Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters

Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, Minyi Guo

Department of Computer Science and Engineering, Shanghai Jiao Tong University

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Content

Introduction & Background

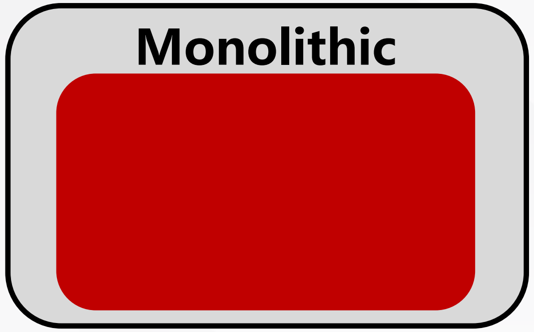
Shifting to Microservices

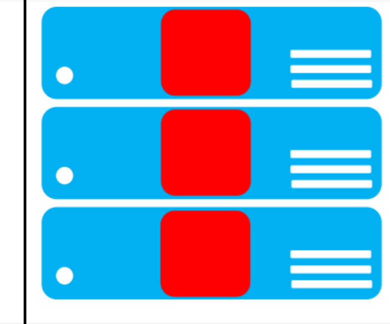
Datacenters host user-facing applications with the QoS target.

Traditional applications

Coupled & Central

Monolithic





- X Complex app
- X Centralized deployment
- X Unified programming language
- X ……

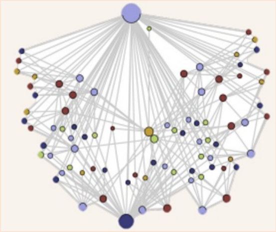
Shifting to

Distributed

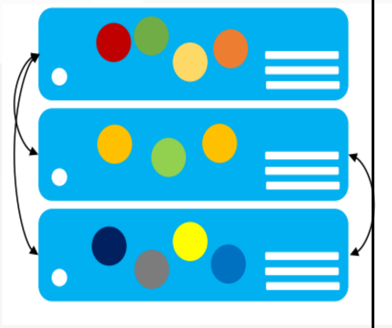
Microservice applications

Decoupled & Distributed

Microservices



- ✓ Multiple microservices
- ✓ Fine-grained management
- ✓ Heterogeneous program languages
- ✓ ……

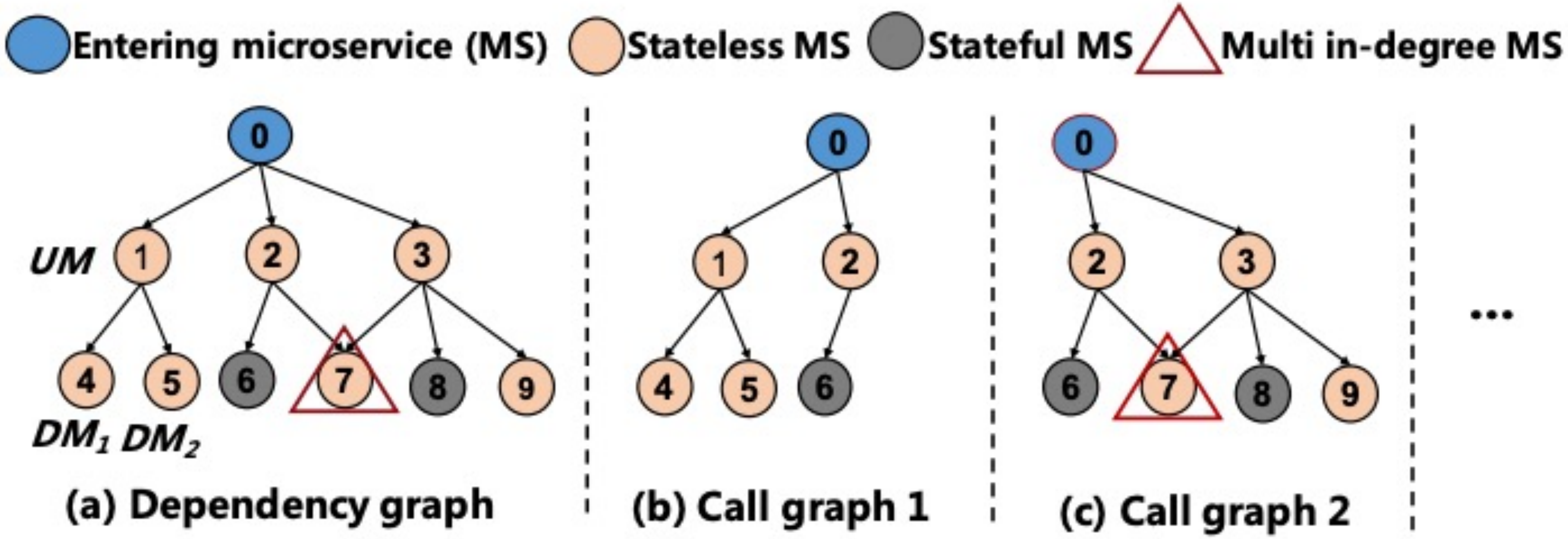


Microservice Architecture

Dependency graph: DAG, tree-like, less multi in-degree; **Entering microservice:** frontend web service

Call graph: Part of microservices, different query patterns; **Example:** user-selected recommendation methods

