QoS-Aware Co-Scheduling for Distributed Long-Running Applications on Shared Clusters

Jianyong Zhu, Renyu Yang, *Member, IEEE*, Xiaoyang Sun, Tianyu Wo, *Member, IEEE*, Chunming Hu, Hao Peng, Junqing Xiao, Albert Y. Zomaya, *Fellow, IEEE*, Jie Xu, *Member, IEEE*

Abstract—To achieve a high degree of resource utilization, production clusters need to co-schedule diverse workloads – including both batch analytic jobs with short-lived tasks and long-running applications (LRAs) that execute for a long time frame from hours to months – onto the shared resources. Microservice architecture advances the manifestation of distributed LRAs (DLRAs), comprising multiple interconnected microservices that are executed in long-lived distributed containers and serve massive user requests. Detecting and mitigating QoS violation become even more intractable due to the network uncertainties and latency propagation across dependent microservices. However, current resource managers are only responsible for resource allocation among applications/jobs but agnostic to runtime QoS such as latency at application level. The state-of-the-art QoS-aware scheduling approaches are dedicated for monolithic applications, without considering the temporal-spatio performance variability across distributed microservices. In this paper, we present TOPOSCH, a new scheduling and execution framework to prioritize the QoS of DLRAs whilst balancing the performance of batch jobs and maintaining high cluster utilization through harvesting idle resources. TOPOSCH tracks footprints of every single request across microservices and uses critical path analysis, based on the end-to-end latency graph, to identify microservices that have high risk of QoS violation. Based on microservice and node level risk assessment, we intervene the batch scheduling by adaptively reducing the visible resources to batch tasks and thus delaying their execution to give way to DLRAs. We propose a prediction-based vertical resource auto-scaling mechanism, with the aid of resource-performance modeling and fine-grained resource inference and access control, for prompt recovery of QoS violation. A cost-effective task preemption is leveraged to ensure a low-cost task preemption and resource reclamation during the auto-scaling. TOPOSCH is integrated with Apache YARN and experiments show that TOPOSCH outperforms other baselines in terms of performance guarantee of DLRAs, at an acceptable cost of batch job slowdown. The tail latency of DLRAs is merely 1.12x of the case of executing alone on average in TOPOSCH with a 26% JCT increase of Spark analytic jobs.

Index Terms—resource scheduling, cluster management, QoS, tail latency, datacenters

1 INTRODUCTION

Production clusters are increasingly consumed by various workloads mainly including batch jobs for data analytics [1], [2], [3], [4] and long-running applications (LRAs) for online cloud services (e.g., Storm [5], Flink [6], HBase [7], MongoDB [8], Tensorflow [9], etc.) for transaction analytics, streaming process, and data store and query. By co-managing diverse workloads onto the same host server, workload co-location has become a common practice in improving resource utilization and cost efficiency. As opposed to batch analytic jobs that usually consist of a large number of short-lived tasks and are measured by the end-to-end job completion time, LRAs have now become another mainstream workloads in production clusters (Google [10], Microsoft [11], Alibaba [12]). LRAs are latency-critical – the stringent quality-of-service (QoS) such as response latency and throughput is of the upmost criticality and must be met to deliver the business promise in the face of network jitters or load spikes. For example, the 95th percentile of requests need to complete within a latency threshold.

Microservice architecture is an approach to constructing a single application as a set of small interconnected services. Each microservice runs individually and communicates with each other mostly using remote procedure calls (RPC) [13]. In this context, a *Distributed Long Running Application* (DLRA) is referred to as the microservice-based application with microservices executed in the long-lived distributed containers. Compared to monolithic applications, request latency is prone to any network turbulence that will coherently affect massive communications in the DLRA. Pinpointing the QoS violation (e.g., mean or tail latency over a threshold) is ever-increasingly intricate because the latency in a single microservice can promptly propagate across all dependent microservices and ultimately result in the entire performance slowdown [13].

However, traditional cluster managers [14][15][3][16] are originally designated for short-running batch jobs. The central resource manager (RM) is only responsible for resource allocation among applications/jobs, yet leave all application-specific logic to application managers (AMs). This means that RM is completely unaware of the runtime QoS requirements of the interactive and latency-sensitive applications. Other workload co-location solutions either diminish the performance interference through resource

- *: J.Zhu and R.Yang are co-first authors with equal contribution.
- J.Zhu are with Department of Computing and with Engineering Research Center of Intelligent Computing for Complex Energy Systems, Ministry of Education, North China Electric Power University, Baoding, 071003, China. Email:zhujy@ncepu.edu.cn.
- R.Yang, X.Sun and J.Xu are with the School of Computing, University of Leeds, Leeds LS2 9JT, UK. Email: {r.yang1, scxs, j.xu}@leeds.ac.uk.
- T.Wo, C.Hu and H.Peng are with Beihang University, Beijing 100083, China. Email:{woty, hucm, penghao}@act.buaa.edu.cn.
- J.Xiao is with Alibaba Group, China. Email: junqing.xjq@alibaba-inc.com
- A.Y.Zomaya is with the University of Sydney, Australia. E-mail: albert.zomaya@sydney.edu.au.

Manuscript received Nov 2021; revised July 2022 (corresponding author: Tianyu Wo)

partition and isolation [17][18][19] or minimize the performance interference when co-locating different workloads [20][21][22]. Nevertheless, they are exclusively devised for monolithic applications and cannot be directly applied to tackle the sophisticated component dependencies and latency variations when substantial and dynamic requests manifest in the constituent microservices of the DLRA.

In this paper we present TOPOSCH, a QoS-centric resource management and runtime execution framework that can prioritize the QoS of DLRAs whilst balancing the performance of batch jobs and maintaining high cluster utilization. TOPOSCH encompasses two coherent stages to tackle the QoS violation: (i) In *QoS violation containment phase*, we first exploit the instrumentation to trace footprints of each request across different microservices to localize the QoS violation. We take into account timing information – including sojourn time on individual microservice and transmission time between microservices – to establish a latency graph, and periodically perform the critical path analysis to ascertain the chain of invocations with the longest end-to-end latency. The microservices on the critical path are recognized as the victim microservices with higher risks of QoS violation. Based on microservice-level and node-level risk assessment, a risk-aware mechanism is proposed for adjusting the resource reservation for DLRAs and the visibility to batch tasks. We can therefore intervene the scheduling of batch tasks by preventing packing excessive batch tasks onto saturated nodes without exacerbating the QoS violation of DLRAs. (ii) In *QoS violation mitigation phase*, we perform prediction-based *vertical auto-scaling* by learning the QoS sensitivity of long-running containers – particularly those risky microservices such as core databases or data streaming components in the DLRAs – to multi-resources and devising low-cost task preemption and resource reclamation. We infer the proper resource to be vertically scaled based on the QoS-resource model to reach the targeted QoS of the victim microservices. Multi-dimensional resource isolation (CPU cores, caches, main memory, memory bandwidth, etc.) is enforced to precisely control the resource binding and runtime usage. As opposed to the mandatory kill-based preemption that lead to substantial termination of running tasks, we propose a new task preemption mechanism for gradual resource reclamation from low-priority opportunistic batch tasks and leverage multiple pluggable preemption strategies to determine the tasks to be preempted.

TOPOSCH is integrated with the Resource Manager and Node Manager of Hadoop YARN. Experiments show TOPOSCH outperforms other baselines in QoS assurance. The tail latency of DLRAs when co-locating with Spark-based batch jobs is merely 1.12x of the case of executing alone on average and batch jobs experience 26% JCT increase on average when compared with the case of running in native YARN. If the QoS-driven auto-scaling mechanism is disabled, the tail latency of the variant TOPOSCH-n is 1.27x – with less QoS assurance – but the JCT is only increased by 17%. This indicates a performance balance when the proposed auto-scaling design comes into effect. Additionally, the proposed gradual preemption schemes can reduce the JCT by 26.3% and 15.1% as opposed to the kill-based scheme and the least-preempted scheme.

This paper makes the following contributions:

- proposing a mechanism for QoS violation assessment based on critical path analysis which is conducted upon the breakdown of end-to-end request latency among constituent microservices of DLRAs.
- devising an adaptive co-scheduling approach that delays the scheduling of batch tasks according to the runtime risk of QoS violation.
- developing a new mechanism for mitigating QoS violation through prediction-based resource inference and cost-effective auto-scaling of key microservices.

We expand upon our previous work [23] that only focused on the basic scheduling and QoS protection strategy in the containment phase, by (1) scheduling framework redesign to underpin co-scheduling (centralized and decentralized) of DLRAs, batch tasks and opportunistic tasks; (2) significantly augmented scheduling framework to support prediction-based and on-demand mitigation phase, with the particular aid of an enhanced node agent for precise QoS prediction and runtime multi-resource inference and management, and a new resource autoscaler in the DLRA Master for cost-effective QoS recovery and resource reclamation; (3) more comprehensive experimental study with an additional set of workload co-location compositions and with different preemption strategies and the state-of-the-art approaches for comparison.

Organization. Background and solution overview are presented in §2 and §3. We show the technical details in §4 and §5 before the evaluation in §6. We discuss related work in §7 and conclude the paper in §8.

2 BACKGROUND

2.1 Resource Management for Shared Clusters

Cluster scheduling systems typically separate the resource management layer from the job-level logical execution plans. YARN[24] and Fuxi[3] share the following components: *Resource Manager (RM)* is the centralized resource manager, tracking resource usage, node aliveness, enforcing resource quotas among tenants through either capacity or fairness control. *Application Master (AM)* is an application-level scheduler which coordinates the logical plan of a single job by requesting resources from the RM, generating a plan from received resources, and coordinating task execution. *Node Manager (NM)* is a daemon process within each cluster node and responsible for managing task life-cycle and monitoring node information. Traditional workloads in clusters include the data *batch analytic jobs* (abbr. batch jobs) [1], [2], [3], [4] – with short-lived tasks typically in the order of seconds – and the *long-running applications* (LRAs):

LRAs are instantiated by long-standing containers or executors to enable iterative computations in memory or unceasing request-response. Examples of LRAs include applications using streaming processing frameworks (Storm [5], Flink [6], Kafka streams [25]), latency-sensitive database applications (HBase [7] and MongoDB [8]), and data-intensive in-memory computing framework (Tensorflow [9]). Response latency and throughput are the key performance indicators and applications must meet strict QoS.

For the batch workloads, there are typically two classes: regular jobs/tasks and opportunistic jobs/tasks (aka. best-effort or speculative in other systems [26], [10], [16], [27]).

