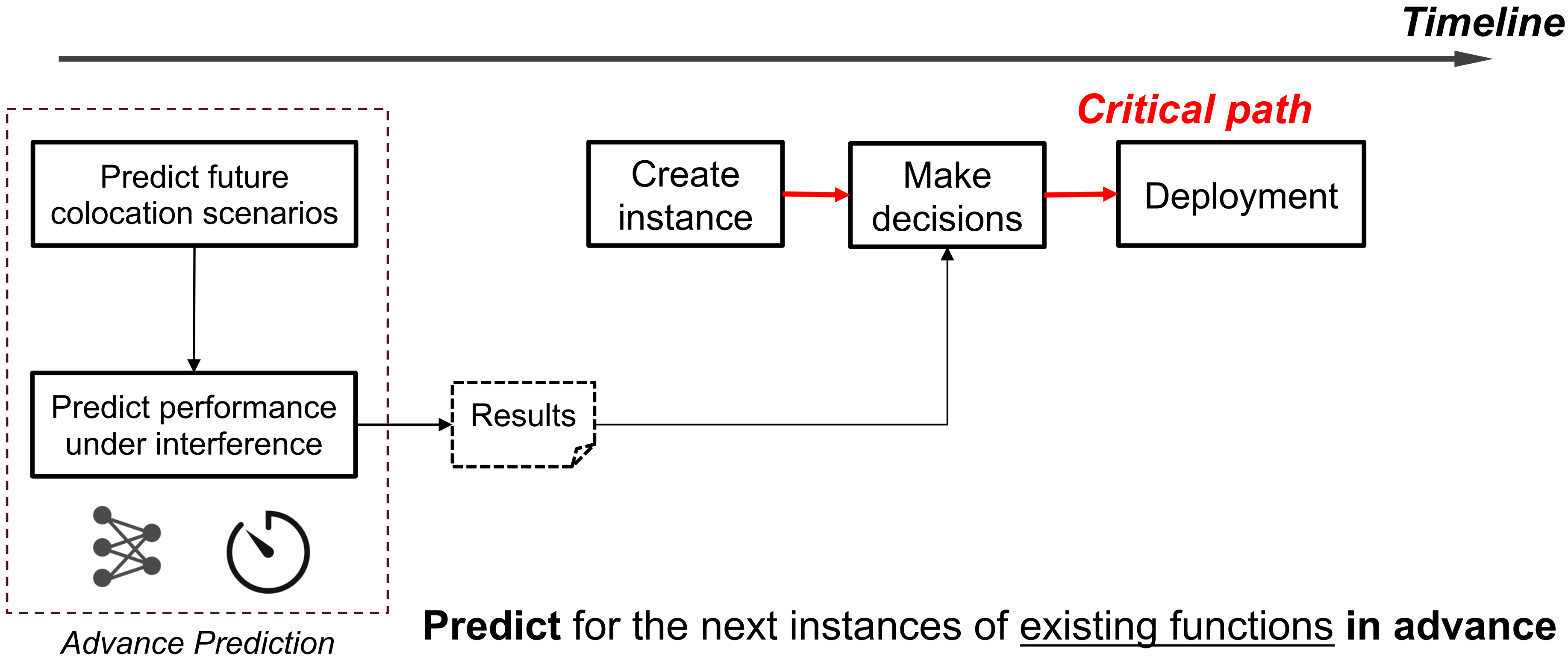
***~56%* instances are of functions that have *>12* replicated instances**



(a) Weighted instance concurrencies of functions.

(b) Weighted instance concurrencies (<13) of functions.

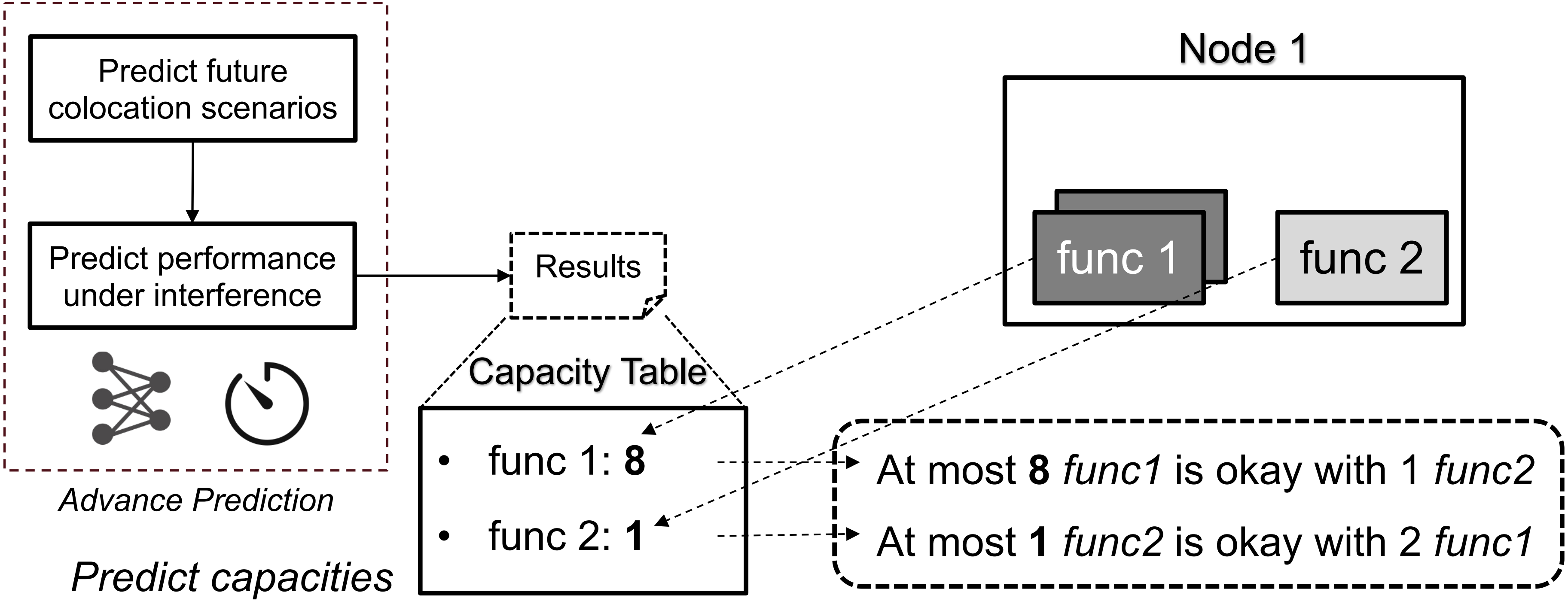
Insight I: Decouple Prediction and Decision Making