Dependency graph and call graphs



Complex dependency structure; Various call graphs of user queries

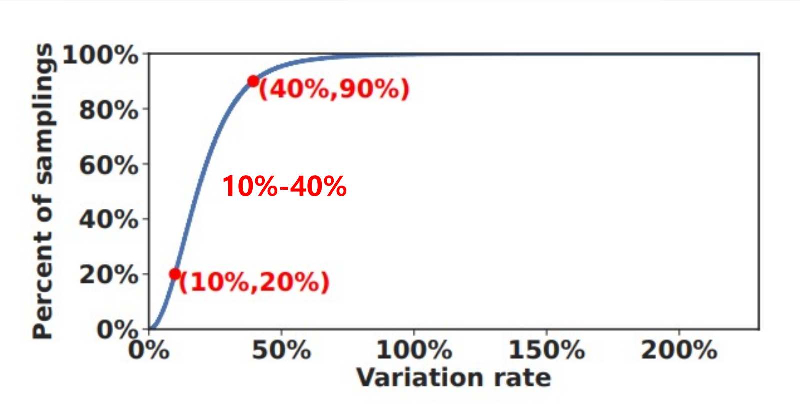
Motivation

Motivation

SJTU

Load and Call Graph Dynamics

Alibaba microservice trace 2021: 3000+ applications in 12 hours

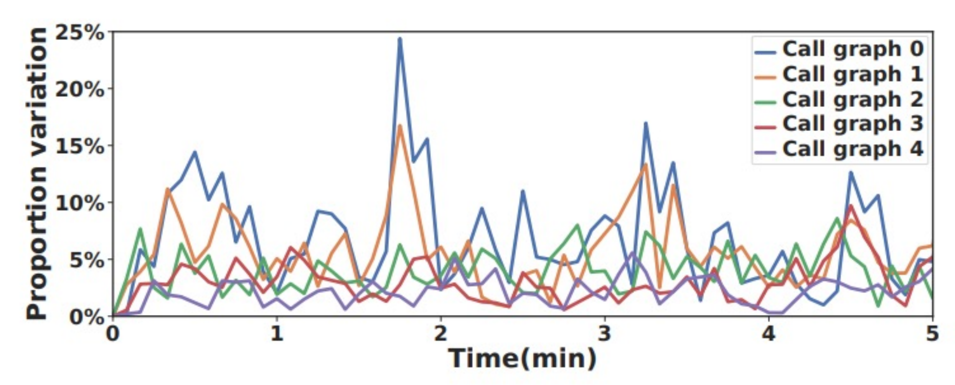


Load dynamics over time

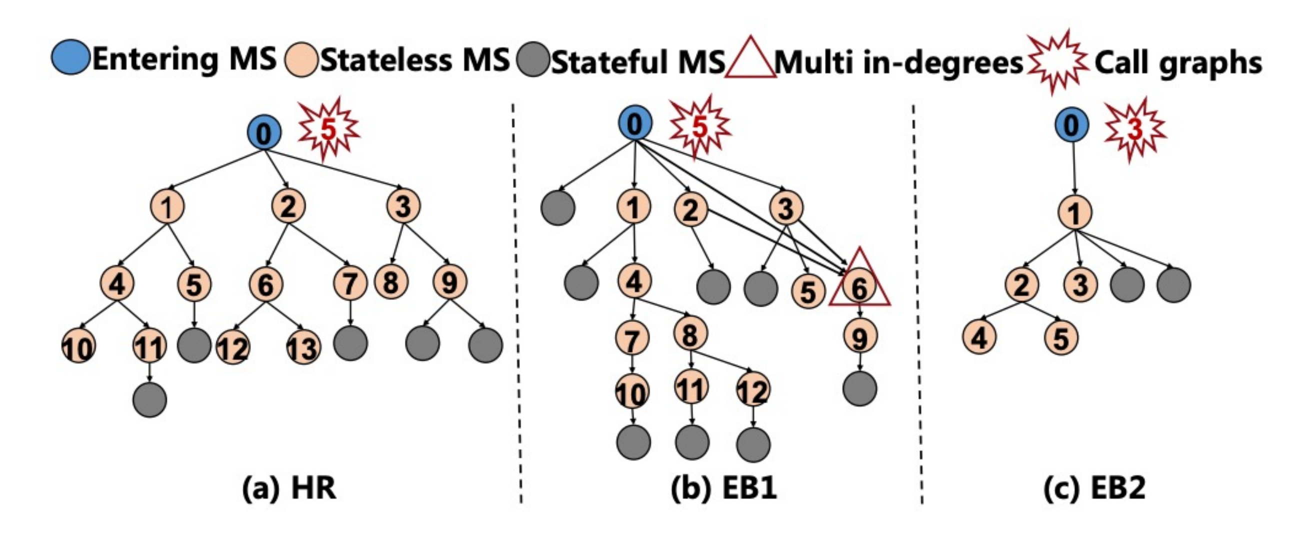
Load dynamics: Application load changes over time.

Call Graph dynamics: Call graph proportion changes over time.

Microservice dynamics: Load dynamics + Call graph dynamics



Call graph dynamics over time



Three investigation benchmarks

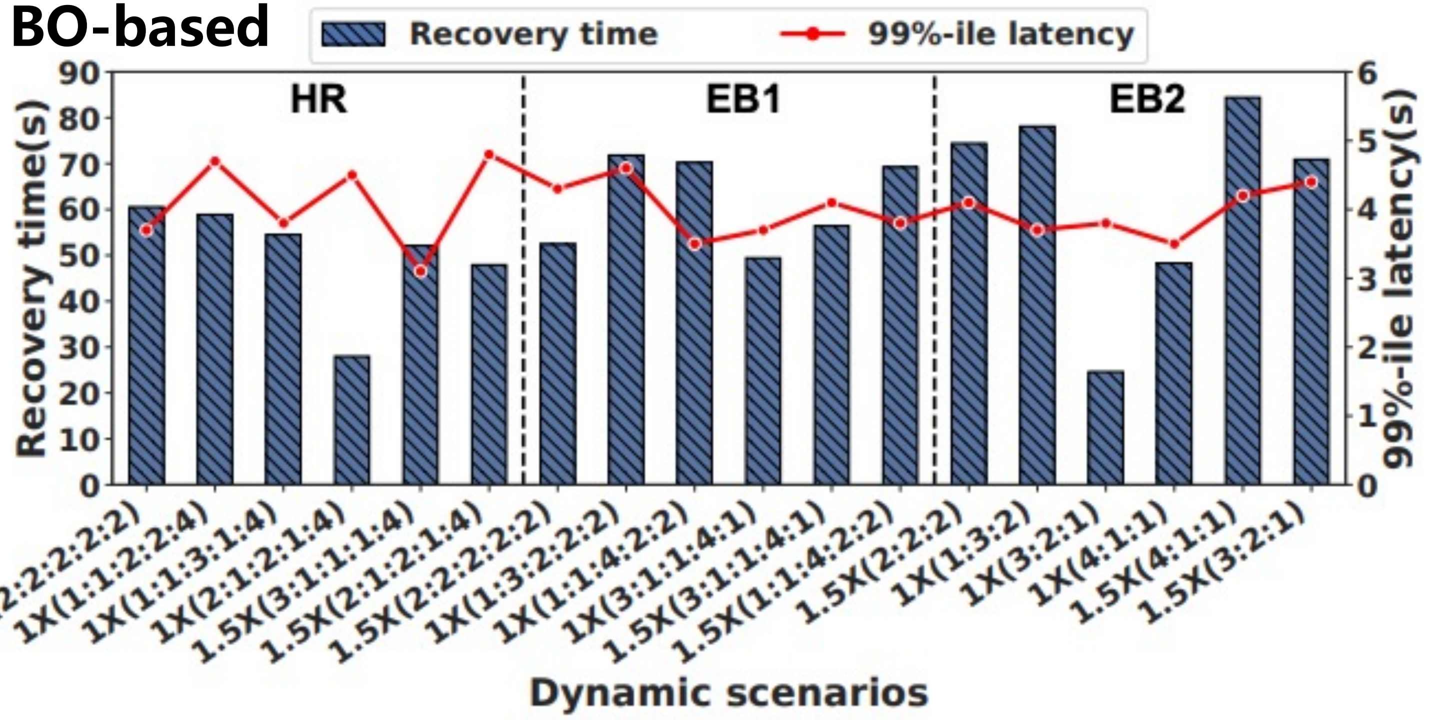
Current Works of Microservice Management

Proactive methods: Predict load/performance, Call graphs, Unpredictable dynamics

Reactive methods: Latency monitor, ML-based, Adjust individually

QoS recovery time: Time needed to reduce the 99%-ile latency below a fixed target

ELIS: BO-based



Evaluation Setup:

3 benchmarks

18 dynamic scenarios

Load+Call Graph dynamics

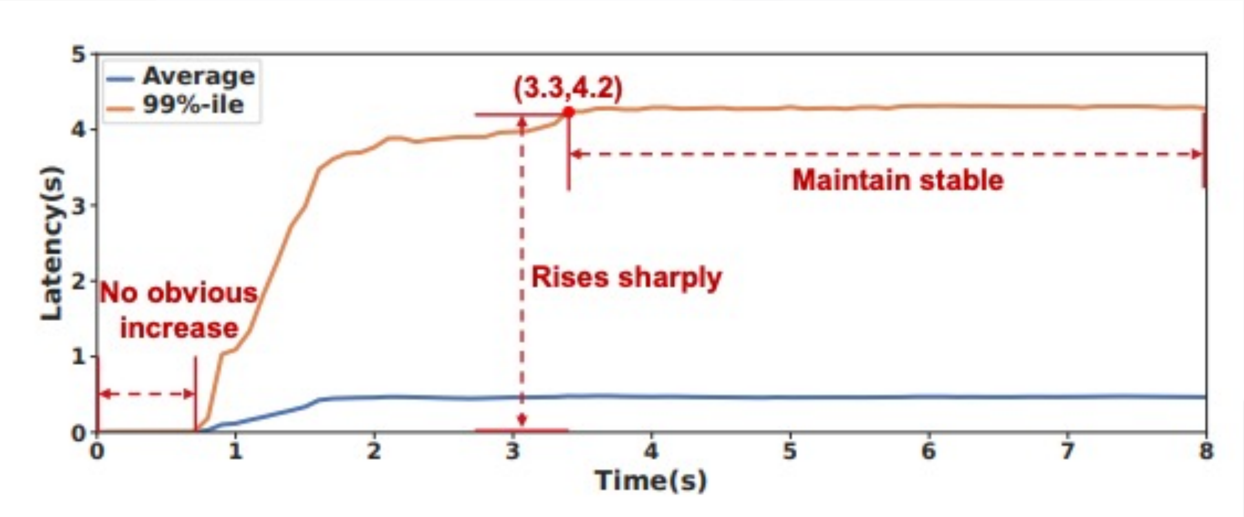
QoS recovery time:
24.6 to 84.4 seconds

Current works have long QoS recovery time.

Causes of the Long Recovery Time

Long Monitoring Interval

- Monitoring real-time latencies of microservices
- Allocate resources based on latencies
- Interval: seconds or tens of seconds



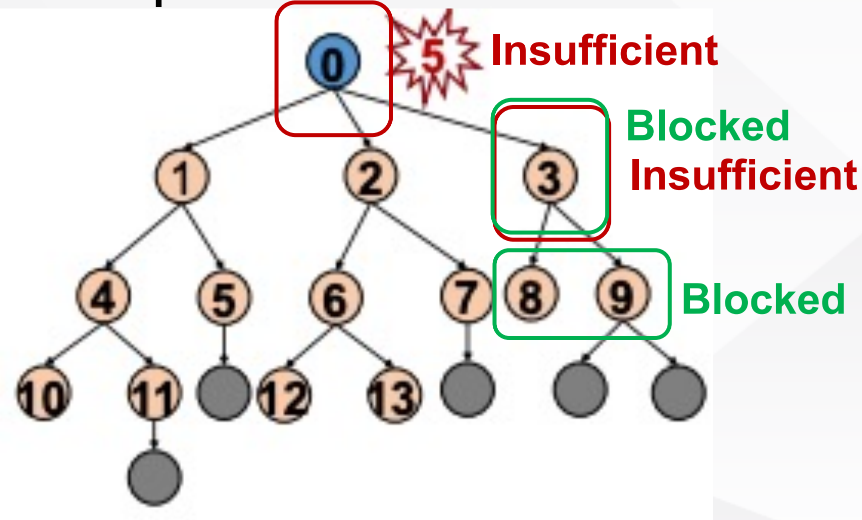
Latency change after dynamics happen

Latency monitor interval needs to be long enough.