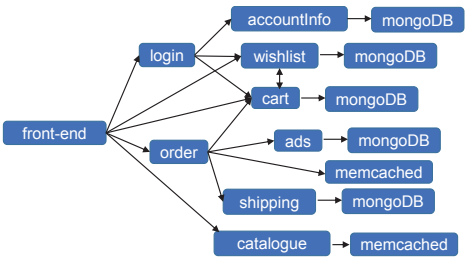


Fig. 1. An e-commerce DLRA for online clothing store [13]

Regular tasks are submitted and managed by the centralized resource scheduler, while the opportunistic tasks are managed in a decentralized manner and used for resource oversubscription and high resource utilization – they are submitted to fill in the slack left by LRAs and regular tasks.

2.2 Distributed Long-Running Applications (DLRAs)

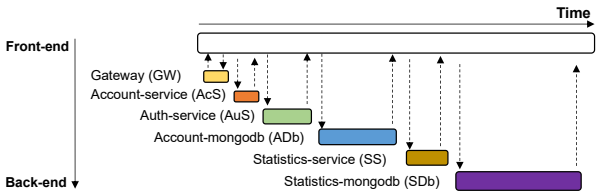
In the nature of component decoupling and distributed execution, a DLRA typically comprises multiple microservices, which are deployed on multiple nodes subject to their resource requirements. Multiple transactions within a DLRA have strong dependencies across multiple microservices. However, temporal-spatio load variability manifests over time and across nodes [28][29][30][31]. A user request (e.g., an application request, a database query, a file access operation) will transverse a collection of microservices before being responded. Therefore, end-to-end (E2E) response latency is broadly used to indicate the execution time of any operation to complete. Fig. 1 exemplifies a typical DLRA for online e-commerce store [13] which consists of nine business microservices (ranging from account related services to order management services) and seven data warehouse microservices. The arrow represents the calling dependency. After logging in the system, users can browse the inventory through *catalogue* or add items into the *cart* before finishing an *order*. *Shipping* service will also be connected with the order service so that one can check the shipping status of a given order. All information needs to be queried and fetched from underlying database services.

2.3 End-to-End (E2E) Latency in DLRAs

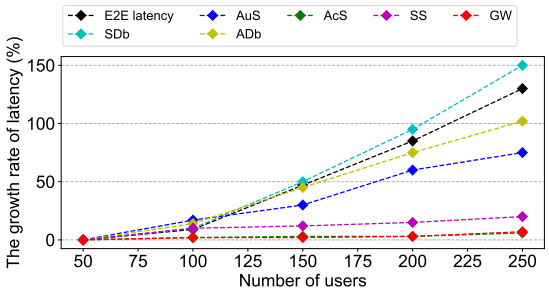
We use PiggyMetrics [32], a financial advisor app built upon microservice-based architecture, to showcase how an increase of end-to-end latency can break down into, and attribute to, the individual microservices.

Motivating Example. Fig. 2(a) shows a test case that covers microservices associated with account and statistics. The detailed calling chain is as follows: a user first launches a request to the system, and the request is then reversely routed to the *Account-service* (AcS) via the *Gateway* (GW). AS is largely dependent upon the authentication in *Authentication-service* (AuS) to complete the account verification. To obtain the relevant account information, it needs to access the local database service *Account-mongodb* (ADb). Once logged in, the user can then obtain the required statistics by initializing another requests to the *Statistics-service* (SS) and querying the back-end database *Statistics-mongodb* (SDb).

Latency Increase and Its Breakdown. Failing to handle spikes of users and requests is one of the common root causes to the latency increase. To emulate this scenario,



(a) The calling chain in the test case



(b) Latency increase breakdown

Fig. 2. Latency increase in individual microservices in PiggyMetrics

we conduct a case study by ramping up the number of users. We track the holistic request processing chain and measure the 95 percentile latency increase ratio of each individual microservice. As depicted in Fig. 2(b), different microservices exhibit different sensitivity to the growing number of users. Noticeably, two database microservices are the dominating factor to the E2E latency, while GW and SS are scalable to, and less prone to the changing system loads.

The result implicates tracking and analyzing the response latency is of great importance in QoS assurance for the long-running services and applications. Unawareness of such application-level latency at runtime could lead to higher performance interference among co-located workloads. It is thus highly imperative to localize such key components and take necessary actions of restricting and mitigating the manifestation of performance degradation.

3 Our Approach

3.1 QoS-Aware Co-scheduling

We enforce two distinct QoS management stages onto the cluster scheduling in the face of QoS violation:

- *Containment*: A QoS violation of a single microservice may propagate and lead to cascading violations across the entire system. We therefore locally restrict such propagation once the QoS measures are observably degraded within a compute node. We then delay the procedure of scheduling more batch tasks onto the node to maintain the current level of co-location and thus give way to the existing DLRAs. This intervention aims to contain the spectrum of influence and diminish the aggravation of the QoS violation.
- *Mitigation*: As opposed to the delay-execution policy used in the containment stage, it is also desirable to proactively and dynamically adjust the existing resource allocations (aka. *vertical auto-scaling*), most notably for latency-sensitive core DLRA components such as databases or data stream operators, in the event of transient but severe QoS degradation. The best-effort tasks that harvest idle resources need to be properly reclaimed.

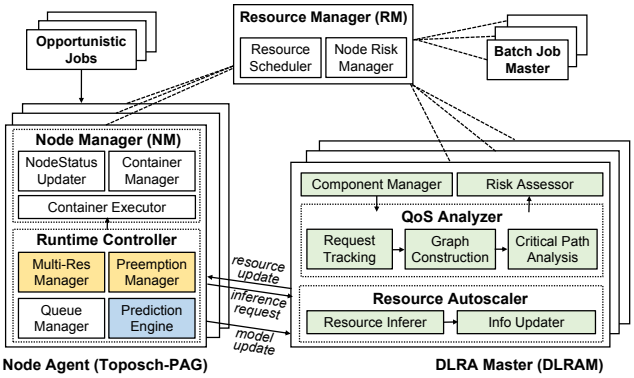


Fig. 3. Architecture overview of TOPOSCH

To fulfill the two-stage QoS management for the diverse workload co-location, we need to answer the following research questions: **[Q1]** How to localize the performance hotspots from DLRAs and identify the most vulnerable microservices? **[Q2]** How to isolate the victim microservices from the co-located batch tasks? **[Q3]** How to effectively auto-scale the containers of the risky microservices with a proper resource adjustment to ensure the required performance recovery? **[Q4]** How to minimize the cost of preempting the running batch tasks during the auto-scaling?

3.2 System Architecture

Overview. TOPOSCH is built based on the state-of-the-art open source resource management platform YARN [24] to co-schedule both latency-sensitive containers of DLRAs and tasks of batch jobs. TOPOSCH encompasses both a centralized resource manager, for high-quality resource allocation with fairness and capacity guaranteed, and a decentralized scheduling with distributed resource oversubscription extended from [16][27] to support high job throughput and high cluster utilization.

Fig. 3 describes the overall architecture of TOPOSCH and it comprises three main components: the central resource scheduler *Resource Manager* (RM), the per-DLRA manager *DLRA Master* and the per-node agent TOPOSCH-*PAG*[1], a co-resident module with the native Node Manager (NM). Furthermore, batch jobs in TOPOSCH will be separately managed according to their priority – The regular batch job will be managed by native per-job Job Master (JM). The JMs and DLRAMs are responsible for negotiating resources with the centralized RM, i.e., submitting resource requests and coordinating the resource allocation after obtaining the resource response from RM. By contrast, the opportunistic jobs will be directly submitted for improving utilization and the pertaining opportunistic tasks will be executed onto the nodes without a need of resource grant from RM.

DLRA Master (DLRAM). To align with the design of AM in YARN, we devise a specific programming framework to launch a DLRA consisting of microservices, request resources from the central RM, and provide standard functionalities of performance tracing and inter-component communication (e.g., RPC). The working mechanism is similar to the AM of DAG jobs; users can outline the topological relationships among microservices and specify the resource amount in the configuration file. At the core of DLRAM are QoS Analyzer and Resource Autoscaler:

- *QoS Analyzer* is the key component to track the request footprints generated within a certain time frame and build a weighted DAG that depicts the calling relationship. To tackle **[Q1]**, TOPOSCH exploits instrumentation to trace the footprints of all requests through each microservice. We can then monitor, extract and calculate key measures – the average sojourn (processing) time on individual microservice and average transmission time. TOPOSCH periodically constructs a request calling graph based on the microservice dependencies and localize the microservices based on critical path analysis (§4.1). Those components are regarded as QoS victims and have higher risks of further slowdown and failures. The risk information will be passed on to RM to perform preventive delay-scheduling of batch tasks (§4.2).
- *Resource Autoscaler* is the controller to infer and vertically adjust the resource allocation to each microservice container on demand to keep up with the varying QoS. In response to **[Q3]**, the aim is to work out a proper (just-enough) slice of resources, to dynamically rescuing the degraded performance whilst minimizing the impact on the neighboring jobs. To conduct the resource inference, we need a predictor to understand the sensitivity of the DLRA container to the multiple resources, i.e., the relationship between the resource allocation and the resultant QoS. TOPOSCH pre-trains an initial predictor in an offline manner, and the parameters will be synchronized to the autoscaler periodically when the resource usage on-the-fly is leveraged to tweak and update the model. We take as inputs the current resource allocation, system loads and target performance, and yield a new resource plan that can deliver a specific performance recovery. The resource change will be used for notifying the corresponding node agent and determining the detailed plans of task preemption (§5.2).

Resource Manager (RM). To raise the awareness of DLRA-level latency, RM differentiates the available nodes by the level of co-resident victim microservices. To cope with **[Q2]**, once node's risk of performance degradation is perceptible, TOPOSCH recalculates and throttles the resource amount visible to YARN capacity scheduler – according to the current risk assessment on per-node basis – so that only a fraction of real available resources can be assigned to batch tasks, thereby delaying their execution (§4.2).