Predict for the next instances of existing functions **in advance**

Advance Prediction to Construct the Capacity Table

Predict for the next instances of existing functions **in advance**



Predict future colocation scenarios

Predict performance under interference

Advance Prediction

Results

Capacity Table

- func 1: **8**
- func 2: **1**

Node 1

func 1

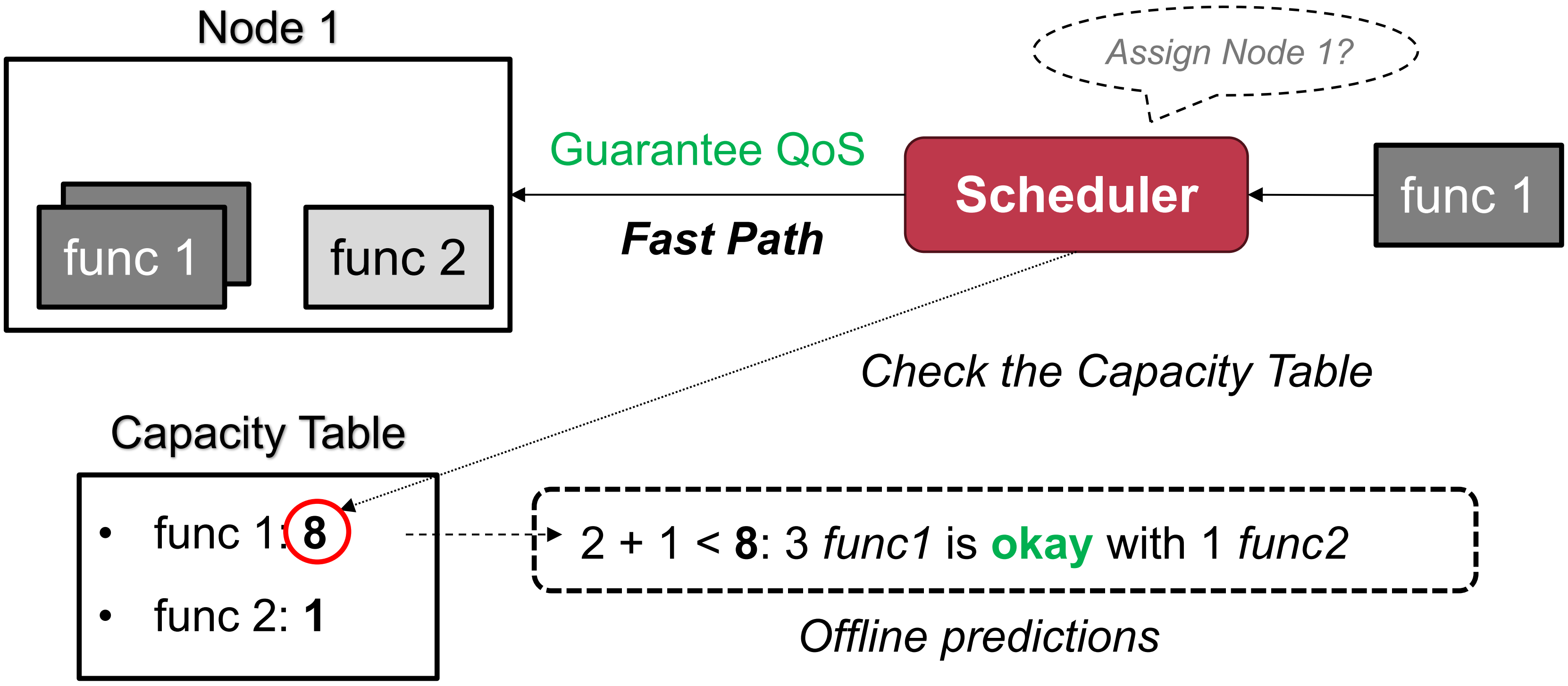
func 2

At most **8** *func1* is okay with 1 *func2*

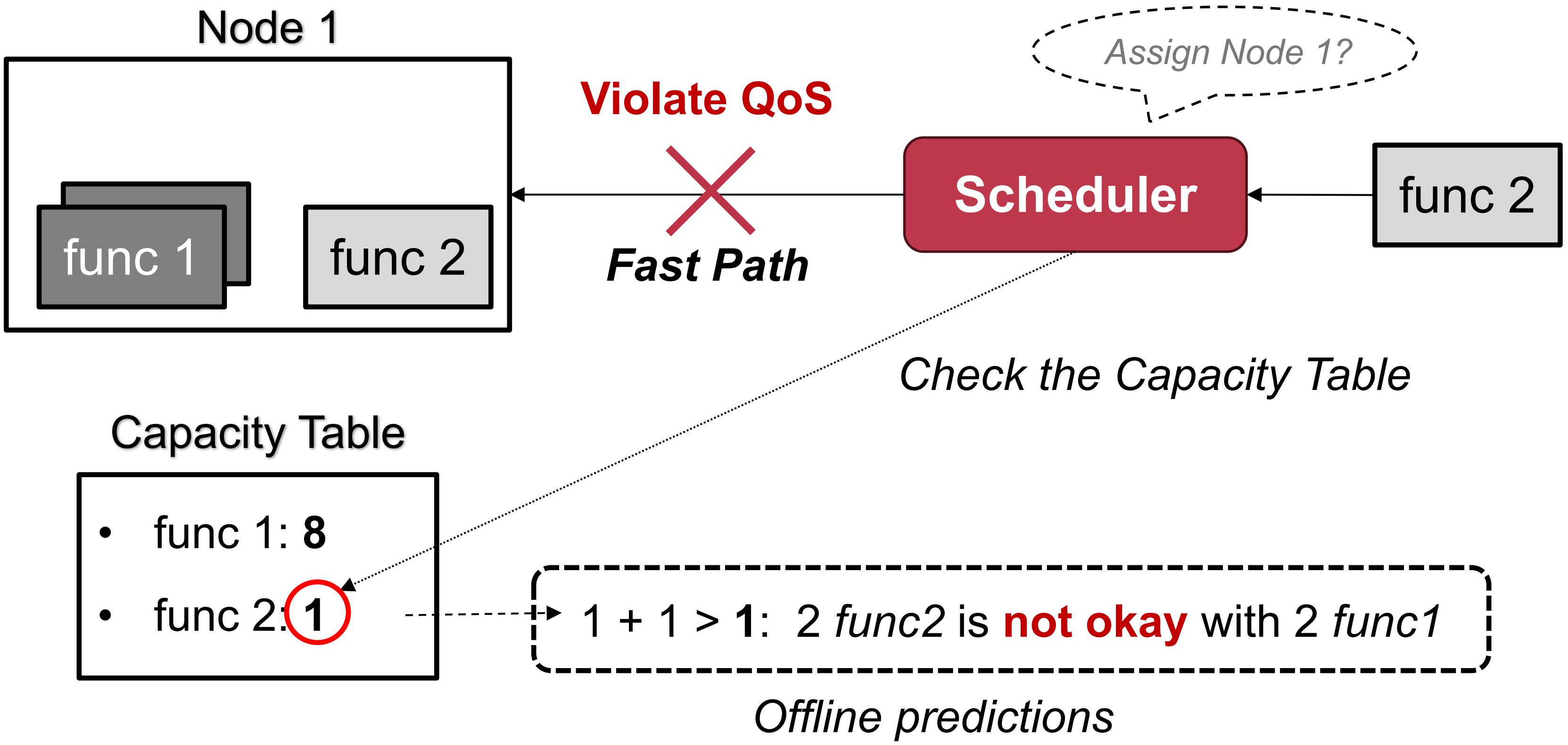
At most **1** *func2* is okay with 2 *func1*