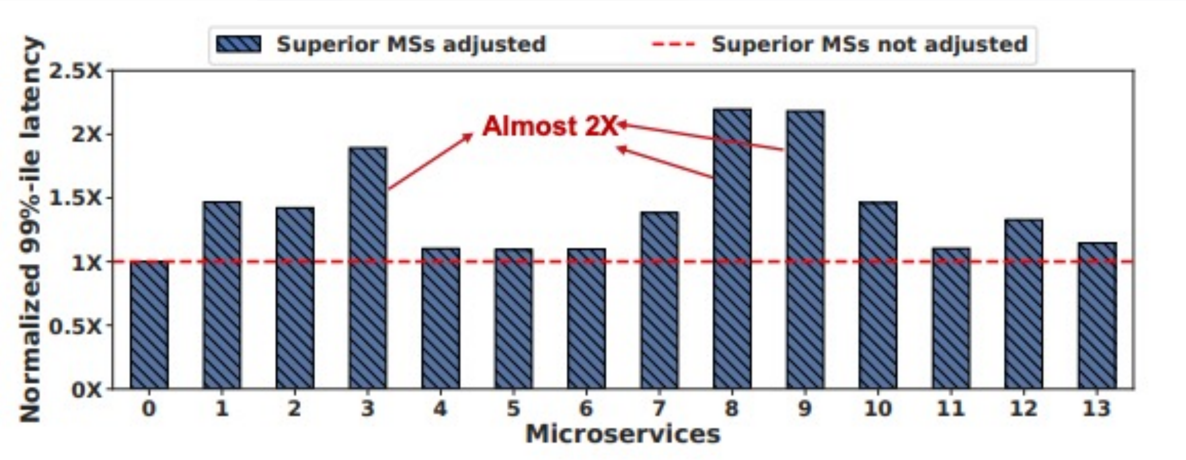
Causes of the Long Recovery Time

Execution Blocking Effect

- Monitored ≠ to-be-processed
- ms-0 blocks ms-3, ms-3 blocks of ms-8+9
- Latencies increase when their superiors get enough resources
- Get to-be-processed loads when blocking is alleviated



Blocking examples



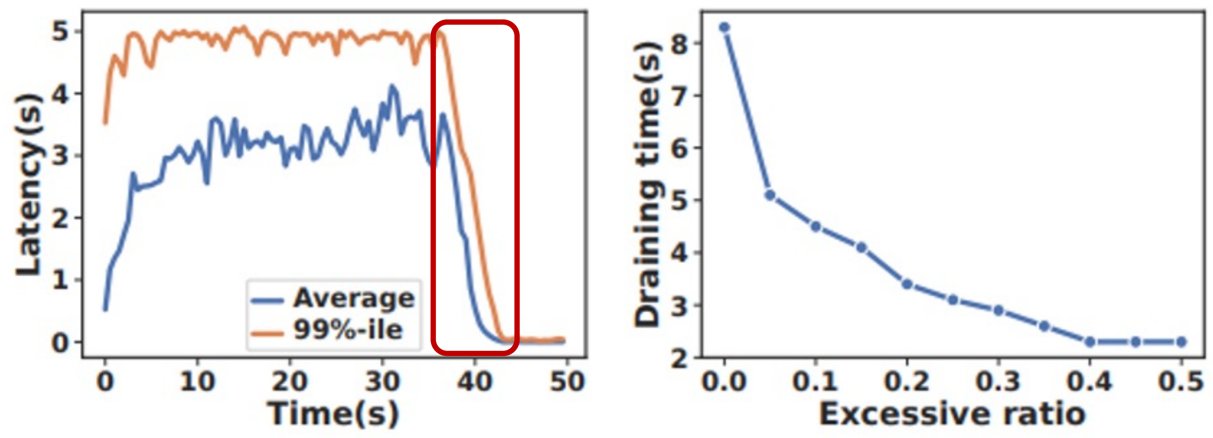
Execution blocking effect among MSs

Need to adjust resources for multiple times => long QoS recovery time

Causes of the Long Recovery Time

Slow Query Draining

- **Queued queries** can lead to long QoS recovery time.
- Queued queries need **extra time** to be drained under just-enough resources.
- More **excessive resource allocation**, shorter queued query draining time.



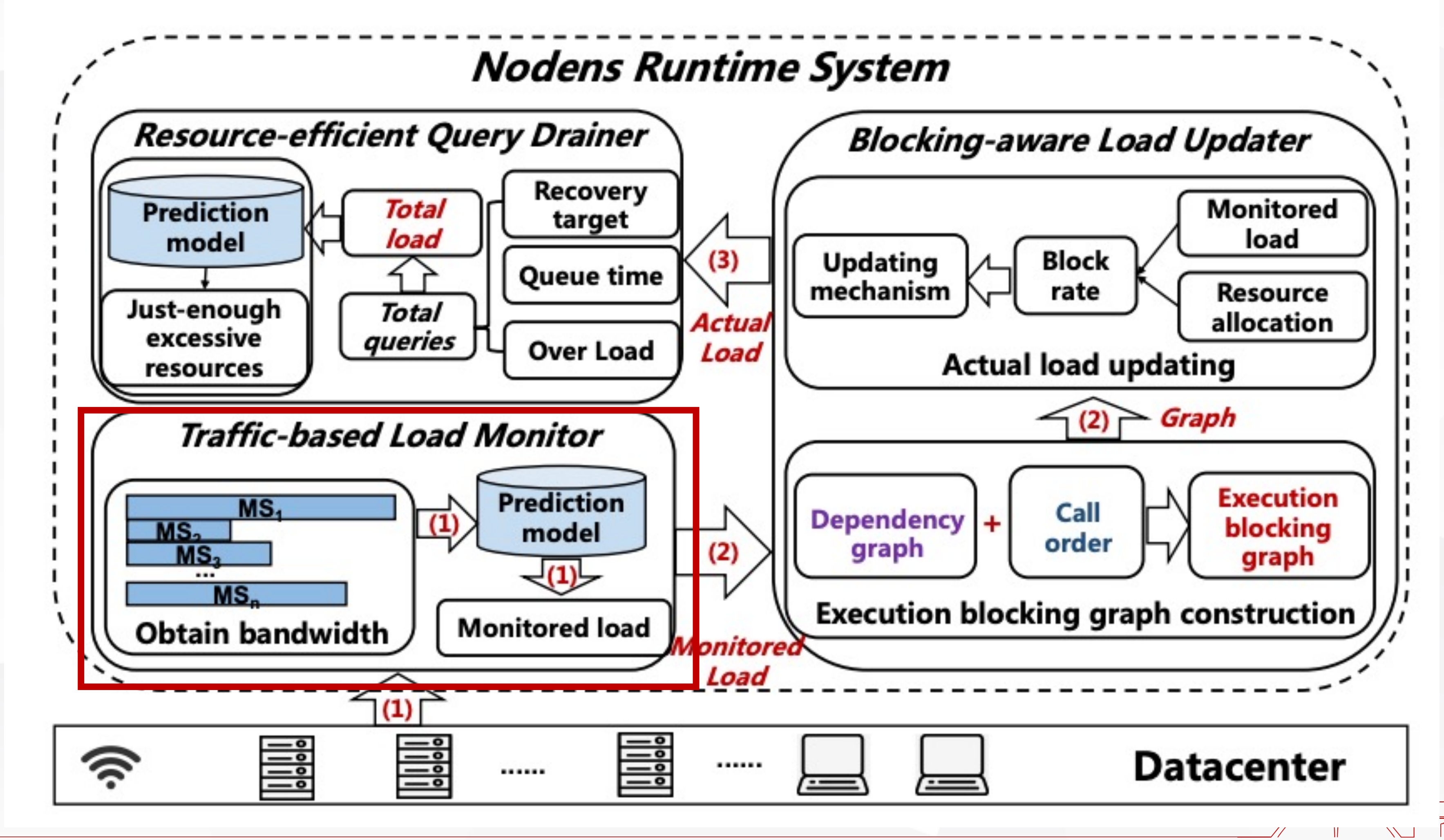
(a) QoS recovery process. (b) Excessive resource ratio.

Excessive resource allocation can reduce overall QoS recovery time.