Node Agent (TOPOSCH-PAG). We inherit the main functionalities of default NM for container management and status update. We devise a multi-resource manager to control the access of a variety of resources such as CPU, memory, LLC and MBW among different containers. We employ Docker containers to fulfill an isolated execution environment for the tasks. Upon receiving the request for launching a new task or DLRA container, the container executor will then launch a docker container (§5.1). In addition, as opportunistic tasks are submitted and executed in a distributed manner, such queueable tasks are allocated by the YARN distributed schedulers [33], without going through the central RM, and managed by the local queue manager of each node. In response to **[Q4]**, Preemption

1. PAG represents Performance prediction based node Agent

TABLE 1
Definitions of Identifiers

Parameter	Meaning
url	the endpoint of a DLRA-level API
serviceID	The microservice name in DLRA
requestID	The unique identifier of the request, i.e., UUID in DLRA
nextServiceID	The down-streaming microservice of the request
timestamp	Timestamp of event occurrence
eventType	Event type (i.e., send or receive)
statusCode	Event status (i.e., success or failure)

TABLE 2
Main Symbol Notations

Symbol	Descriptions
s_k	the kth microservice in the DLRA
t_k^i	inbound timestamp of the request i in s_k
$\widehat{t_k^i}$	outbound timestamp of request i in s_k
S	the collection of microservices on the critical path
ST_k^i	the sojourn time of the request i in s_k
$TT_{k,l}^i$	the transmission time of the request i between s_k and s_l
$\widetilde{ST}_k^{(u)}$	the intra-API average sojourn time of uth url in s_k
$G_{k,l}$	the set of requests between s_k and s_l
G_k	the set of requests sent to s_k
E_k	the set of error requests sent to s_k
M_n	the set of microservices running on node n
r_k, R_n	the risk of a microservice s_k and a node n
ϵ	the mini step size of resource reclamation

Manager is devised to determine which opportunistic tasks to be preempted and perform a graceful resource reclamation, upon receiving the updated information of allocation changes from the Resource Autoscaler (§5.3).

4 QoS-Aware Workload Co-scheduling

This section presents how we co-schedule the microservices of DLRAs and batch tasks by pinpointing the vulnerable microservices (§4.1) and scheduling intervention of low-priority batch tasks based on risk assessment (§4.2).

4.1 Pinpointing Vulnerable Microservices

Request Instrumentation and End-to-End Latency Tracing. To obtain as many footprints as possible, we aim to record per-request and per-microservice latency at RPC granularity. We instrument the incoming requests and output responses by tracking information including endpoints destination, inbound/outbound timestamp and request status. We use a set of identifiers to depict the information of each RPC call including *url*, *requestID*, *serviceID*, *eventType*, *nextServiceID*, *timestamp*, *statusCode* (see Table 1). We can infer the elapsed latency of a specific request within a microservice. Those traces will be aggregated into a centralized database, e.g., redis (`https://redis.io`). TOPOSCH integrates the database with DLRA's AM to ensure effective data access whilst reducing the memory consumption of RM.

The aggregated requests/responses over a period of time constitute the latency trace graph (LTG). Formally, $LTG = (\mathcal{V}, \mathcal{E}, \phi)$ comprises a set of microservice vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ denoting the interconnection links between microservices, i.e., $\phi : \mathcal{E} \rightarrow (s_i, s_j)|(s_i, s_j) \in \mathcal{V}^2 \wedge s_i \neq s_j$ where an incidence function maps each edge to an ordered pair of distinct microservices. There are a number of hierarchical execution entities in the system. A microservice provides multiple access points via RPC or RESTful APIs. TOPOSCH estimates the average sojourn time per request on microservices and transmission time between microservices.

Critical Path Analysis on the LTG. To be precise, the Mean Sojourn Time (MST) is the amount of time that a user request spends on average in each microservice; the length of MST is equal to the mean waiting time plus the mean service time. As a microservice may provide its clients multiple APIs, hundreds of thousands of requests are performed and aggregated through the API gateway before routing to specific microservices.

Through the latency instrumentation and tracing, we can easily obtain the entry and exit timestamps of a given request into a microservice. t and $\hat{t}$ represent the inbound and outbound timestamp. For a given request, the sojourn latency of a request i within microservice s_k and the transmission latency of a request j between microservice s_k and s_l can be measured by using two adjacent timestamps:

$$\begin{aligned} ST_k^i &= \widehat{t_k^i} - t_k^i, \\ TT_{k,l}^j &= t_l^j - \widehat{t_k^j}. \end{aligned} \tag{1}$$

At the core of generating LTG is to set the weight for vertices and edges. We assign the edge weight as the mean transmission latency $\overline{TT_{k,l}}$ among all requests:

$$\overline{TT_{k,l}} = \frac{\sum_{j \in G_{k,l}} (t_l^j - \widehat{t_k^j})}{|G_{k,l}|}, \tag{2}$$

where $G_{k,l}$ is the set of requests between microservice s_k and s_l, and the size is denoted by $|G_{k,l}|$. Notably, we do not differentiate the latency among different endpoints here based on the assumption of uniform RPC communication between two microservices[2]. Similarly, we assign the weight of a single vertex as the mean sojourn latency of all requests passing through the microservice s_k.

$$\overline{ST_k} = \frac{\sum_{i \in G_k} (\widehat{t_k^i} - t_k^i)}{|G_k|}, \tag{3}$$

where G_k is the set of requests to the microservice s_k.

We then divide the vertices into two distinct categories: *functional vertices* and *auxiliary vertices* to embed the mean sojourn latency $\overline{ST_k}$ and mean transmission latency $\overline{TT_{k,l}}$, respectively. To facilitate the graph algorithms, we retain main attributes including the *service_id*, relevant microservices *upstream_id/downstream_id*, and the timing information. We exploit Bellman-Ford [34] to find the longest path of LTG as the critial path.

2. It is a common practice to only adopt one type of standard RPC library such as gRPC, Apache Dubbo, Apache Thrift, etc. rather than using multiple RPC libraries. This means all requests within a DLRA will use the same underlying RPC library, and thus the latency graph can simply depend upon the mean transmission time without involving the variation due to RPC frameworks by different DLRAs and even their cross-language performance.

4.2 Batch Scheduling Intervention

4.2.1 Risk Assessment of QoS Violation

Microservice-Level Risk Assessment. The goal of microservices risk assessment is to quantitatively estimate the victim microservices on the critical path. We mainly take into account the following factors:

- *Request sojourn time.* Longer request latency indicates the pertaining microservice is prone to QoS violation, as the increased latency from the microservice would be amplified and cascaded to the whole critical path.
- *API call frequency.* Any QoS violation in the microservice with higher API call frequency will involve more requests and intrinsically influence a wider range of users.
- *Request failure rate.* Higher failure rate indicates a reduced reliability of request handling of the microservice. Without further resource adjustment, those microservices have higher risks of QoS violation.

To combine the first two factors, we consider both inter-API and intra-API request sojourn time. We calculated the weighted average sojourn time among different APIs because of the unbalanced number of requests coming into different APIs:

$$WST_k = \frac{\sum_{u \in U_k} \omega_u \widetilde{ST}_k^{(u)}}{\sum_{u \in U_k} \omega_u}, \tag{4}$$

where ω_u is the proportion, taken up by uth API url of the microservice k, of the total requests and $\widetilde{ST}_k^{(u)}$ denotes the intra-API averaging measure of uth API url. Particularly, we use the geometric mean of all requests pertaining to the url to mitigate the impact of outliers and smooth the average calculation. We then calculate the weighted sojourn proportion (WSP) to indicate the proportion and importance of the targeted microservice in the whole critical path:

$$WSP_k = \frac{WST_k}{\sum_{i \in S} WST_i}, \tag{5}$$

where S is the microservice collection on the critical path. We involve the request failure rate into the risk assessment, through the weighted request failure proportion (WFP):

$$WFP_k = \frac{|E_k|}{|G_k|}, \tag{6}$$

where the ratio of error requests are calculated. We integrate them into the risk assessment by setting a configurable weight α, which indicates a balance between sojourn latency and failure rate.

$$r_k = \alpha * WSP_k + (1 - \alpha) * WFP_k. \tag{7}$$

Node-Level Risk Assessment. TOPOSCH infers the risk level of QoS violation on a per-node basis, and thus we need to aggregate the risk score of each microservice i.e.,

$$R_n = \sum_{k \in M_n} r_k, \tag{8}$$

where M_n is the microservice set running on the node n and then forming the node-level risk R_n by normalizing the overall risk level (e.g., using min-max normalization) among all running nodes. The node risk measures over a fixed time frame are maintained within RM. RM then transforms the obtained risk information into a dynamic resource adjustment, in terms of both available resources for batch tasks and reserved resources exclusively for DLRAs.

4.2.2 Resource Reservation and Scheduling Intervention

Risk-Aware Slack Resource Reservation for DLRAs. TOPOSCH aims to achieve a dynamic and healthy co-existence of DLRAs and batch jobs with balanced performance among different forces – trading the performance of batch jobs to some extent for prioritizing the runtime latency of interactive DLRAs. Intuitively, a node with higher risk level need to reserve more slack resource for DLRAs from its available resource pool. In this context, this piece of slack resource is only visible to DLRAs and cannot be used for batch tasks for a period of time. Namely, the visible resource to batch tasks adapts to the on-the-fly risk level, according to the estimation based on Eq. 5 to Eq. 7. This intervention mechanism can avoid unnecessary batch task placement onto the node, thereby reducing the performance interference in-between.

In practice, we use a simple yet effective linear model with a reservation coefficient ν to determine the resource reservation for DLRAs on a specific node. ν represents the relationship between the risk level and the resource reservation. A higher value indicates that the amount of resource reservation is more sensitive to the change of risk level, and vice versa. The extreme case of zero ν means no dedicated slack resource for the DLRAs, i.e., completely switch-off of TOPOSCH with default YARN scheduler enabled. Correspondingly, the ratio of visible resources to batch tasks can be calculated by $1 - \nu R_n$. To make the value always valid, ν is set to be ensure νR_n between 0 and 1.

Batch Scheduling Intervention. Alg. 1 describes the procedure of resource allocation for task scheduling. YARN uses *Container*[3] as the *basic unit* of resource allocation in the scheduler of Resource Manager and then as resource lease to run a task. A *Container* will be reclaimed when a task is completed or killed. Unsatisfied *Containers* that represent the resource requests of the pending tasks will be queued in the scheduler's queue.

We select the *Container* from the waiting queue in a descending order by the waiting time and filter out a node list $\mathcal{N}$ where each node has sufficient capacity to meet the task's requirement (Lines 1-4). The scheduler will go through all potential nodes and calculate each node's *visible available* resource $\mathcal{R}_n^{vis}$ against the *real available* resource $\mathcal{R}_n^{real}$ according to the risk-aware reservation for DLRAs (Lines 10-12) if the default QoS violation policy is enabled (zero-violation policy will be discussed below).

Only if the visible resource is big enough to underpin the requested amount, the current *Container* can be assigned to the node by reusing `Assign()`, the default scheduling procedure of the native YARN (Lines 14-16). Otherwise, we will hold up the *Container* from scheduling for a given number of times (e.g., setting `maxRetryTime` as 1 indicates the delay only occurs once). This design is out of consideration of performance trade-off – we can prioritize the QoS protection without too much delay of batch task executions. Once a task petitions for resources more than `maxRetryTime`, TOPOSCH attempts to allocate resources to its *Container* as

3. In YARN's resource model, resource scheduler responds to a resource request by granting a Container. *Container* is the logical bundle of resources that grants rights to a Job Master to use a specific amount of resource (e.g., 1 Core CPU, 2GB RAM, etc.) on a specific node.