Predict capacities (maximum concurrency)

Pre-decision Scheduling: Basic Idea



Pre-decision Scheduling: Basic Idea



Node 1

func 1

func 2

Assign Node 1?

Violate QoS

Fast Path

Scheduler

func 2

Check the Capacity Table

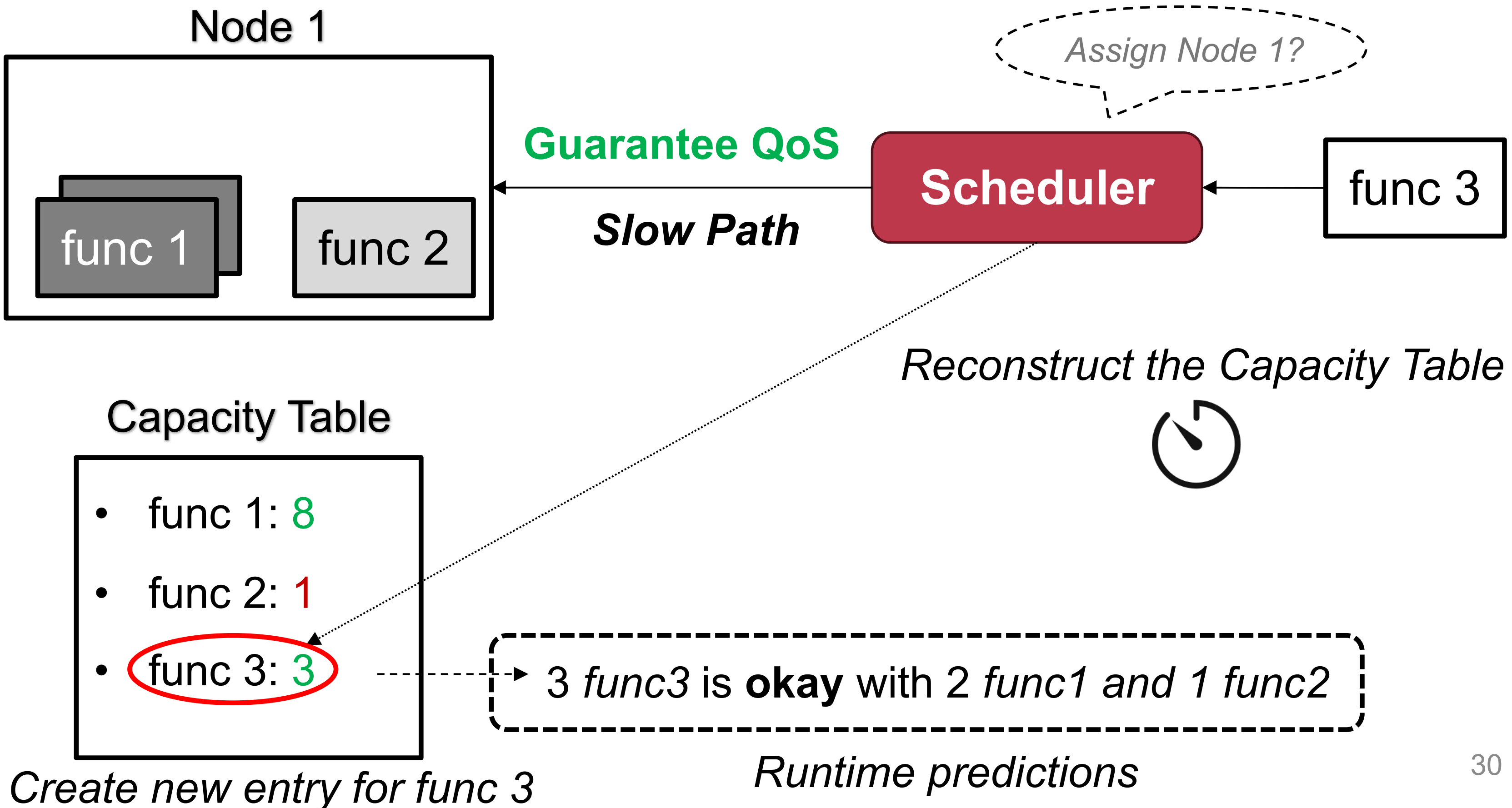
Capacity Table

- func 1: **8**
- func 2: **1**

1 + 1 > **1**: 2 *func2* is **not okay** with 2 *func1*

Offline predictions

Pre-decision Scheduling: Basic Idea



Node 1

func 1

func 2

Guarantee QoS

Slow Path

Assign Node 1?