Nodens Overview and Design

Nodens Overview

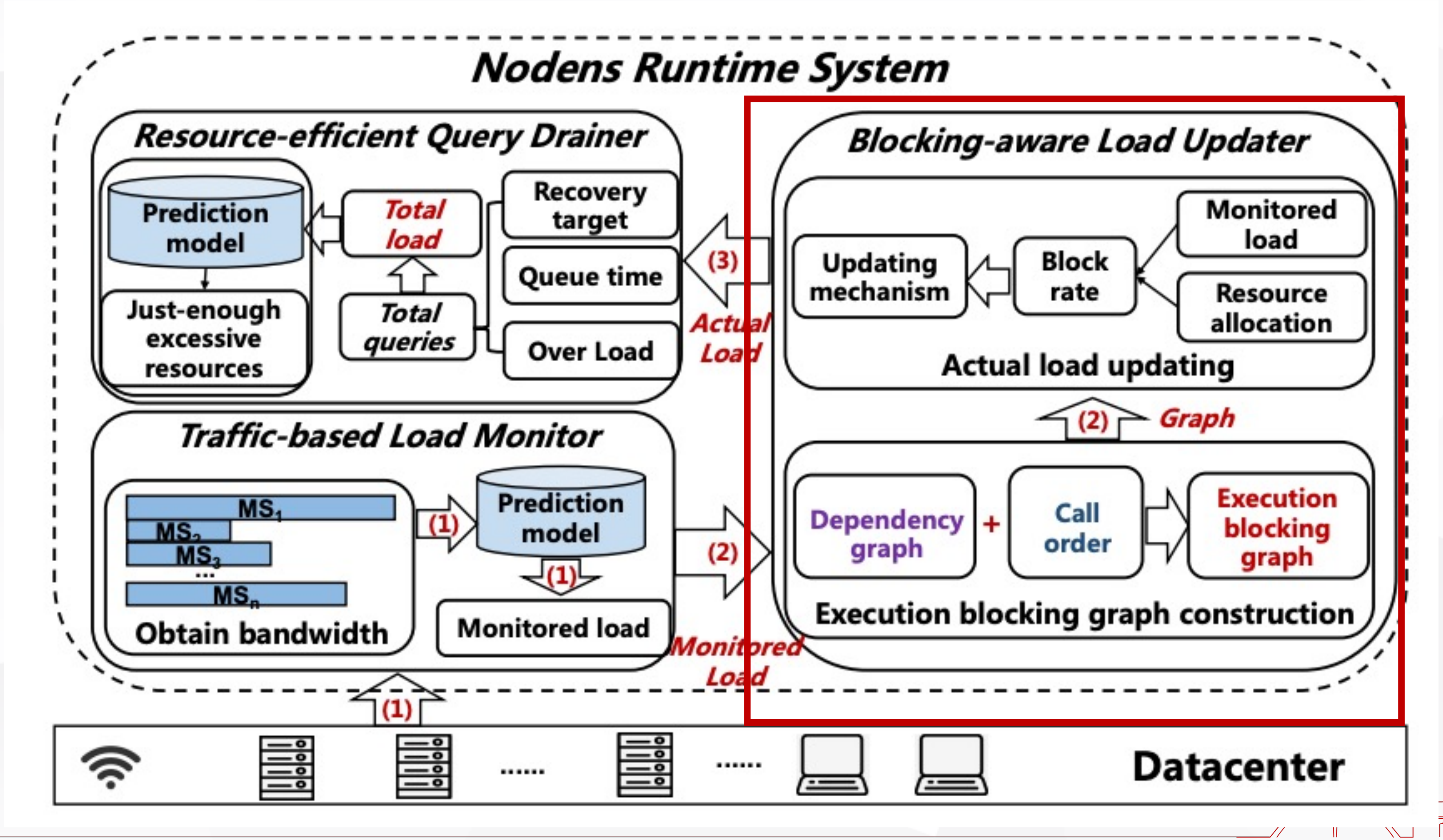
***Traffic-based Load Monitor*:**
Obtain monitored loads of microservices based on monitored network traffic.