Algorithm 1: Batch Scheduling Algorithm

```
   Input: Q: the waiting queue consisting of pending batch
          Containers
1  while Q.sort(waiting_time) is not empty do
2      c ← the head Container of Q
3      // filter out available nodes
4      N ← nodes with sufficient resources for c
5      for n in N do
6          if Zero QoS violation is enabled then
7              // no resources visible to batch tasks
8              R_n^vis ← 0
9          else
10             // set the visible resources by removing reserved ones
11             R_n ← aggregate microservice-level risks via Eq.8
12             R_n^vis ← R_n^real (1 − ν R_n)
13         end
14         if R_n^vis >= c.resReq then
15             Assign(c, n)
16             break
17         else if c.retry >= maxRetryTime then
18             // the task has a locality requirement to the node n
19             if HasLocality(c, n) then
20                 Assign(c, n)
21                 break
22             // the task has no locality requirement to the node n
23             else if ! HasLocality(c, n) then
24                 n̂ ← the node with the lowest risk in N
25                 Assign(c, n̂)
26                 break
27             end
28         end
29     end
30     c.retry += 1
31 end
```

soon as possible. In this case, the Container with a data locality requirement will be directly placed, despite the fact of temporarily aggravating the QoS violation (Lines 18-21). For the *Container* without a locality requirement, TOPOSCH can relax the scope of node selection – the scheduler will choose the node with the lowest risk level to reduce the impact of co-location on the increased latency (Lines 22-26).

Parameter Setting. Finding a suitable system parameter configuration is a non-trivial task. One common practice based on our large-scale engineering experience is to initially set conservative ν for validation in a small-scale test system that has the same hardware configurations before deploying into larger-scale production. This procedure can significantly help to understand system behaviors in a controlled manner. We can set a starting point, such as 1.0, and gradually relax the parameter to allow for more co-located batch tasks by a step of 0.1 while observing the latency variations (e.g., slowdowns or failures) through daily regression tests. This procedure can help us gradually revise the configuration with a small step until all regression tests deliver stable outputs and achieve acceptable performance level of both latency-sensitive applications and batch jobs. Recent advancement in reinforcement learning can facilitate the parameter auto-tuning which is beyond the scope of this paper and will be left for future work.

Note that we also allow application-specific decision making to achieve a customized performance trade-off. Stricter violation policy, e.g., zero violation, could be applied to disallow any batch execution further and avoid worsening the QoS of the existing components of DLRAs. This could be easily implemented by setting up a global binary flag variable in the configuration file and allowing cluster administrators to specify the specific targeted scenario. If zero violation is enabled (Alg. 1 Lines 6-8), all the available resources on a node will be entirely invisible to batch tasks until all the targeted DLRA's QoS recovered.

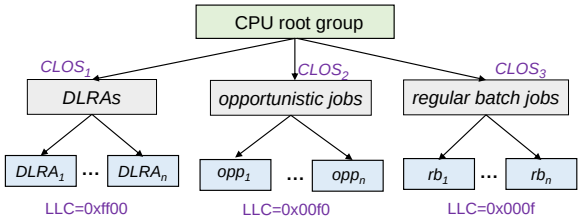


Fig. 4. Cgroup and CLOS based resource isolation

5 QoS-Aware Auto-scaling

This section addresses how to manage multi-dimensional resources and isolate resources for a given task (§5.1), how many resources to revoke for auto-scaling (§5.2), and which batch tasks to be preempted in a cost-effective manner (§5.3).

5.1 Multi-dimensional Resource Control

TOPOSCH-PAG mainly uses Linux control groups (cgroups) and Intel RDT technology to achieve fine-grained software-programmable control over the amount of resource allocation for different tasks.

We use *cgroup cpuset* subsystem to fulfill the CPU isolation: we set the `cpuset.cpus` to indicate the CPU affinity for different process group and allocate logical cores of the same CPU slot, as much as feasible, to a given microservice or batch task container. This can avoid frequent switches between CPU cores and cache contention in hyper-threading. We exploit *cgroup memory* subsystem for limiting the amount of available memory to the LRA by setting `memory.limit_in_bytes`. Fig. 4 outlines how TOPOSCH agent manages CPU and memory with the group hierarchy.

We adopt Intel RDT to monitor and control the access to LLC ways and MBW to avoid resource starvation and consequent performance degradation. We leverage Cache Allocation Technology (CAT) to group different DLRAs and batch jobs into different classes of service (CLOS) – seen as resource control tags – and then assign different capacity bitmasks (CBM) to show the amount of LLC available to each CLOS. Similarly, we use Memory Bandwidth Allocation (MBA) to specify the portion of MBW that each CLOS can access. TOPOSCH-PAG will predict the required cache ways according to the result of runtime resource re-allocation. For example, assuming that there are currently two resource control tags, CLOS1 and CLOS2. If the required cache ways are estimated to be 4 and 8, respectively, TOPOSCH will set `0x000f` for CLOS1 and `0x0ff0` for CLOS2.

5.2 QoS Prediction Engine

We investigate the relationship between multi-dimensional resources and the QoS through systematic profiling and prediction model. We can then use the model to infer how much the QoS could be mitigated by a given plan of resource re-allocation. We leverage Million Instructions Per Second (MIPS) as the QoS indicator to guarantee the measurement accuracy. Compared with Instructions Per Cycle (IPC) or

TABLE 3
Model Accuracy Comparison for MongoDB microservice

Modeling Algorithm	Accuracy Indicators		
	RMSE	*MAE*	R^2
Linear Regression	48.20	44.67	0.918
KNN	81.53	59.30	0.374
Adaboost	44.46	39.59	0.920
ElasticNet	55.40	47.54	0.718
GBRT	14.09	19.32	0.983

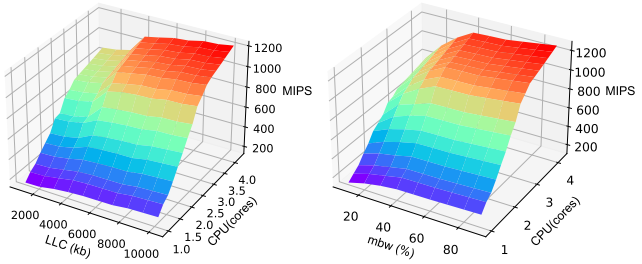


Fig. 5. The relationship between multi-resources and QoS for a MongoDB microservice

Cycles Per Instruction (CPI) [35], [27], MIPS is less dependent upon the measure of CPU frequency and the number of clock cycles in the event of frequency conversion or over-clocking techniques, and thus more accurate when an application experiences an interrupt IO.

Formally, the prediction engine take as input the normalized multi-dimensional vector $\mathcal{R}$ of existing resource allocation $(R_{CPU}, R_{mem}, R_{LLC}, R_{MBW}, i)$ for the profiled microservice i, to estimate the targeted QoS $\mathcal{Q}$ (the MIPS value). let $\mathcal{F}$ be the regression function trained and fitted on the resource and resultant QoS. As $\mathcal{F}$ is microservice-specific, each key component of DLRA will be profiled by the DLRA Master. We pre-train the prediction model in an offline training stage, similarly to existing approaches [36], [18], [37], based on a set of workload benchmarking and profiling, but will update the model parameters periodically according to the on-the-fly resource usage.

More specifically, we enumerate all possible amount of the multiple resource vectors by going through the available range of each individual resource and using a given step-size. For example, the memory allocation starts from 256MB to 4G while we increase the LLC cache ways by one way for each step. To exemplify the procedure, we showcase how prediction models are trained for a MongoDB microservice. Diverse regressors are applied into the model training including Linear Regression, k-Nearest Neighbor (KNN), Adaboost, ElasticNet and Gradient Boost Regression Tree (GBRT), etc. Model accuracy is determined through the Root Mean Square Error (RMSE) – an established measure of regression accuracy when the under-prediction error is enlarged. We also evaluate metrics such as Mean Absolute Error (MAE), and R^2 (coefficient of determination) to indicate the measurement effectiveness. Table 3 shows that GBRT has the smallest RMSE and highest R^2, indicating its minimal prediction error. We also observe a stable prediction effectiveness in GBRT with merely 1.2 RMSE deviation. This is not surprising simply due to the ensemble nature of combining several base models to produce one optimal predictive model.

The learnt model will be periodically synchronized to the corresponding DLRAM to conduct the resource re-allocation plan that can help the victim component back to the targeted QoS. Assume $\mathcal{R}$ is the current resource allocation vector and ρ is the reallocation to be enforced. Our goal is to ascertain ρ such that the subsequent QoS could reach the targeted QoS as much as possible, i.e., $\mathcal{F}(\mathcal{R}+\rho) \rightarrow (1-\varepsilon)\mathbb{Q}_{tgt}$ where ε is a small number, e.g., 0.01 or 0.05. Practically, ρ can be determined by starting from setting up the CPU steps followed by fine-tuning the memory allocation. This stems from the fact that reclaiming CPU is a much easier and dominant step – it could effectively throttle disk reads and thus speed up the memory reclamation [38]. Subsequently, a vector of memory, LLC and MBW can be then finalized to achieve the approximated QoS.

5.3 Low-cost Task Preemption

Key Idea. The eviction of running tasks is particularly expensive. Many existing solutions such as the default YARN capacity or fair scheduler forcibly kill the preempted containers without saving the task context, which would incur substantial repeated task failover and re-submission. This inevitably results in non-negligible system cost and delays the job completion. We aim to minimize the cost of task preemption by progressively reclaiming resources of opportunistic tasks, keeping task containers alive instead of interrupting them directly, without introducing noticeable performance degradation. We uniformly preempt resources from multiple tasks, a simple yet effective means to amortize the reclamation among tasks and affect each task as gently as possible. It can avoid excessive resource withdraw from one single task which may lead to dramatic execution slowdown or failures. While elaborating the characteristics of batch tasks and formalizing the preemption as an optimization problem may help to find the optimal solution to task preemption, it comes with a prohibitive implementation cost of instrumentation and profiling, and is not generally applicable (i.e. job-dependent and the huge number of tasks).

At the core of the resource reclamation is to re-throttle the resource upper limit. Reclaiming CPU can be achieved simply by revoking CPU time slices and pinning them to other tasks. We adopt pageable memory mechanisms for assigning memory to applications. We use `memory.limit_in_bytes` to reduce the upper memory limit and then `memory.memsw.limit_int_bytes` to move the memory parts beyond the limit into the swap space on disks, without terminating the tasks.