Scheduler

func 3

Reconstruct the Capacity Table

Capacity Table

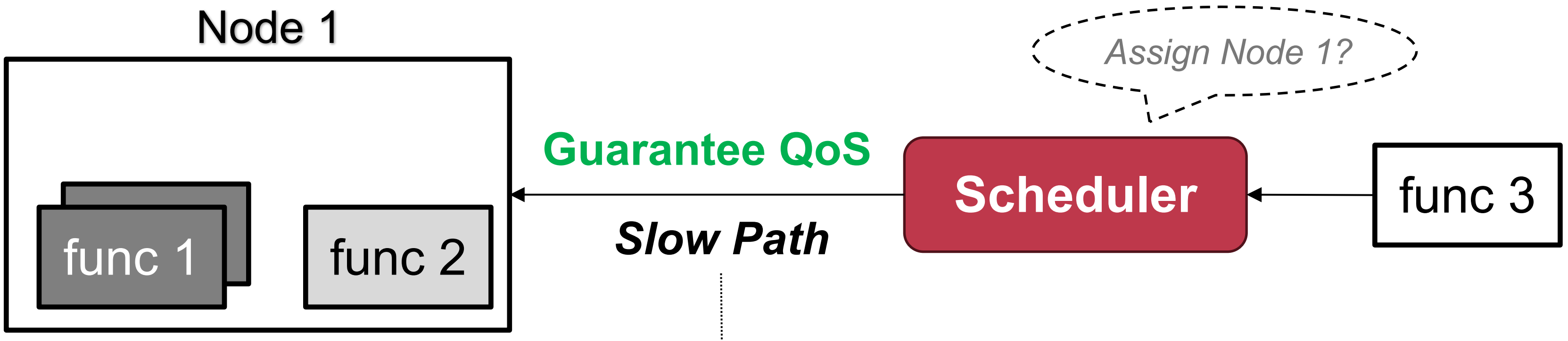
- func 1: 8
- func 2: 1
- func 3: 3

3 *func3* is **okay** with 2 *func1 and* 1 *func2*

Create new entry for func 3

Runtime predictions

Pre-decision Scheduling: Basic Idea



Highly-replicated nature:
number of **slow path << fast path**

Asynchronous Update

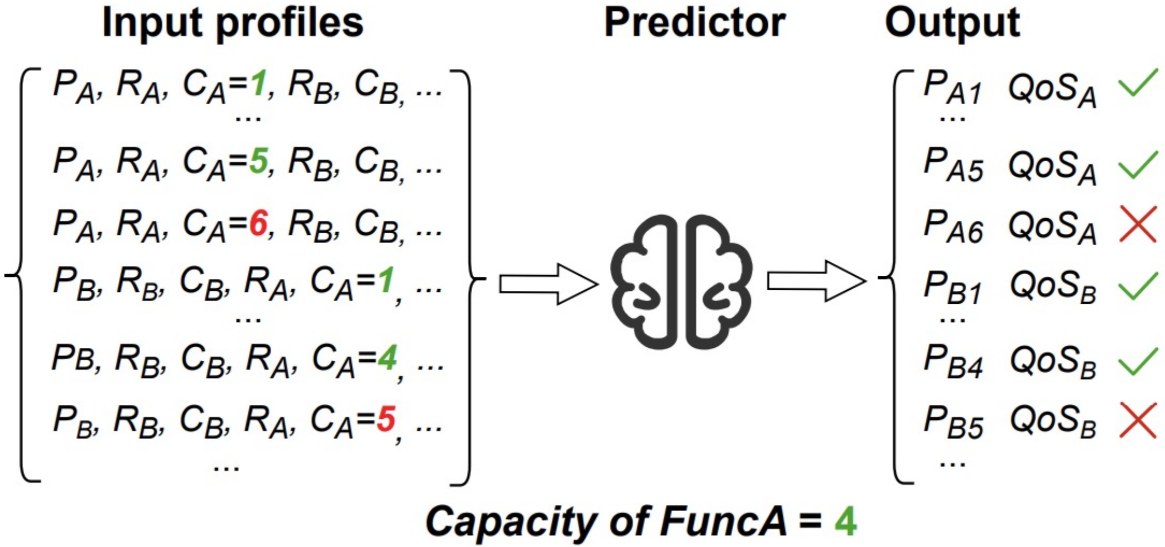
- **Capacity table:**
  - Require timely update: the colocation environment is constantly changing
- **Asynchronous update:**
  - Keep the capacity table up-to-date
  - Prevent the updating from introducing prediction overhead in the critical path

(Details in the paper)

Calculate the Capacity

$$P_{A\cup\{B,C,...\}} = RFR\{P_A, R_A, C_A, \overbrace{R_B, C_B, R_C}^{\text{Features of co-located functions}}, C_C, ...\}$$

Solo-run performance Profiles Concurrency



Input profiles

P_A, R_A, C_A=1, R_B, C_B, ...
...
P_A, R_A, C_A=5, R_B, C_B, ...
P_A, R_A, C_A=6, R_B, C_B, ...
P_B, R_B, C_B, R_A, C_A=1, ...
...
P_B, R_B, C_B, R_A, C_A=4, ...
P_B, R_B, C_B, R_A, C_A=5, ...
...

Predictor

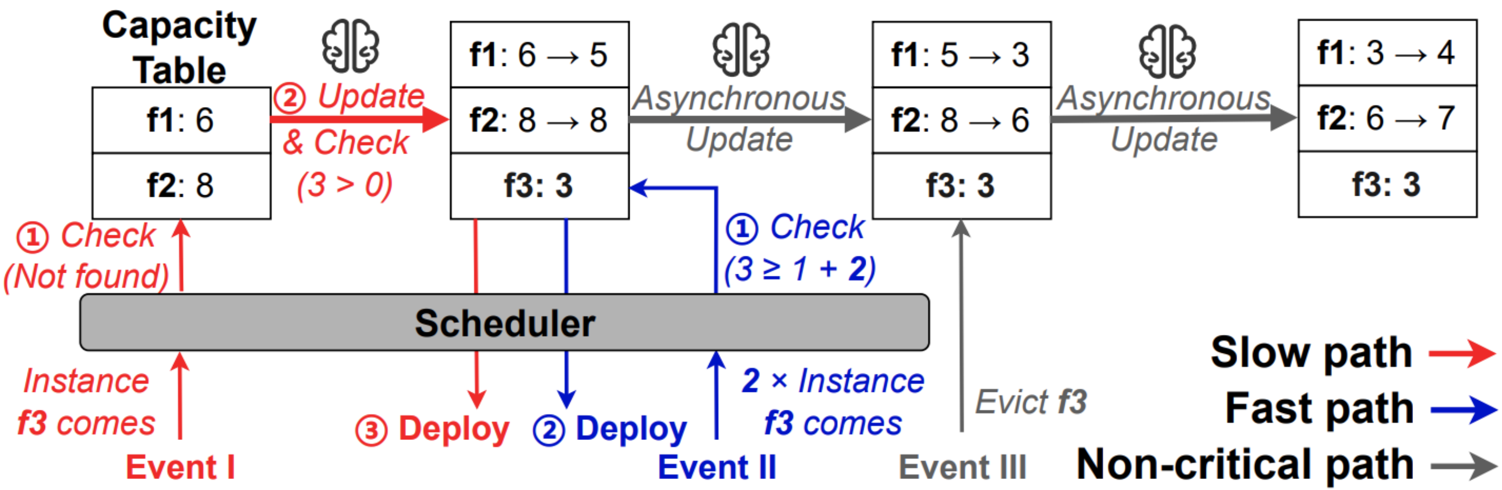
Output

P_{A1} QoS_A ✓
...
P_{A5} QoS_A ✓
P_{A6} QoS_A ✗
P_{B1} QoS_B ✓
...
P_{B4} QoS_B ✓
P_{B5} QoS_B ✗
...