Nodens Overview

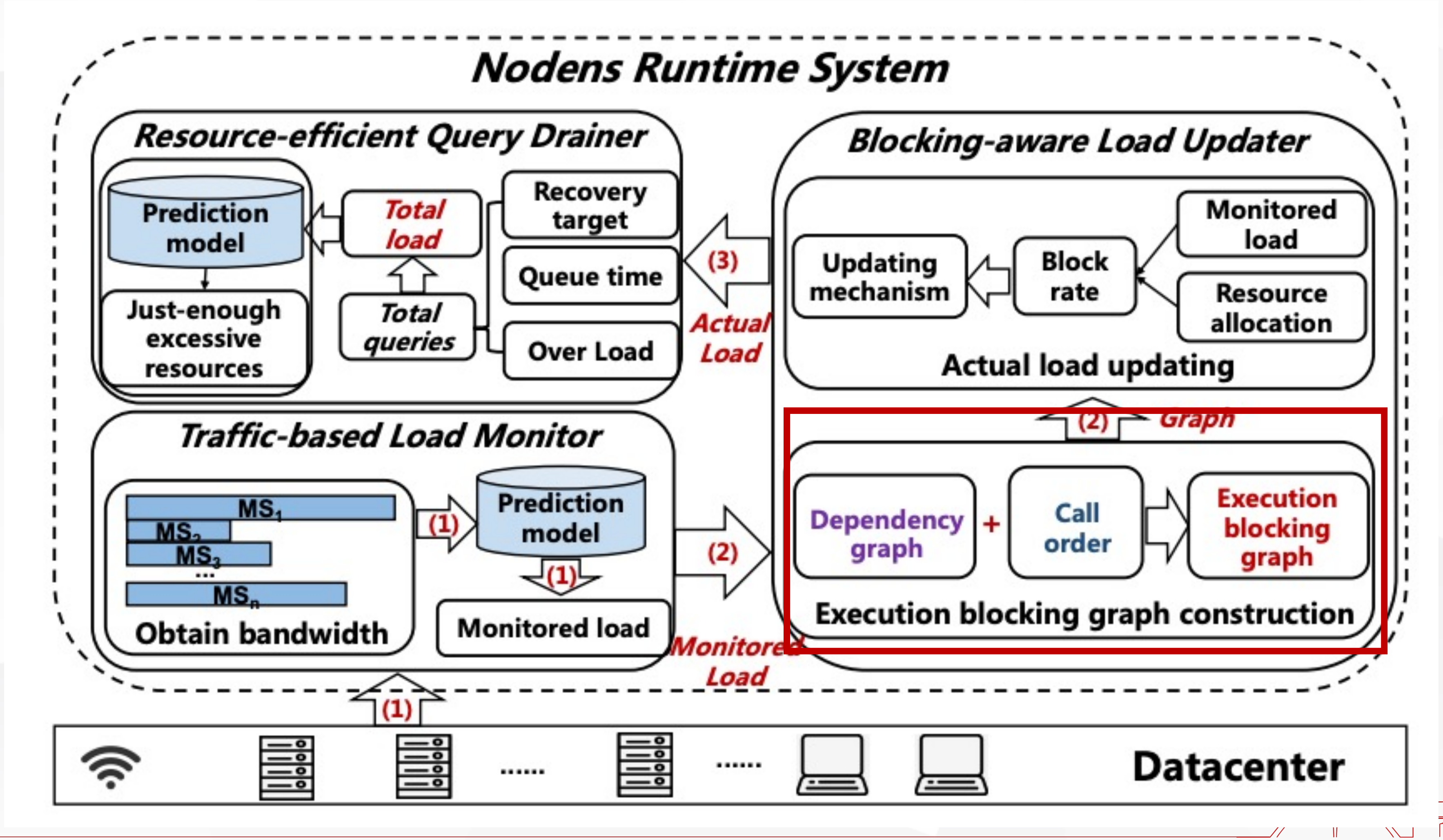
***Blocking-aware Load Updater*:**

(1) Execution Blocking Graph; (2) Actual Load Updating



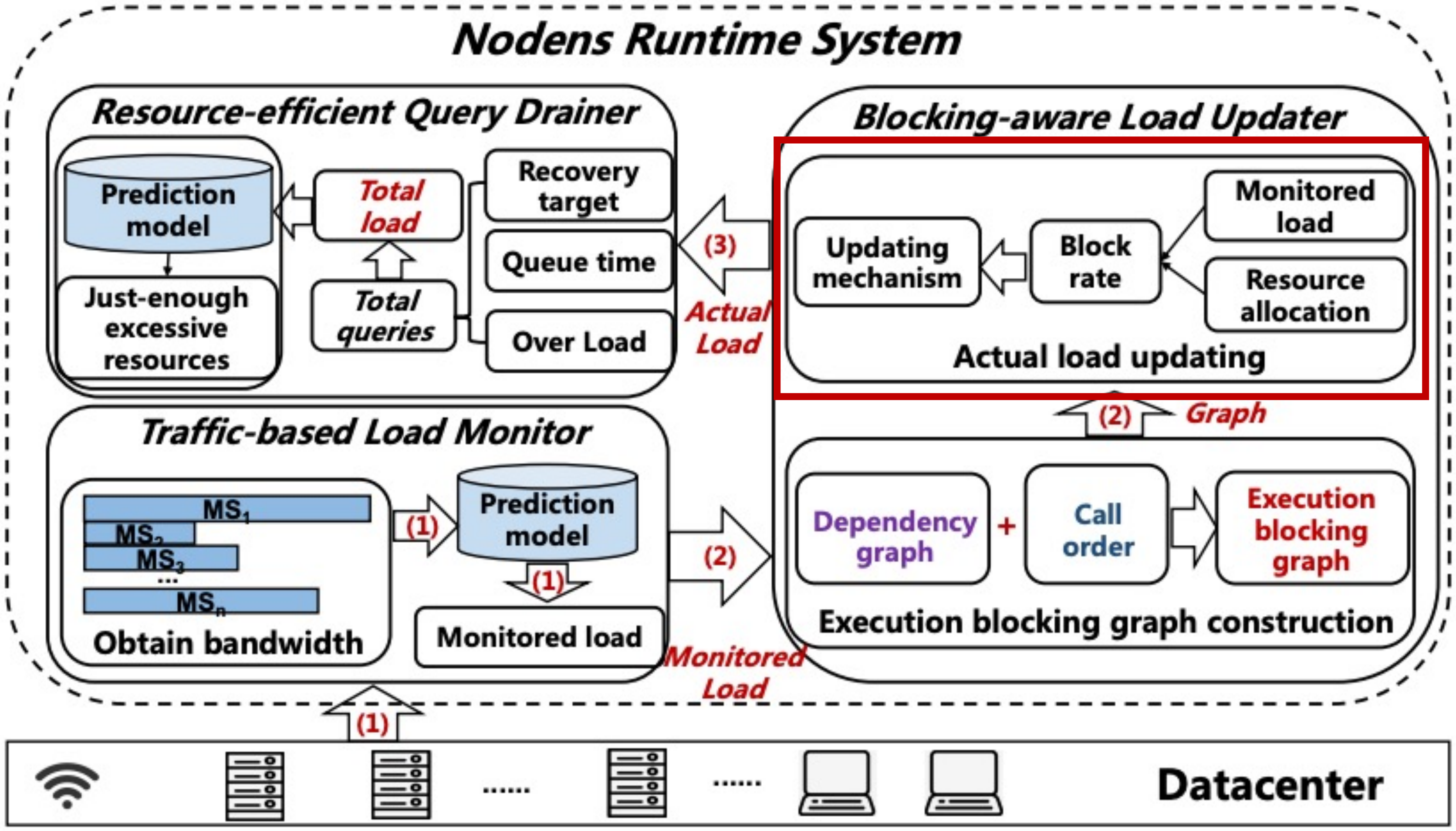
Nodens Overview

Execution Blocking Graph:
Capture all the execution blocking relationships among different microservices.



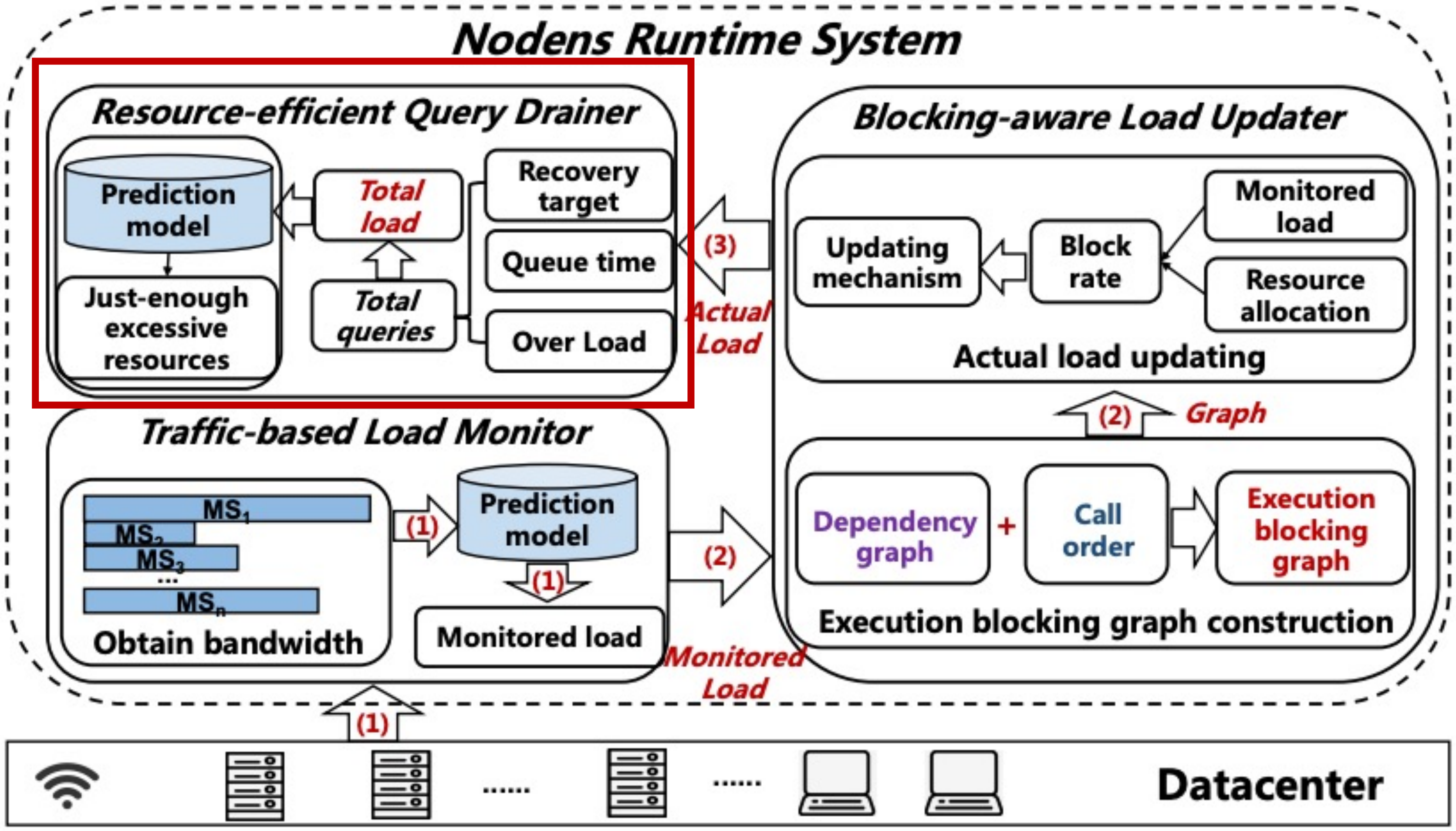
Nodens Overview

***Actual Load Updating*:**
Updating actual to-be-processed loads of microservices based on execution blocking graph.



Nodens Overview

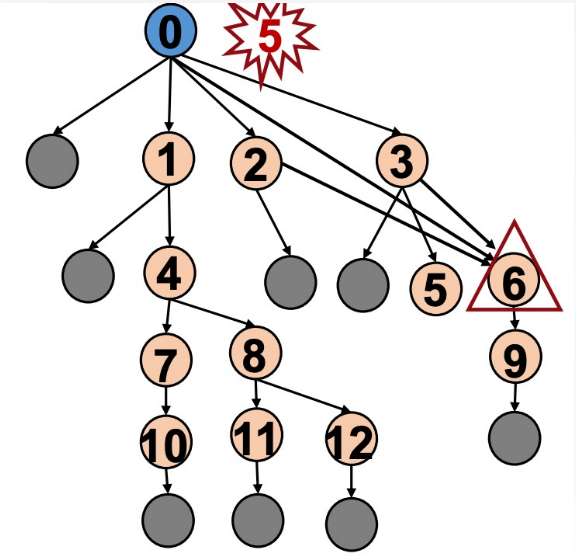
***Resource-efficient Query Drainer*:**
Allocate just-enough excessive resources for microservices to drain queued queries.



Traffic-based Load Monitor

① Obtain upper network bandwidth usage of microservices

Tree-like dependency graph: Linux file /proc/net/dev, calculate hierarchically	**Graph-like dependency graph:** Use Libpcap to capture packets among MSs



② Obtain monitored loads based on network traffic

Upper network traffic → Predict models → Monitored loads of microservices

linear ***linear***

Traffic => *Load* => *CPU demand*

microservice-6: data communication amount per second from microservices 0, 2, and 3.