Typically, memory management can be achieved in either static (page-locked/pinned memory allocation) or dynamic (pageable/unpinned memory allocation) policies, which have their inherent advantages and limitations. While page-locked memory can achieve higher efficiency of memory r/w operations without the need of communicating with the hard drive, developers must be responsible for memory allocation and free, which brings additional management overheads and potential performance uncertainties due to misuse. On the other hand, pageable memory is more widely-adapted in modern operating systems to virtually enlarge the memory capacity. It swaps the pageable segmentations between memory and hard drive based on page replacement algorithms; it may, however, lead to

Algorithm 2: Low-Cost Task Preemption

```
Input: m: The targeted microservice
T: Opportunistic tasks queued on the node,
Q_tgt(m): Targeted QoS (MIPS) of m,
c: A pre-defined amount of resource to preempt from each task
w: A mini step of resource reclaim for each task,
1  while Q_tgt(m) is unsatisfied do
2      // get the resources to be preempted, from the autoscaler
3      ρ ← InferPreemptedResource()
4      // determine the number of preemption
5      K ← ⌈ρ/c⌉
6      // pick up K tasks to be preempted
7      T ← GetKPreemptedTasks(T, B)
8      for t in T do in parallel
9          // initialize the preemption plan for each task
10         σ ← c
11         // reclaim resource in mini-steps
12         while σ > 0 do
13             // incrementally reclaim resource
14             σ ← σ − ϵ
15             // reclaim the basic stepsize from the preempted task
16             r_t ← r_t − ϵ
17             // task preemption with reduced runtime resource
18             Preempt(t, ϵ)
19             // check the task aliveness
20             if !AlivenessCheck(t) then
21                 // blacklist the task to be exempted from selection
22                 B ← B + t
23                 break
24             end
25         end
26     end
27 end
```

performance jitter due to the variation of swap availability. We adopted swapping-based dynamic allocation, but leave the option of pinned memory to the developers, who can decide whether to transfer and store the data from the pageable segmentation to the pinned memory based on the application-specific requirement, e.g., r/w frequency.

QoS-Driven Gradual Resource Reclamation. Alg. 2 depicts the procedure of low-cost task preemption. Upon receiving the auto-scaling request – together with the resource preemption update (ρ) – from the Autoscaler of the corresponding DLRAM, Preemption Manager will launch the iteration of task preemption by choosing K opportunistic tasks from the node's queue according to a given preemption strategy and then reclaim resources from multiple task containers evenly and simultaneously (Lines 2-8). We introduce several pluggable algorithms to implement `GetKPreemptedTasks()` (detailed below). For each individual task, we revoke the pre-defined amount of resource c by multiple mini-steps to reduce the noticeable performance degradation to the preempted task. Specifically, each step of the preemption will be performed by merely depriving a certain amount ϵ at once in `Preempt()` (Lines 13-18). The value of ϵ is tuneable and should be set moderately – a big step can ensure rapid performance recovery for the DLRA but would lead to unexpected slowdown, or even failure of the opportunistic tasks. In contrast, a smaller value would delay the performance rescue and thus not ideal for real-world settings.

To minimize the risk of task failover, we introduce an aliveness checking process `AlivenessCheck()` to ensure the affected task can keep alive as much as possible. Once the task is detected to lose its heartbeat or hanged due to memory shortage, we will instantly cease the resource claim and add it in the blacklist to avoid any further task preemption (Lines 20-23). The Autoscaler in DLRAM will measure and check if the MIPS dropdown is mitigated, i.e., the targeted QoS is satisfied. If not, another round of preemption will be launched – Preemption Manager will petition for inferring the amount of resource to be reclaimed from the Autoscaler, and then the aforementioned procedure repeats.

Pluggable Preemption Strategies. The following pluggable preemption strategies are configured in TOPOSCH-PAG:

- **Random Based Scheme (RB):** Opportunistic tasks are randomly selected for preemption.
- **Longest Tasks First (LTF):** Opportunistic tasks with the longest execution time are most likely to be preempted. This policy is based on the assumption the longest task is likely to be a straggler [39], [40] compared with its peer tasks. Reclaiming resources from a task that is already slow may not incur substantial slowdown further and even accelerate the straggler handling.
- **Newest Tasks First (NTF):** The latest tasks are most likely to be preempted. The intuition is reclaiming partial resources could have limited impact on the execution progress at an early execution stage.
- **Non-locality Tasks First (NLTF):** The tasks without required data locally are most likely to be preempted. This policy assumes that such tasks may resume and execute faster in other nodes with data to be processed.

To analyze the impact of preemptive scheduling on the execution efficiency of co-located jobs, we also introduce a preemption scheme which works against the even distribution of the reclaimed resources among tasks:

- **Least Preempted Scheme (LP):** The policy will select the minimal number of tasks that can satisfy the requirement of resource reclaims. This policy is equivalent to Most Resources First (MRF) [38] where tasks with the most allocatable resources will be preempted. The intuition behind this scheme is to reclaim resource as fast as possible and reduce the scope of the affected tasks.

6 EXPERIMENTS

6.1 Experiment Setup

Hardware and Software. TOPOSCH was deployed onto a 12-machine cluster with each machine containing two 16-core (32 logical cores) Intel-Xeon(R)-Silver 4110CPU@2.10GHz, 187GB RAM, 11MB LLC and 10 Gb Ethernet network. Each node was installed with Debian 4.9.82. We have implemented TOPOSCH in 5k+ lines of Java and fully integrated with YARN 3.0-Beta1. The prediction engine is written in Python and operates as a separate container. To submit a DLRA, the topology of microservices was specified in a configuration file `DAG_SERVICE.xml`, and all requests are tracked and recorded in *redis* key-value database. Each DRLAM periodically calculates microservices' risk level at a time interval such as `60s` or `120s`.

Workloads. We emulate a mixture of realistic workloads in cloud datacenters.

- *DLRAs.* We adopt PiggyMetrics [32], a microservice architecture based financial management application, as the representative DLRA in our experiment. It consists of 12 components and each of them is encapsulated in a docker image. We embed the instrumentation and tracing

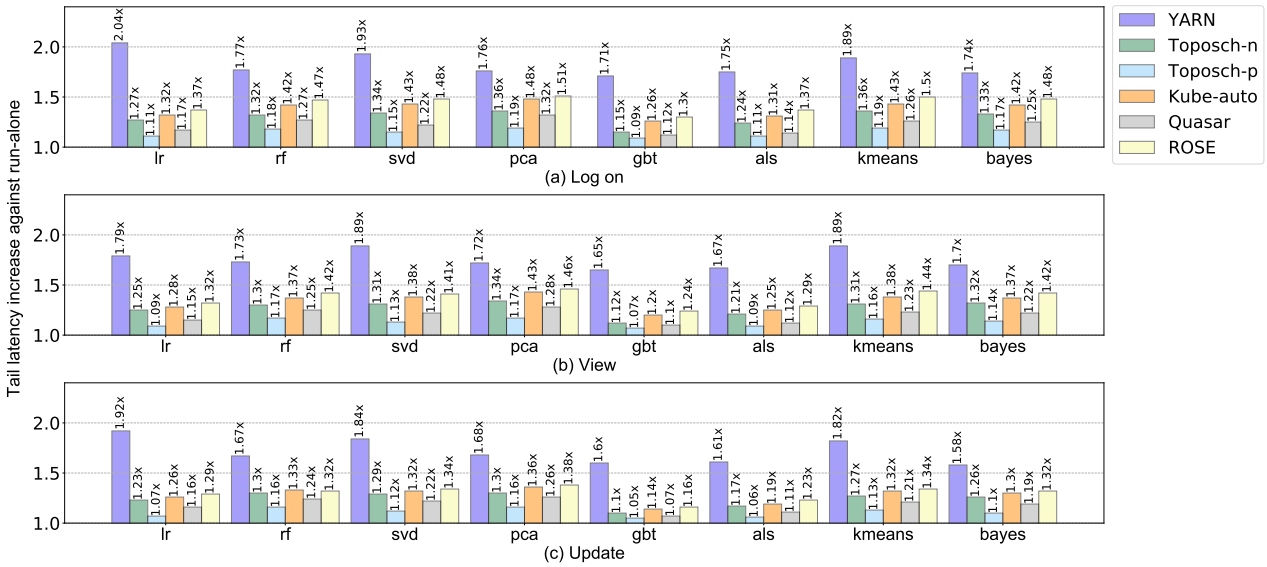


Fig. 6. Tail latency increase of (a) Log on operation (b) View operation and (c) Update operation in Account service against run-alone. The submitted PiggyMetrics instances are co-scheduled with different Spark jobs

mechanisms detailed in §4.1 into each component. We use JMeter [41] to generate workloads to PiggyMetrics and emulate the user behaviors via TPC-W [42]. There are two latency-critical components each PiggyMetrics instance: MongoDB serves as the primary database for each microservice while Kafka is used to support publish-subscribe model (pub-sub) and the messaging system among different microservices.

- *Batch jobs.* We employ Hibench [43] to generate batch jobs using Spark-2.4.6. They include 8 ML workloads: logistic regression (`lr`), random forest (`rf`), Bayesian classification (`bayes`), singular value decomposition (`svd`), principal component analysis (`pca`), gradient boosted trees (`gbt`), alternating least squares (`als`), and `kmeans`. The default configuration for each job is: *spark.dirver.memory*=512M, *spark.executor.memory* = 6G, *yarn.executor.cores* = 4, *map.parallelism* = 12, *shuffle.parallelism* = 8, *hibench.yarn.executor.num* = 60, based on the profiling of internal traces and daily practice used in Alibaba's testing clusters.

Metrics. We measure the following metrics:

- *Tail (95th Percentile) Latency* of Piggymetrics, indicates the average performance of DLRAs when handling requests.
- *Operations Per Second (OPS)* indicates the throughput of database transactions.
- *Throughput* of Kafka counts the number of messages per second, indicating the runtime QoS of stream messaging.
- *Million Instructions per Second (MIPS)* indicates, as an operating system level counter, the performance of both the database and Kafka streaming.
- *Job Completion Time (JCT)* denotes the entire completion time of a batch job.

Comparative Approaches and Methodology. Generally, to validate the effect of QoS assurance, we generate and compare two variants of TOPOSCH as an ablation study, by switching on/off the procedure of performance prediction and auto-scaling, and compare against the following two baselines:

- *YARN*: The native capacity scheduler of Apache YARN used for default co-location [24].
- *Run-Alone*: The run-alone case where Piggy Metrics or batch jobs are independently executed in an isolated environment without the related interference.
- TOPOSCH-*p*: TOPOSCH with auto-scaling enabled with performance-driven task preemption. Opportunistic tasks are throttled to prioritize the latency-sensitive components, driven by performance modeling and prediction engine.
- TOPOSCH-*n*: TOPOSCH with auto-scaling disabled without performance modeling and opportunistic preemption.