Capacity of FuncA = 4

Find the **maximum QoS-guaranteed concurrency:**
(Details in the paper)

Pre-decision Scheduling: Put It All Together

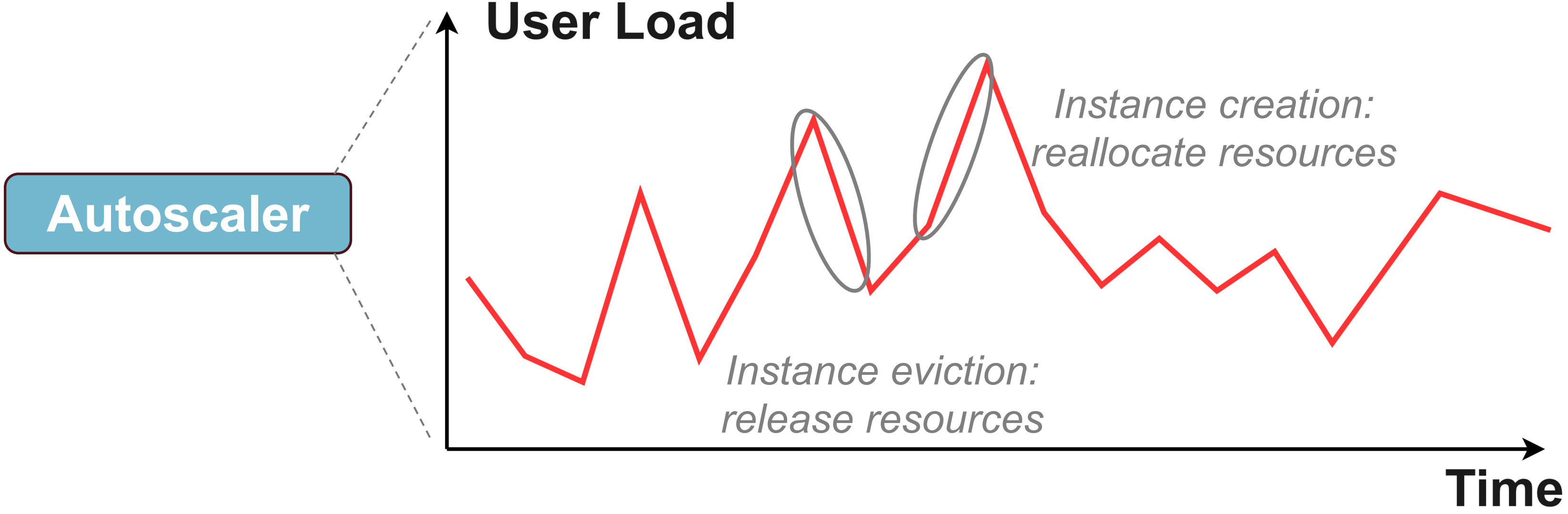


(Details in the paper)

2. Achieve sensitive autoscaling for high utilization without additional cold starts

DESIGN II: DUAL-STAGED SCALING

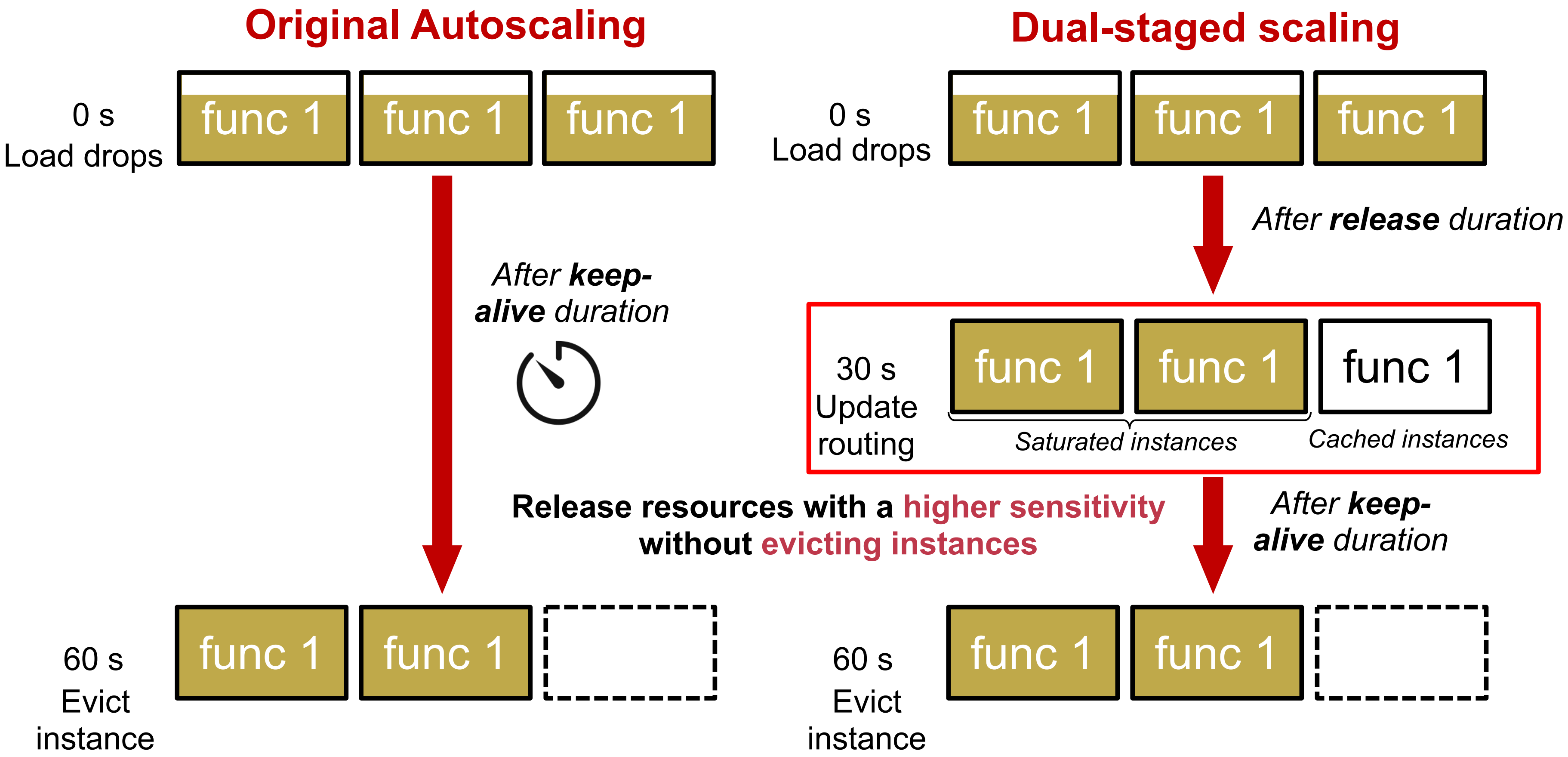
Insight 2: Decouple Resource Releasing and Instance Eviction