③ Results input to load updater

Blocking-aware Load Updater

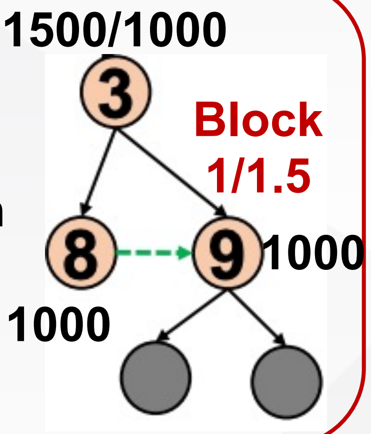
Execution Blocking Effect

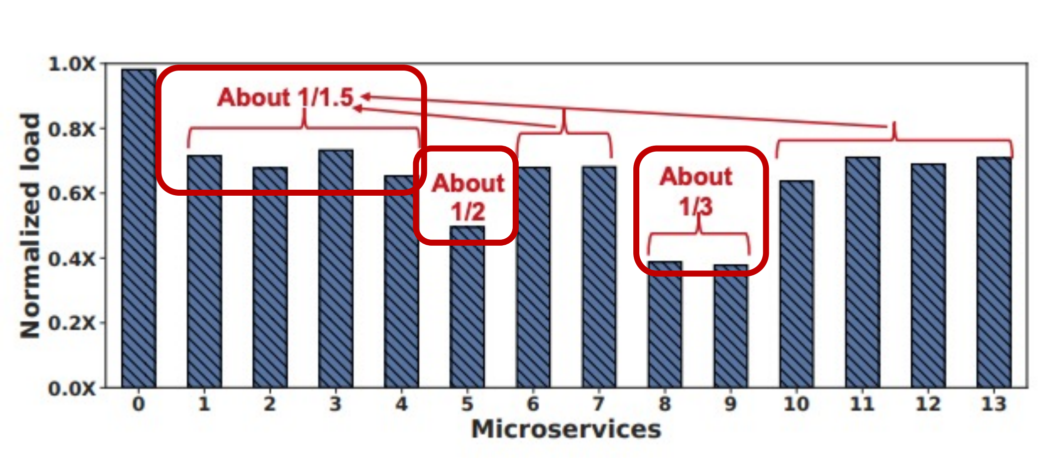
- ❑ 1.5X load + call graph dynamics
- ❑ Call dependencies among microservices
- ❑ Call order among microservices

microservice-3: log-in

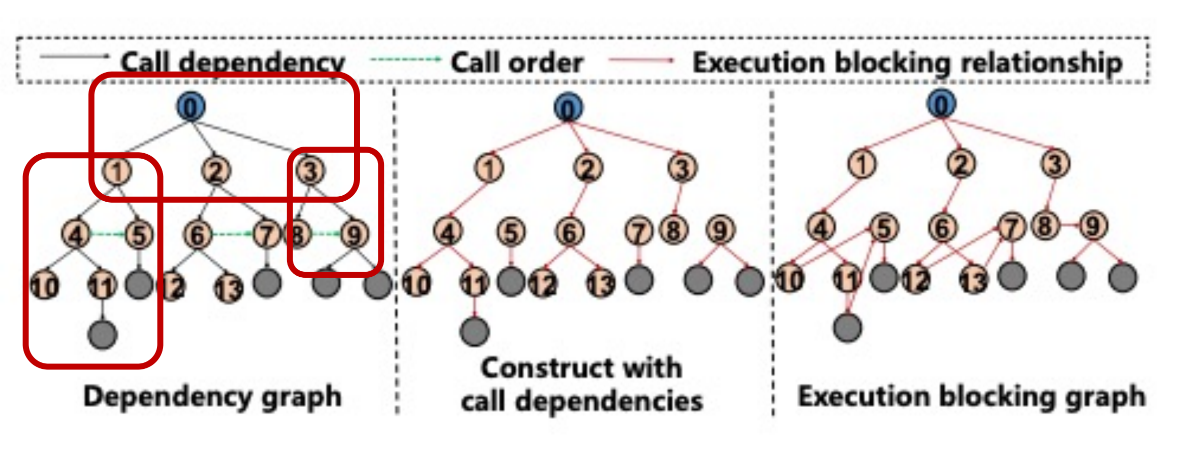
microservice-8: authentication

microservice-9: reservation





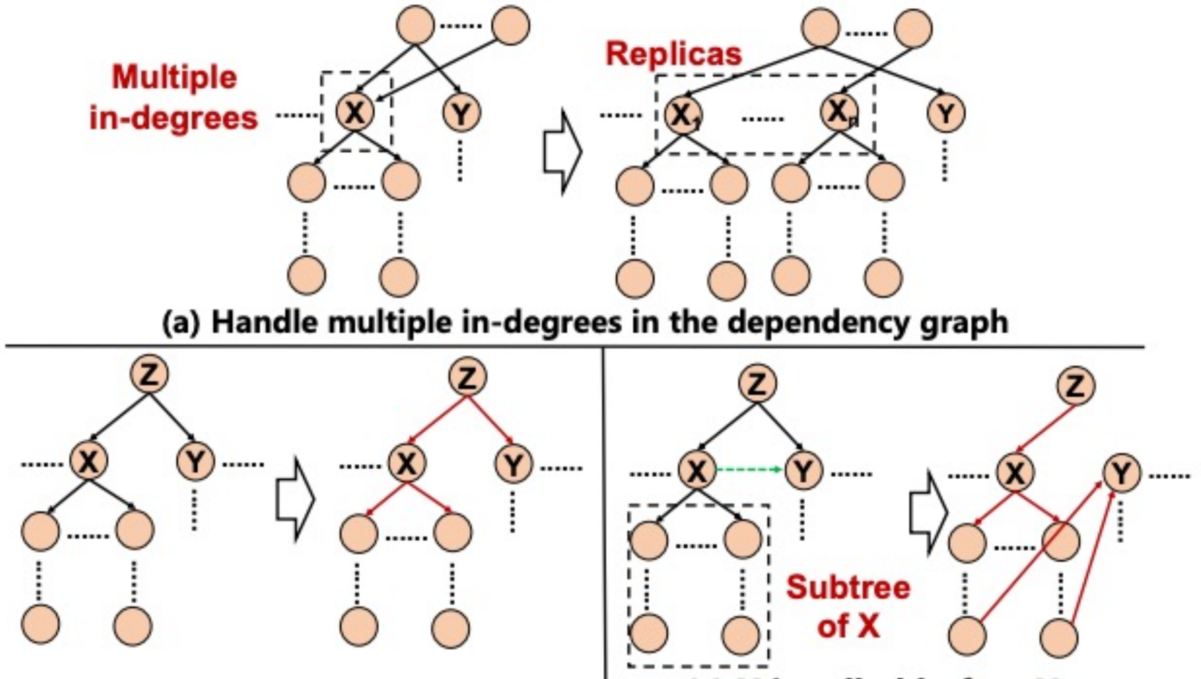
Monitored/Actual loads of an example



Graphs of HR benchmark

Blocking-aware Load Updater

Execution Blocking Graph



(a) Handle multiple in-degrees in the dependency graph

(b) X and Y have no call order

(c) X is called before Y

❑ Construction process

① Multiple in-degrees => multiple replicas

② Construct blocking relationship for sub-structures

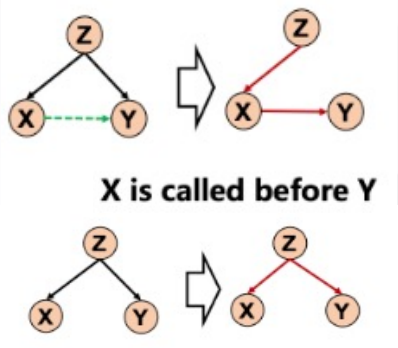
③ No order: blocking=dependency relationship

④ Call order: ends of blocking subtree of X

Z: log-in

X: authentication

Y: reservation



X is called before Y

X and Y has no call order

Blocking-aware Load Updater

Actual Load Updating Mechanism

❑ Blocking rate

$$rate_j = max(\frac{ActualLoad_j}{min(HandleLoad_j, MonitoredLoad_j)}, 1) \quad (1)$$