We also compare our approach with other baselines, the state-of-the-art performance-aware scheduling strategies for co-locating LRAs with batch jobs in shared clusters. For a fair comparison, we adapt their algorithms to the YARN setting and conduct their scheduling and QoS control schemes at the scheduler level:

- *Quasar*: A scheduling approach that uses collaborative filtering to predict the performance of monolithic workloads. We implemented it to guide the placement of batch tasks and microservices [44].
- *ROSE*: A performance-aware scheduling approach that harvests idle resource by opportunistic tasks and guarantees the QoS of long-running applications by tracking the application-specific performance counters such as CPI and MPKI [27].
- *Kube-auto*: Autoscaling [45] is an industry standard for elastically scaling allocations to acquire resources on demand. We implement a utilization-based auto-scaling policy adopted by Kubernetes, one of the most appealing container management systems. It triggers pod auto-scaling based on CPU or memory utilization.

We mainly evaluate TOPOSCH in terms of the overall effectiveness of workload co-scheduling, effectiveness of autoscaling, and the individual contribution of each system component. Specifically, the experiments are three-fold:

- We evaluate the performance balance of both DLRAs and

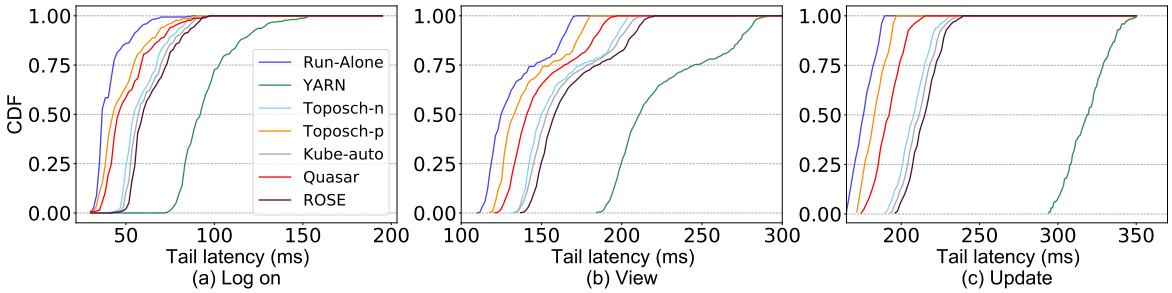


Fig. 7. The CDF of tail latency of (a) Log on operation (b) View operation and (c) Update operation in Account service when the submitted PiggyMetrics instances are co-located with Spark jobs

batch jobs. We compare two variants of TOPOSCH with the baseline approaches and the ground truth when running the DLRAs alone. (§6.2).

- We examine the effectiveness of auto-scaling with different preemptive strategies. For comparison, we evaluate our proposed schemes against the killing-based mechanism adopted by native YARN (*Kill*) and the Least Preempted scheme *(LP)* (§6.3).
- We perform several micro-benchmarks to demonstrate the performance gains and system overhead. We first evaluate the impact of multi-dimensional resource control and isolation, particularly on the key microservices. We mainly compare the proposed method against the default isolation mechanism in native YARN Node Manager (*YARN*) and the isolation mechanism provided by cpu subsystem without LLC and MBW control and isolation (*CPU-SBS*), typically adopted by cluster management systems[3], [27], [16], [26], [36]. We then analyze the parameter sensitivity, time consumption of conducting critical path analysis, and the overall system overhead. (§6.4)

Result Report. To minimize the noise, we repeat each experiment 10 times independently and compute the average running time or performance.

6.2 Overall Scheduling Effectiveness

To emulate realistic production-level workloads, we submit 100 Spark ML jobs in several rounds. 30 of them are opportunistic jobs, consisting of approximately 400 opportunistic tasks, to improve the cluster utilization. 3 PiggyMetrics application instances are initially launched. To further investigate the impact of different workloads on the effectiveness, we increase the submitted number of PiggyMetrics instances with varying resource requirements, concurrent users, and request distributions. Specifically, we measure the tail latency of three types of requests including Log on, View, and Update operations to the Account service as the performance indicator of DLRAs.

Performance of DLRAs. Fig. 6 shows the tail latency increase ratio against Run-Alone when the DLRAs are co-scheduled with different Spark jobs. Overall, TOPOSCH-p outperforms all baselines in all cases and the native YARN has the worst effectiveness in assuring QoS. For example, over all co-location scenarios, the tail latency of TOPOSCH-p is merely 1.12x on average (1.05x∼1.19x) compared with the case of Run-Alone, and can be significantly reduced by 47% on average when compared with the native YARN. This observation derives from the synergetic effect of both the batch intervention mechanism and the elastic auto-scaling mechanism for prioritizing the QoS of latency-critical workloads over other Spark jobs. When the auto-scaling mechanism is disabled, the tail latency of TOPOSCH-n increase to 1.27x of Run-Alone on average (1.1x∼1.36x) due to the single source of QoS protection by the scheduling intervention.

Regarding other baselines, Quasar ranks the second lowest in guaranteeing QoS, in the midst of TOPOSCH-p and TOPOSCH-n, due to its elaborate mechanism in profiling and performance modeling of co-located workload performance. However, it is designated for monolithic applications and thus lacks fine-grained end-to-end track of distributed components and timely adjustment of resource allocation and task scheduling at runtime. We will also demonstrate its inferior effectiveness of batch JCTs and inflexibility of handling task re-scheduling. Compared with TOPOSCH and Quasar, Kube-auto has higher tail latency due to the low accuracy of using straight-forward threshold-based control scheme to trigger auto-scaling. ROSE relies on CPI and MPKI, high-level and fluctuated performance counters, to throttle batch tasks for monolithic long-running applications without auto-scaling mechanism. This drawback limits the accuracy of QoS assurance, leading to less competitive results than other auto-scaling based approaches.

Fig 7 depicts the corresponding cumulative distributed function (CDF) of the absolute values of tail latency in three types of requests, separately, when co-scheduling with all these Spark jobs. Aligned with the observations in Fig. 6, the curve of TOPOSCH-p is the closest to Run-Alone, followed by the Quasar and TOPOSCH-n.

Performance of Batch Jobs. Fig. 8 illustrates the normalized JCT of the Spark jobs when co-located with DLRA against the jobs are executed alone. Overall, YARN and ROSE have the shortest JCT unsurprisingly, due to their native focus on batch job scheduling. Nevertheless, their capability of QoS assurance is insufficient and thus are not ideal for co-location of DLRAs and batch jobs. Compared with native YARN, the adoption of TOPOSCH-n and TOPOSCH-p result in an average increase of 17% and 26%, respectively. This phenomenon conforms to the expectation of compromising the performance of batch jobs for the QoS assurance of DLRAs. By contrast, Quasar and Kube-auto have longer average JCT because of the lack of low-cost resource reclamation when making room for DLRAs.

The result shows the trade-off achieved in our design; considering the characteristics of offline processing, such an execution delay could be acceptable. Note that one can flexibly tweak the performance balance between DLRA

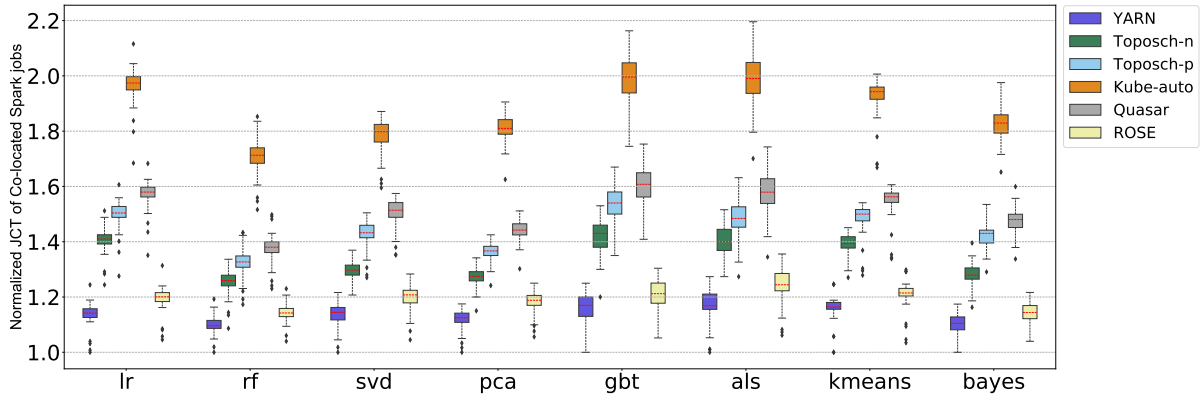


Fig. 8. JCT of Spark jobs when co-scheduled with DLRAs

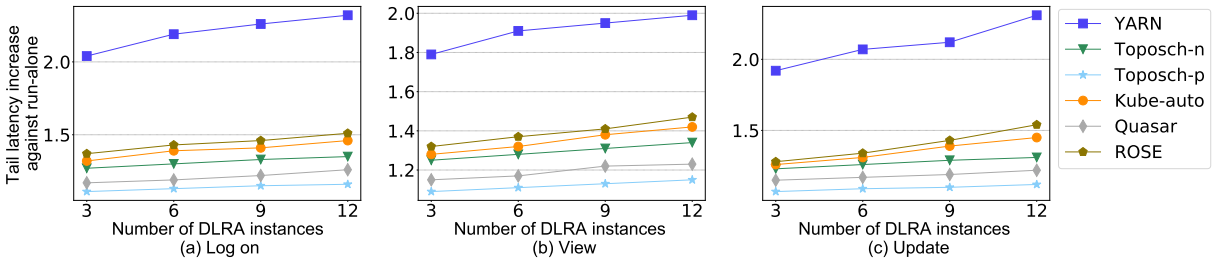


Fig. 9. Latency increase of PiggyMetrics under different number of DLRA instances when co-scheduled with Spark `lr` jobs

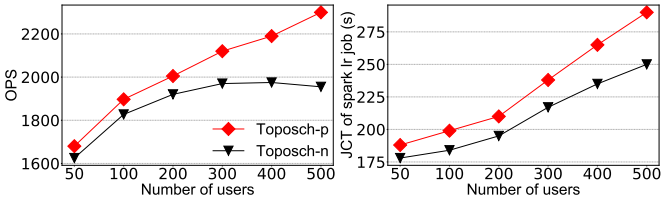


Fig. 10. Performance of MongoDB co-located with Spark `lr` jobs

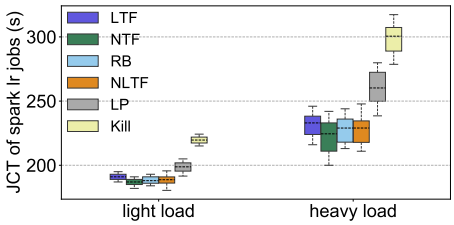


Fig. 11. JCT under different preemption schemes

and batch jobs by fine-tuning the parameter setting of the resource visibility in the containment phase and re-setting up a moderate QoS model in the mitigation phase.