Root cause of overheads:

The coupling between **resource allocation/release** and **instance creation/eviction**

Dual-staged Scaling: Basic Idea



Original Autoscaling

0 s
Load drops

func 1 func 1 func 1

*After **keep-alive** duration*

60 s
Evict
instance

func 1 func 1

Dual-staged scaling

0 s
Load drops

func 1 func 1 func 1

*After **release** duration*

30 s
Update
routing

func 1 func 1 func 1

Saturated instances

Cached instances

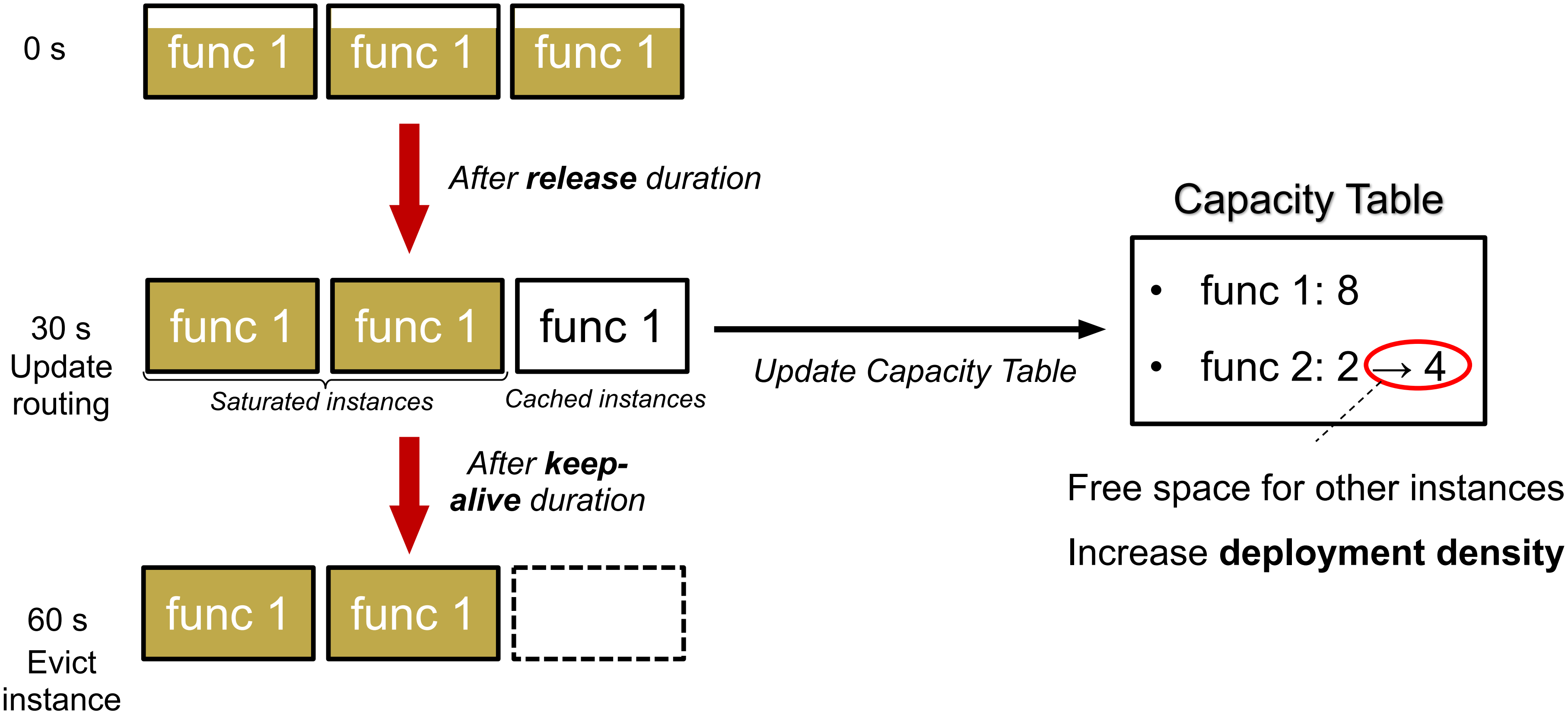
*After **keep-alive** duration*

60 s
Evict
instance

func 1 func 1

Release resources with a higher sensitivity without evicting instances

Dual-staged Scaling: Basic Idea



0 s

func 1 func 1 func 1

*After **release** duration*

30 s Update routing

func 1 func 1 func 1