❑ Load updating mechanism

① Follow the **BFS** process of the blocking graph

② Calculate **Blocking Rate** of the front node in BFS queue

③ Update **Actual Loads** pass to downstream microservices

④ Push the **Updated Microservice** into the BFS queue

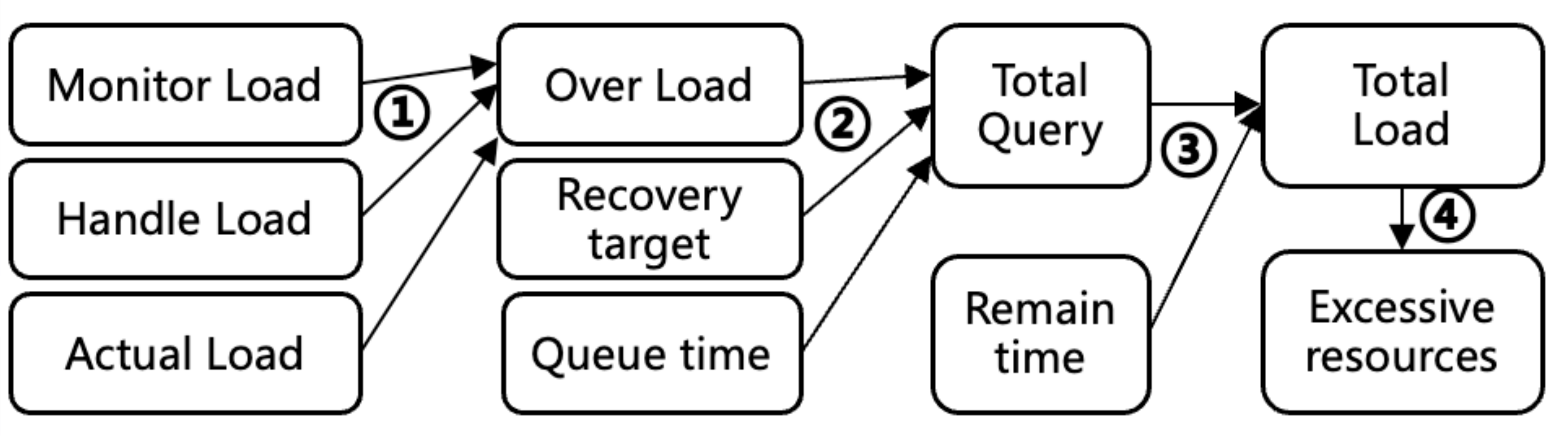
⑤ All **Actual Loads** are updated after the BFS process

Algorithm 1: Actual Load Updating Mechanism

Initialize $(MonitoredLoad_i, ActualLoad_i, HandleLoad_i)$
Initialize execution blocking graph EBG with edge weights EW_{ij}
Initialize a queue q for the BFS process
$q.put(EBG.root)$
while $q \neq \emptyset$ **do**
  $j = q.get()$
  $ActualLoad_j = \sum_{i \to j} EW_{ij}$
  $rate_j = max(\frac{ActualLoad_j}{min(HandleLoad_j, MonitoredLoad_j)}, 1)$
  for each downstream microservice k of j **do**
    $EW_{jk} = EW_{jk} \times rate_j$
    if all entry edges of k are updated **then**
      $q.put(k)$
    end if
  end for
end while
return $ActualLoads$ for all microserivces

Resource-efficient Query Drainer

- **QoS recovery time target:** the recovery time is within 3 seconds after dynamics happen
- **Minimize:** excessive resource allocation, **s.t.** Recovery time target is ensured
- Input: MonitoredLoad, HandleLoad, ActualLoad
- Output: **Just-enough excessive resources** for microservices



Evaluation

Evaluation Setup

- Hardware: Eight-node server
- Software: Docker, Kubernetes; three benchmarks
- Testing cases: 18 dynamic scenarios with load+call graph dynamics
- Baselines: ELIS and FIRM, directly allocate optimal resources, Nodens's query drainer

Table 1: Experiment specifications

	Specifications
Hardware	Eight-node cluster, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 256GB Memory Capacity, 25 MiB L3 Cache Size (20-way set associative)
Software	Ubuntu 20.04.2 LTS with kernel 5.11.0-34-generic Docker version 20.10.18, Kubernetes version v1.20.4

Reducing QoS Recovery Time

- Just-enough resources with 1X initially, then change load and call graph proportion
- Nodens eliminates QoS violation in given recovery targets in all dynamic scenarios
- Reduce the QoS recovery time by 12.1X and 10.2X than ELIS and FIRM
- FIRM < ELIS: Directly adjust critical microservices

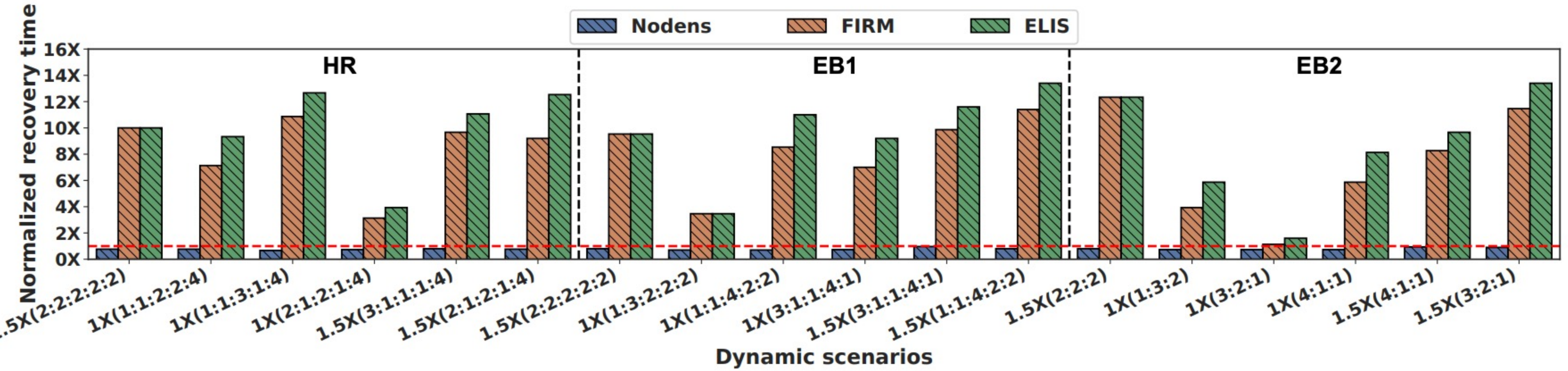


Figure 14: The normalized QoS recovery time relative to the recovery time target of benchmarks with Nodens, FIRM, and ELIS.

Nodens has shorter QoS recovery time under microservice dynamics.

Resource Usage during Recovery Period

- Use the longest QoS recovery time (ELIS's) to calculate cores×hours.
- Nodens uses 1.5% and 6.1% more resources on average than FIRM and ELIS
- FIRM ＞ ELIS: ELIS spends extra time to actively recycle over-provisioned resources.

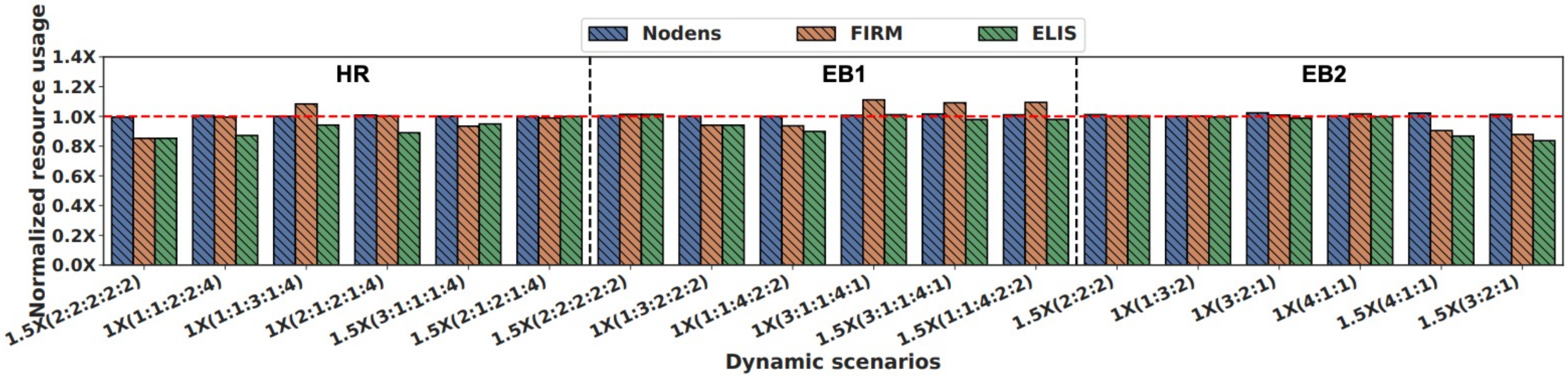
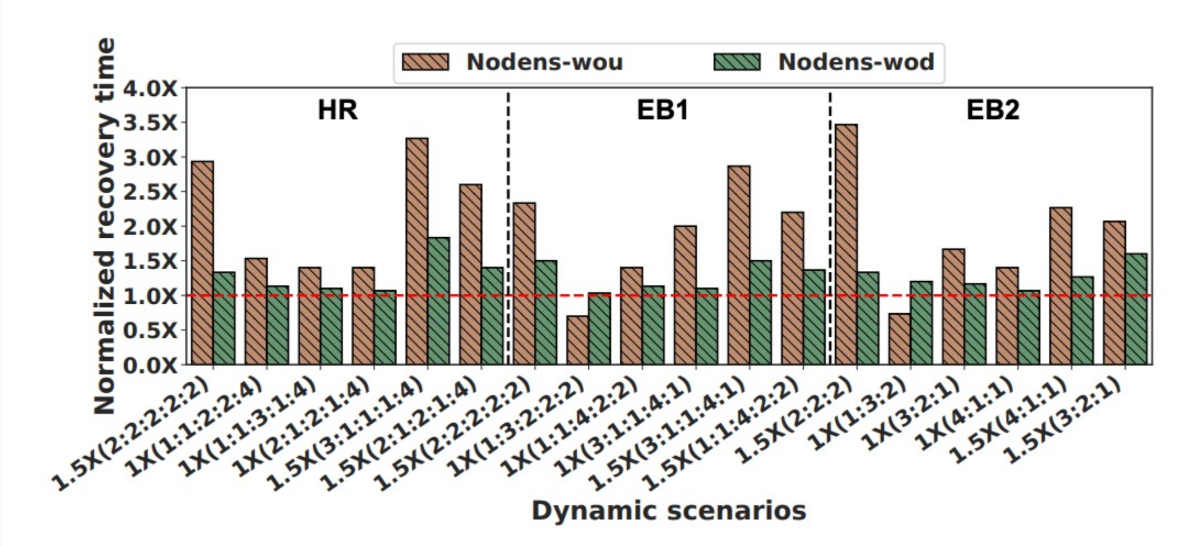