Impact of different size of workloads. We increase the number of DLRA instances from 3 to 12 when co-scheduling with Spark `lr` jobs. For generalization, the DLRA instances are submitted with different resource requirements. Fig. 9 shows the increase of 95th percentile latency against the case of DLRA run-alone. All the three types of requests unexceptionally experience a upward trend. Our approach consistently outperforms other baselines when the number picks up. This indicates the performance gain of our approach does not vary much, not particularly sensitive to workload instances with different characteristics.

6.3 Autoscaling and Preemption Effectiveness

Effectiveness of Autoscaling. Our experimental study shows, as opposed to other microservices in the DLRA, the database microservices, e.g., *statistics-mongo-service*, usually exhibit more latency fluctuations, particularly in the event of load spikes, i.e., a surging increase in the user access. To instantiate this, we emulate different numbers of concurrent users, varying from 50 to 500, and measure the performance of database and its co-resident batch neighbors.

Fig. 10 presents the relationship between the growth of concurrent users, the observed OPS of the database component and the corresponding JCT of jobs on the same node. In TOPOSCH-n where the autoscaling mechanism is disabled, the OPS starts to slowdown when user concurrency becomes 200 and to drop gradually when the concurrency reaches 400; meanwhile, the JCT also climbs up promptly from the point of 200 concurrent users. By contrast, when autoscaling is enabled, TOPOSCH-p can ensure more resources reallocated to the key database microservice and retain a high service level. As a result, the OPS growth can be proportionally maintained to match the increasing demand of user access, without any performance degradation. Intrinsically, the JCT of co-resident Spark jobs will be enlarged compared with TOPOSCH-n, simply because more resources are deprived to prioritize the QoS of DLRAs.

Comparison of Difference Preemption Schemes. We investigate how different preemption schemes perform in a controlled execution environment. We create two representative co-location settings with distinct system load – roughly 80% (heavy load) and 40% (light load) utilization by placing different numbers of `lr` opportunistic jobs on the same node of the MongoDB. 100 users are created in the PiggyMetrics and concurrently access the internal microservices, particularly the MongoDB service.

Fig. 11 and Table 4 shows an overall increase of JCT and more tasks are involved in the preemption in the heavy load

TABLE 4
Task preemption rates of different auto-scaling strategies

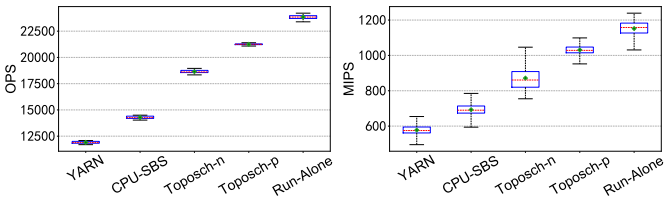
Load Level	LTF	NTF	RB	NLTF	LP
light load	12.5%	12.3%	12.3%	12.4%	7%
heavy load	18.2%	18%	18.1%	18.2%	13.5%

environment compared with light load environment. This is because DLRAs experience fiercer resource contention and need to deprive more resource from batch tasks to recover the QoS target. We can also observe larger deviations of JCTs in the heavy load cases, simply because a growing task-level execution delay or rescheduling caused by resource reclamation will affect the job-level progress in a more stochastic manner. Among all comparative schemes, the gradual preemption based schemes (LTF, NTF, RB and NLTF) significantly outperform LP and Kill-based scheme. For instance, the JCT of NTF can be reduced by 15.1% and 26.3%, respectively, compared with LP and Kill-based scheme. This is because the gradual preemption mechanism reclaims resource from multiple tasks and the mini-steps of resource reclaim can reduce the perceived performance degradation compared with the LP. Although less task containers are preempted in LP than the uniform preemption among different tasks, the MRF policy in LP can cause mandatory failover – the low CPU occupation or memory allocation sometimes fails the heartbeat communication between the running containers and RM, which eventually leads to substantial container restart. Killed-based scheme directly evicts and restarts all relevant tasks, and therefore has the longest JCT.

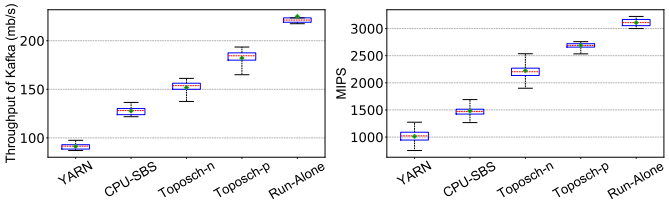
While gradual preemption based schemes have similar JCTs, NTF consistently outperforms others in both light and heavy load scenarios. This is because the impact on each individual task in NTF will be limited although more tasks are involved in the preemption in the heavy load scenario. In fact, reclaiming a thin piece of resource, particularly the CPU, from an early-stage task will have negligible impact on the overall execution. Considering the CPU slack or overclaiming is the norm rather than the exception in cluster management, the residual resource is sufficient for underpinning the task initialization and enabling the execution progress. By contrast, LTF and NLTF identify the longest tasks or the tasks without local data. However, the resource reclaim slows down those tasks further and the system-level straggler mitigation and task rescheduling will be triggered, resulting in longer JCT than NTF.

6.4 Micro-benchmarking

Performance of key latency-sensitive microservices. In this experiment, we evaluate how the key microservices in the DLRA perform in the co-located environment when different loads are enforced onto the application. We specifically count the QoS measure of the key database MongoDB and the key messaging microservice Kafka. We use *ycsb-mongo* to stress the database. Both the record count and operation count are set to be 100 million, and records take up 82GB roughly. We generate 75 million message to Kafka and each message occupies 1KB. 4 `lr` opportunistic jobs with 80 opportunistic tasks are placed onto the same node that executes the containers of these microservices.



(a) MongoDB QoS



(b) Kafka QoS

Fig. 12. QoS of key microservices under different isolation mechanisms

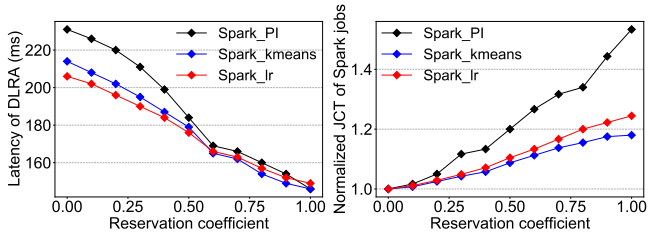


Fig. 13. The performance of DLRAs and batch jobs

As shown in Fig. 12, the proposed TOPOSCH-p outperforms other approaches in ensuring the QoS of both MongoDB and Kafka microservices. For instance, the MongoDB's OPS of TOPOSCH-p is 1.78x and 1.49x that of native YARN and CPU-SBS only approach, respectively. This is primarily due to the synergetic continuum of adaptive delay scheduling of batch tasks, effective isolation over multiple resources and the QoS assurance in the auto-scaling mechanism. In effect, regular batch tasks will give ways to the latency-sensitive DLRA components by leaving enough room when a rising risk has been detected. Meanwhile, opportunistic tasks will be moderately preempted to facilitate the QoS recovery of such latency-sensitive microservices. It can be also seen that, even switching off the auto-scaling, the OPS value of TOPOSCH-n can retain 1.56x and 1.3x that of the two baselines. This again indicates the individual contribution of the key techniques used in the containment stage (§4) and in the resource isolation (§5.1). When compared with executing MongoDB alone, the OPS of TOPOSCH-p and TOPOSCH-n are merely reduced by 10.9% and 21.9%. Similar observations can be found in the MIPS measurement and other experimentation on Kakfa.

Performance balance between DLRAs and batch jobs. As discussed in §4.2, the reservation coefficient ν is leveraged to tune the impact of node-level risk on the amount of reserved resource for microservices of DLRAs. We gradually increase its value and examine the resultant performance of DLRAs and Spark jobs. Fig. 13 shows an increasing trend in the JCT of all batch jobs when ν ramps up. Obviously, for a given node risk, an increased ν will reserve more resources for the DLRA, and thus trade more batch performance for reducing the latency of DLRAs. Specifically, the average JCT of $\nu = 1$

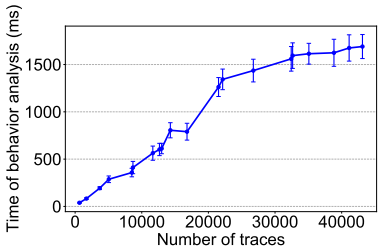


Fig. 14. Time consumption for critical path analysis

is 1.53x higher than that of $\nu = 0$ where no QoS assurance is given, i.e., the native YARN. `Kmeans` jobs and `lr` jobs experience a 23.4% and 18.2% increase, respectively. Tasks without data locality – such as the `Pi` tasks – can be delayed for a longer time. This is because of a higher likelihood of throttling or eviction to yield sufficient resources for the victim microservices. Tasks with data locality requirement such as tasks of `Kmeans` jobs and `lr` jobs, on the other hand, will be directly launched from the second retry for rapid task startup, even if the node is detected risky (depicted in Alg. 1. Accordingly, this will lead to a slightly increased latency of the co-existing microservices.

System Overhead. We analyze a per-AM overhead from DLRA Analyzer in terms of time complexity and memory consumption. *(i) Time Consumption.* As shown in Fig. 14, the time cost linearly increases but slows down when the trace number reaches 30,000. The maximal measured time is no more than 1.6 seconds. Considering the overall time consumption in the resource allocation, the incurred increase to the scheduling latency is less than 1% compared with the native YARN. *(ii) Memory Cost.* The additional memory used for fast data access using *redis* is roughly 126MB, less than 2% increase compared against native YARN. Given the intrinsic diversity in request number and arrival pattern, the number of traces for tracking latency in TOPOSCH over a given period can be customized in AM to balance the scheduling precision and the incurred overhead. It is worth noting that the overhead analysis is on a per-AM basis but can be naturally extended to cases of multiple DLRAs. For cases of multiple DLRAs, memory cost will be increased by multiple times due to *redis* is instantiated to support multi-tenancy; each AM of DLRA will independently store its own request tracing information. Each AM will be encapsulated in a Docker container, and thus the AM can separately run with stringent resource isolation and negligible interference.