Saturated instances

Cached instances

Update Capacity Table

Capacity Table

- func 1: 8
- func 2: 2 → 4

*After **keep-alive** duration*

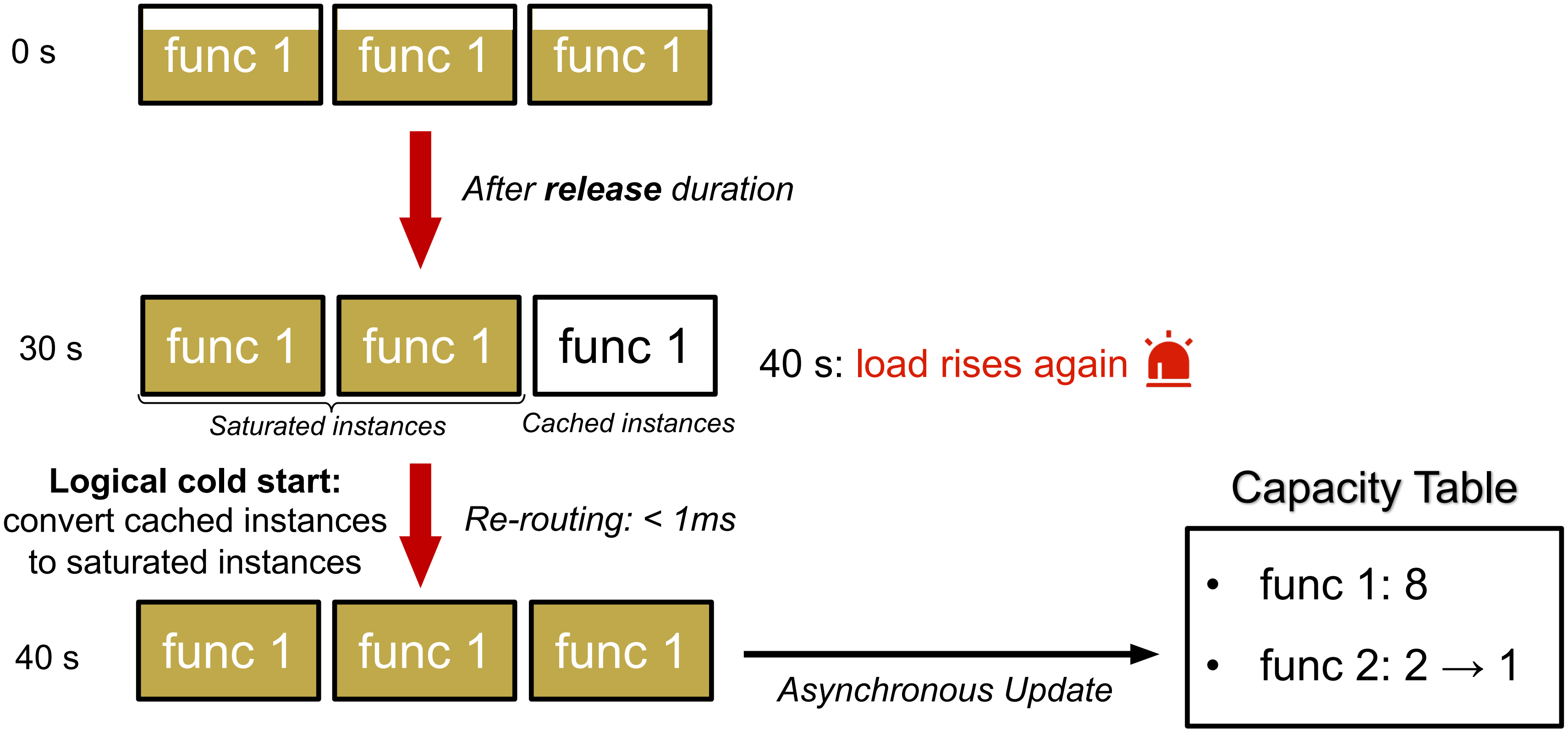
60 s Evict instance

func 1 func 1

Free space for other instances

Increase **deployment density**

Dual-staged Scaling: Logical Cold Start



0 s

func 1 func 1 func 1

*After **release** duration*

30 s

func 1 func 1 func 1

Saturated instances

Cached instances

40 s: load rises again 🚨

Logical cold start:
convert cached instances
to saturated instances

Re-routing: < 1ms

40 s

func 1 func 1 func 1

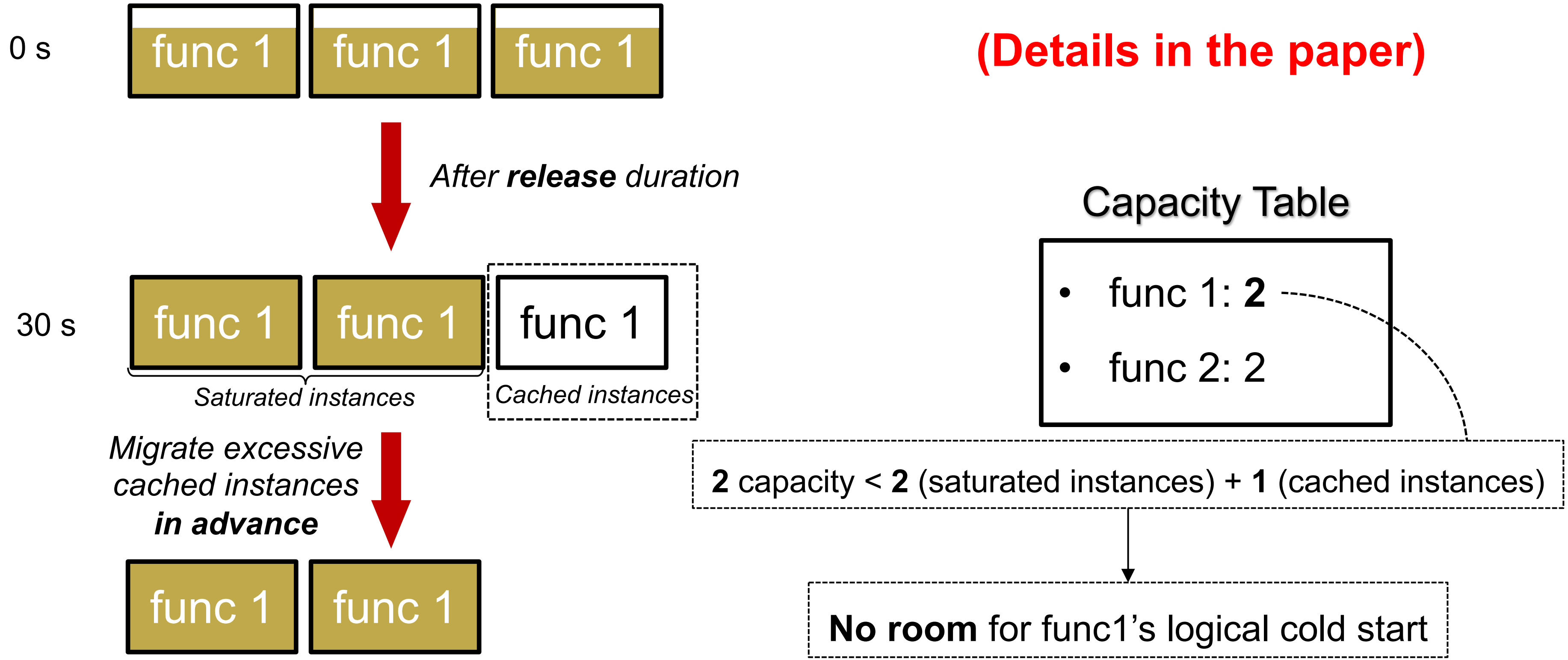
Asynchronous Update

Capacity Table

- func 1: 8
- func 2: 2 → 1

Dual-staged Scaling: On demand Migration

(Details in the paper)