Figure 15: The normalized resource usage relative to the just-enough resources of benchmarks with Nodens, FIRM, and ELIS.

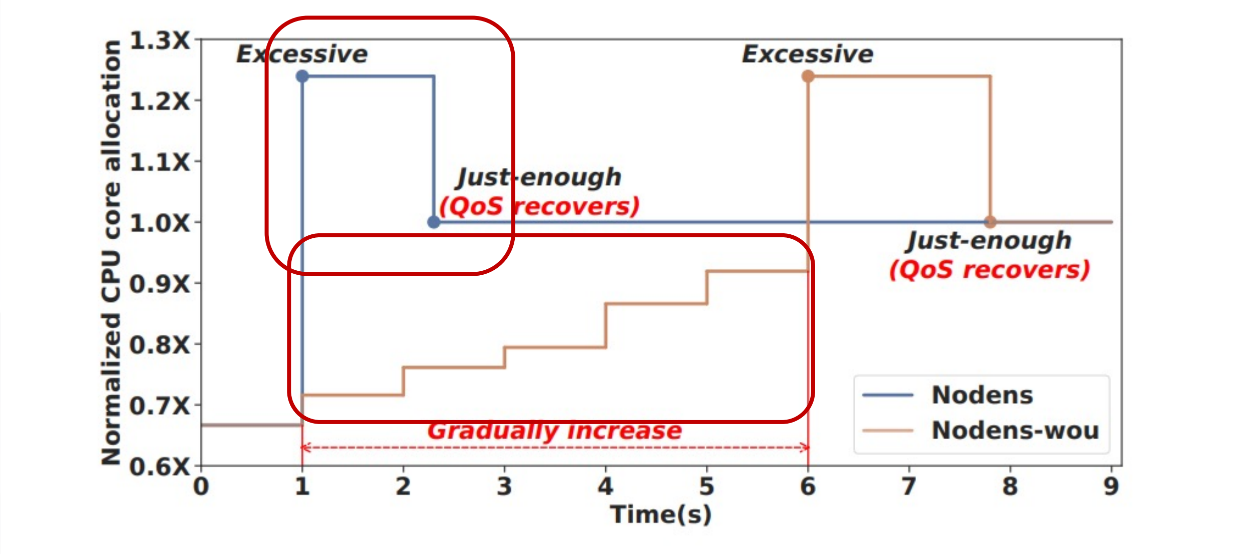
Nodens uses small amount more resources, maintains resource efficiency.

Effectiveness of the Load Updater

- Nodens-wou: disables the blocking-aware load updater
- Nodens-wou recovers in two cases, requires 2.6X time than Nodens
- Execution blocking makes Nodens-wou only obtain actual loads layer by layer.



QoS recovery time

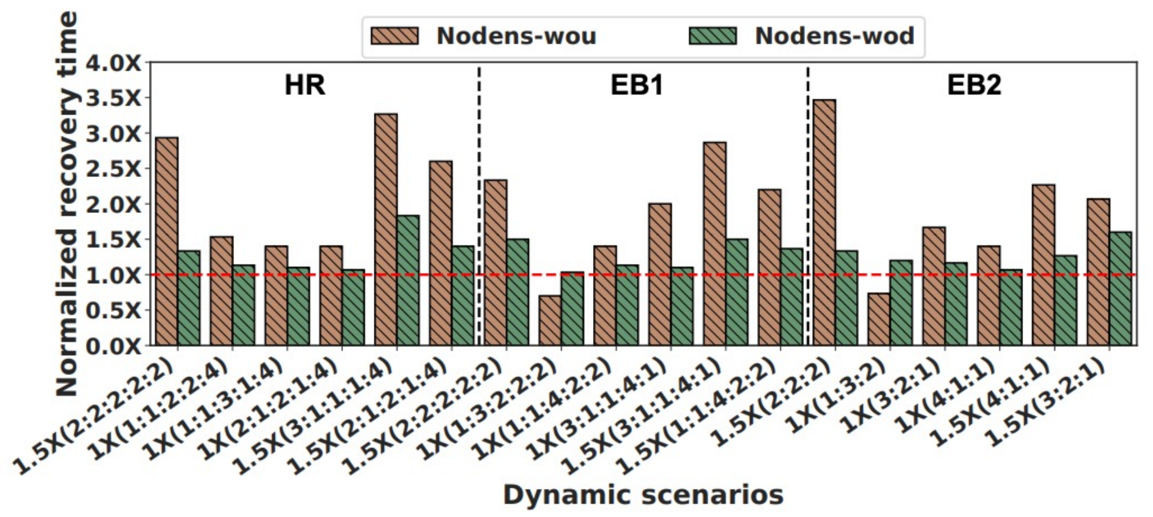


Example of allocation timeline

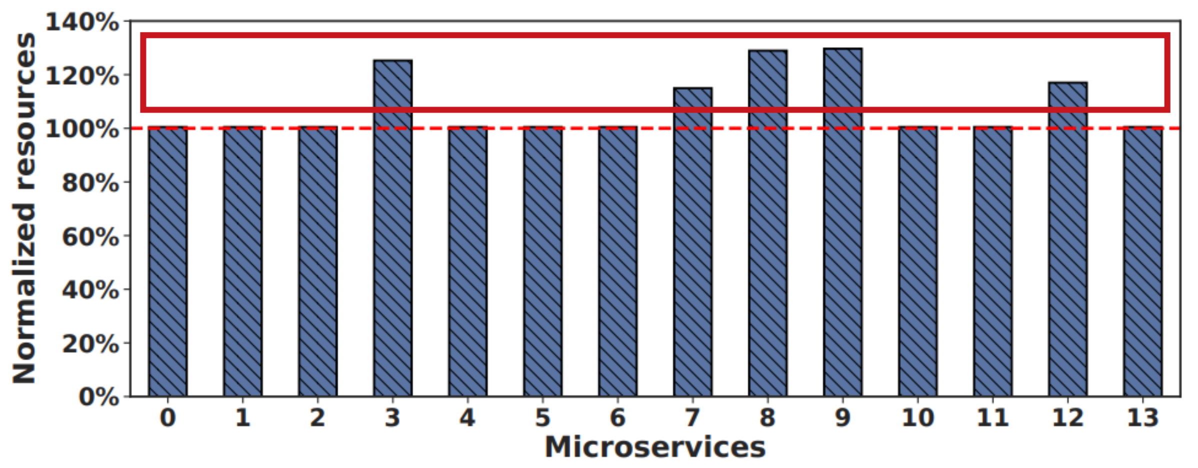
Load updater avoids execution blocking effect by updating actual loads in advance.

Effectiveness of the Query Drainer

- Nodens-wod: disables the resource-efficient query drainer
- Nodens-wod fails in all cases, requires 1.6X time than Nodens
- Query drainer allocates "just-enough" excessive resources to microservices



QoS recovery time



Excessive resource allocation example

Query drainer drains queued queries quickly with high resource efficiency.

Conclusion

Conclusion

Monitor load change quickly: Network traffic based load monitor

Capture blocking relationships: Execution blocking graph construction

Update actual loads in advance: Blocking rate based actual load updating

Drain queued queries quickly: Resource-efficient query draining

Under microservice dynamics, above techniques enable
Fast QoS recovery and high resource efficiency.

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

The end

饮 水 思 源　爱 国 荣 校