Capacity Table

- func 1: **2**
- func 2: 2

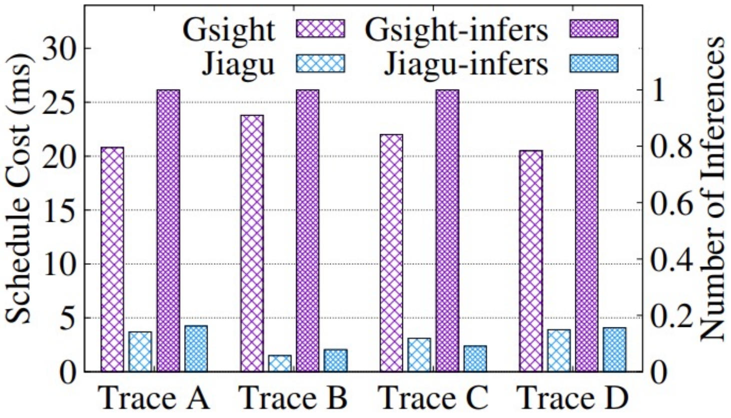
2 capacity < **2** (saturated instances) + **1** (cached instances)

No room for func1’s logical cold start

EVALUATION

Evaluation: Effective Scheduling with **Practical Cost**

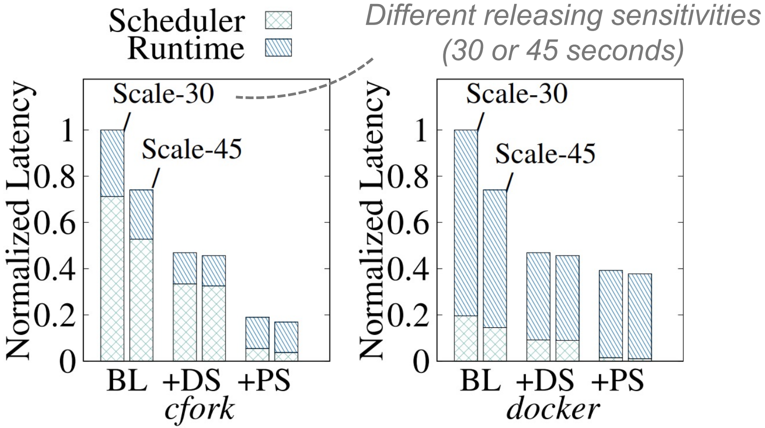
Optimize scheduling costs



Reduce **83.8%–92.1%** inferences on critical path

81.0%–93.7% lower scheduling costs

Optimize total cold start costs



Different releasing sensitivities (30 or 45 seconds)

Dual-staged Scaling (DS):

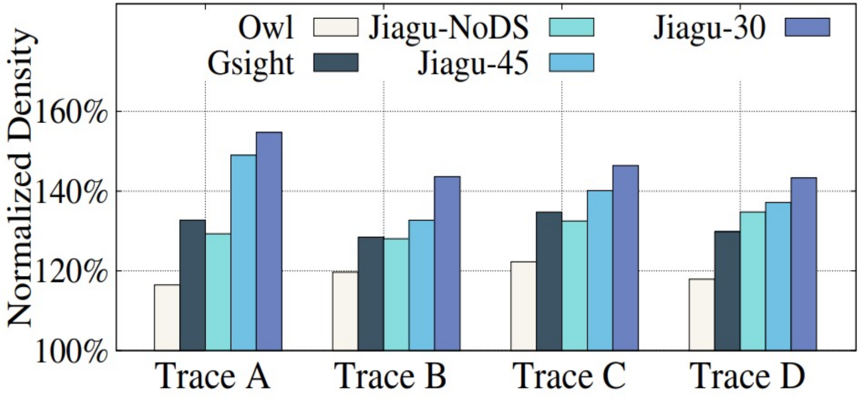
reduce the number of cold starts

Pre-decision Scheduling (PS):

reduce the cost of each cold start

Evaluation: Effective Scheduling with Practical Cost

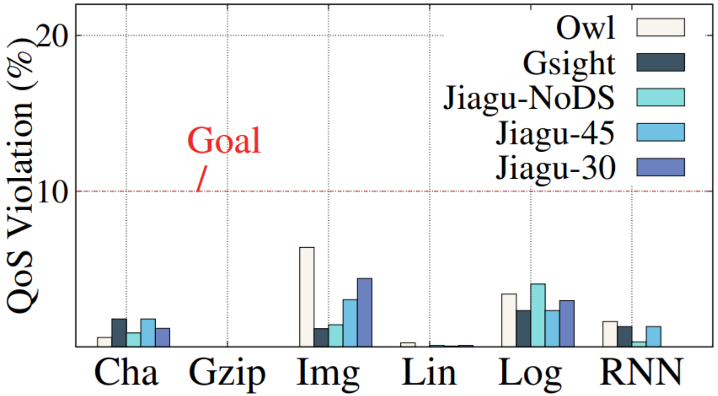
Optimize resource utilization



Up to **22%** higher deployment density than Gsight

38.3% higher deployment density than Owl

Ensuring QoS with accurate prediction



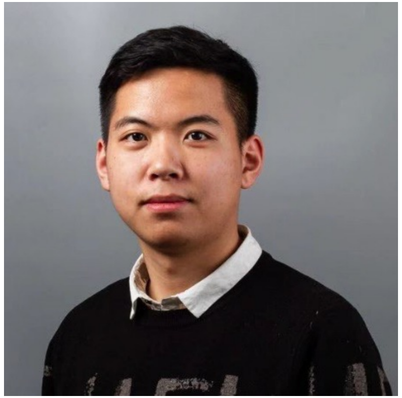
QoS violation rate meets the goal

Conclusion

- **Jiagu: optimize resource utilizations of serverless platforms**
- **Pre-decision scheduling: reduce resource overprovisioning**
  - **Overcommitment:** effective scheduling with accurate performance prediction
  - Reduce the prediction cost in the scheduling critical path
- **Dual-staged scaling: reduce load overestimation**
  - Achieve high resource utilization with sensitive autoscaling
  - Eliminate the side effect of incurring additional cold starts

Q&A / Contact Us

- **Qingyuan Liu**
  - PhD student, IPADS Lab, SJTU
  - liu_qy@sjtu.edu.cn
  - Personal Website (Chinese):
    - https://ipads.se.sjtu.edu.cn/zh/pub/members/qingyuan_liu/



- **Dong Du**
  - Assistant Professor, IPADS Lab, SJTU
  - dd_nirvana@sjtu.edu.cn
  - Personal Website:
    - https://dongd.info/