7 Related Work

Resource managers in shared clusters. Cluster resource management frameworks, such as YARN [14], Mesos [15], Fuxi [3] Borg [10] are based on two-level centralized scheduling. They decouple the inter-job resource sharing and intra-job task scheduling, and the job managers need to negotiate with the centralized manager and then take charge of the job execution. Capacity Scheduling [46] or Fairness Scheduling [47] are proposed to fulfill an efficient quota-based resource sharing among multiple jobs. The objective is the enforcement of scheduling invariants for heterogeneous applications, with policing/security utilized to prevent excessive resource occupation. To further improve cluster utilization and system throughput, many other systems are based on fully decentralized design, such as Apollo [48], Omega [49], or hybrid system design, such as Mercury [26] and ROSE [16]. However, all these systems are devised towards scheduling batch analytic jobs. TOPOSCH is built upon YARN 3.0 and based on a hybrid scheduling design – our key modules are integrated with the centralized resource management framework while the opportunistic tasks are managed in a decentralized manner. The proposed mechanisms are designed to be complementary to, and can be implemented upon, the existing protocols in any two-level resource management systems.

Performance tracing and diagnostics. Many prior works are devoted into anomaly diagnosis and behavior analysis of large-scale distributed applications. They can be classified into two categories: (i) *black-box* approaches using external application states to infer and analyze the problems. [29][50] rely on a tremendous number of log files to extract performance information and infer the dependency models. [51] trains models to predict and localize latent errors in microservices based on log information comprising a set of predefined features. [52] uses fault injection to measure the execution and data flows of distributed applications and to diagnose the bottlenecks. (ii) *white-box* approaches by monitoring causality within microservices instead of inferences through statistical analysis. [53][54] infer the execution path of the application based on the static analysis and symbolic execution. [55][56] provide developers with tracing frameworks to add trace-points within the application to collect runtime footprints. In comparison, TOPOSCH uses a white-box methodology to track and trace the requests over the whole DLRA and avoids over-dependencies upon prior diagnosis conditions, typically pre-defined in black-box approaches. Instead of using the existing fine-grained tracking instrumentation, TOPOSCH adopts a light-weight tracking method to trace DLRA-level latency data, thereby significantly reducing per-DLRA runtime overhead.

QoS-aware workload co-location. The ability to co-locate jobs (i.e., execute within the same CPU or GPU) has been identified as a means to address under-utilization problem. Understanding and achieving high resource utilization or high energy efficiency for heterogeneous workloads in cloud computing is an important topic [44], [57], [58], [27], [37]. Existing work on QoS management when co-locating heterogeneous workloads has two distinct categories: (i) reducing the probability of resource contention by either granting isolated execution environments to LRAs [49][59] or adjusting task placement to reduce the resource contention on a certain node [60][11], primarily for runtime QoS of LRA. (ii) reducing performance interference caused by resource contention through performance prediction and resource inference, prioritizing the resource requests of latency-sensitive LRAs [60][17][18][19][57][61]. Many of them have applied machine learning to precisely characterize the behavioral patterns. For instance, [62], [63] leverage various ML methods such as support vector regression, random forest and extreme gradient boosting tree to predict workloads or system load changes. [64], [65] employ neural networks to estimate JCT and load fluctuation. However, they can hardly take runtime information into consideration and thus fail to provide sufficient insights into timely calibrating the runtime QoS. [36], [44] use complicated multi-

variable statistical classifiers to predict the expected interference among applications. They perform preparatory small-scale interference tests with varied levels of background applications. [19], [18] use performance index to depict contention at the time of resource allocation and conduct offline studies of the relationship between multiple resources and the resulting performance. However, they are designated to guarantee performance for monolithic applications, and not directly applicable to tackle the scheduling problem when there is tempo-spatial latency fluctuation within DLRAs. Nevertheless, the key techniques are orthogonal to our QoS prediction engine and can be modified for profiling the QoS of key microservices. By contrast, TOPOSCH leverages the distributed tracing to pinpoint the risky microservices and intervene the batch scheduling; meanwhile, TOPOSCH adopts the prediction based auto-scaling to reclaim the most suitable resources from batch tasks and minimize the cost of task preemption.

8 Conclusion

Balancing cluster utilization and applications' QoS is a non-trivial task. Microservice architecture advances the manifestation of distributed LRAs (DLRAs), comprising multiple interconnected microservices that are executed in long-lived distributed containers and serve massive user requests. Detecting and mitigating QoS violation becomes even more intractable due to the network uncertainties and latency propagation across dependent microservices.

In this paper, we present TOPOSCH, a scheduling system to adaptively co-schedule and co-locate latency-sensitive applications and batch jobs. TOPOSCH periodically identifies the risk of QoS violation for the running microservices by tracing and analyzing the critical path based on substantial requests and the consequential end-to-end latency graph. we then propose an effective delay scheduling mechanism in the scheduler for intervening the upcoming task placement that can prioritize the QoS assurance of DLRAs. A vertical auto-scaling mechanism, with the aid of resource-performance modeling and fine-grained resource access control, is proposed for promptly mitigating the QoS violation of key microservices in the DLRAs. A graceful task preemption is leveraged to ensure a low-cost task preemption and resource reclamation during the auto-scaling.

It is intricate but imperative to understand the end-to-end and tail latency in a dynamic, highly-concurrent distributed system at Internet scale. An overt observation is cloud-based LRAs have now become another main type of workloads, even more important than the conventional batch jobs. This particularly boost the requirement for strict QoS guarantees when diverse workloads are mixed. The investigated holistic approach at both the cluster-level and node-level leads to potential implications of workload co-location in many real-world domains and thus is apt for adoption in Cloud and HPC schedulers.

In the future, we plan to examine the proposed mechanism over more microservices in production environments and investigate their QoS sensitivity to fine-grained resources at large scale. We also plan to auto-learn the parameter settings by using reinforcement learning.

Acknowledgment

This work is supported by MIIT of China (2105-370171-07-02-860873), the S&T Program of Hebei (20310101D), UK EPSRC (EP/T01461X/1), Alan Turing Pilot Project and Alan Turing PDEA Program.

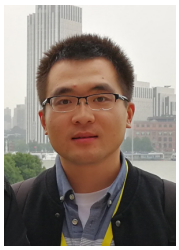
References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.
[2] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. of ACM SIGMOD*, 2015.
[3] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," in *Proc. of VLDB*, 2014.
[4] M. Zaharia, R. S. Xin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, 2016.
[5] Storm. [Online]. Available: https://storm.apache.org
[6] Flink. [Online]. Available: https://flink.apache.org
[7] Hbase. [Online]. Available: https://hbase.apache.org
[8] mongodb. [Online]. Available: https://www.mongodb.com
[9] M. Abadi, P. Barham, J. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. of USENIX OSDI*, 2016.
[10] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. of ACM EuroSys*, 2015.
[11] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *Proc. of ACM EuroSys*, 2018.
[12] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace," in *Proc. of ACM SoCC*, 2018.
[13] Y. Gan, Y. Zhang, and K. Hu, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. of ACM ASPLOS*, 2019.
[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proc. of ACM SoCC*, 2013, pp. 1–16.
[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of USENIX NSDI*, 2011.
[16] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in *Proc. of IEEE ICDCS*, 2018, pp. 949–960.
[17] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of ACM ASPLOS*, 2015.
[18] Y. Sfakianakis, C. Kozanitis, C. Kozyrakis, and A. Bilas, "Quman: Profile-based improvement of cluster utilization," *ACM TACO*, vol. 15, no. 3, pp. 1–25, 2018.
[19] P. Lama, S. Wang, X. Zhou, and D. Cheng, "Performance isolation of data-intensive scale-out applications in a multi-tenant cloud," in *Proc. of IEEE IPDPS*, 2018.
[20] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *Proc. of ACM ASPLOS*, 2014.
[21] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: reconciling scheduling speed and quality in large shared clusters," in *Proc. of ACM SoCC*, 2015.
[22] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, "History-based harvesting of spare cycles and storage in large-scale datacenters," in *Proc. of USENIX OSDI*, 2016.
[23] C. Hu, J. Zhu, R. Yang, H. Peng, T. Wo, S. Xue, X. Yu, J. Xu, and R. Ranjan, "Toposch: Latency-aware scheduling based on critical path analysis on shared yarn clusters," in *Proc. of IEEE CLOUD*, 2020, pp. 619–627.
[24] Apache hadoop yarn 3.0.0. [Online]. Available: https://hadoop.apache.org/docs/r3.1.1/index.html
[25] Kafka stream. [Online]. Available: https://kafka.apache.org
[26] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *Proc. of USENIX ATC*, 2015, pp. 485–497.

[27] R. Yang, C. Hu, X. Sun, P. Garraghan, T. Wo, Z. Wen, H. Peng, J. Xu, and C. Li, "Performance-aware speculative resource oversubscription for large-scale clusters," *IEEE TPDS*, vol. 31, no. 7, pp. 1499–1517, 2020.
[28] F. Nwanganga and N. Chawla, "Using structural similarity to predict future workload behavior in the cloud," in *Proc. of IEEE CLOUD*, 2019.
[29] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proc. of USENIX OSDI*, 2014, pp. 217–231.
[30] Z. Wen, T. Lin, R. Yang, S. Ji, R. Ranjan, A. Romanovsky, C. Lin, and J. Xu, "Ga-par: Dependable microservice orchestration framework for geo-distributed clouds," *IEEE TPDS*, vol. 31, no. 1, pp. 129–143, 2019.
[31] R. Yang, X. Ouyang, Y. Chen, P. Townend, and J. Xu, "Intelligent resource scheduling at scale: a machine learning perspective," in *Proc. of IEEE SOSE*. IEEE, 2018, pp. 132–141.
[32] Piggymetrics. [Online]. Available: https://github.com/sqshq/PiggyMetrics
[33] Hadoop yarn opportunistic containers. [Online]. Available: https://hadoop.apache.org/docs/r3.0.0/hadoop-yarn/hadoop-yarn-site/OpportunisticContainers.html
[34] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, 1958.
[35] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proc. of ACM Eurosys*, 2013, pp. 379–391.
[36] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, 2013.
[37] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE TPDS*, vol. 33, no. 1, pp. 88–100, 2022.
[38] W. Chen, X. Zhou, and J. Rao, "Preemptive and low latency datacenter scheduling via lightweight containers," *IEEE TPDS*, vol. 31, no. 12, pp. 2749–2762, 2019.
[39] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 91–104, 2016.
[40] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. of USENIX NSDI*, 2013, pp. 185–198.
[41] JmeterEB/OL. [Online]. Available: https://jmeter.apache.org.
[42] Tpc-w[eb/ol]. [Online]. Available: http://www.tpc.org/tpcw/specs.asp.
[43] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. of IEEE ICDEW 2010*, 2010, pp. 41–51.
[44] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *Proc. of ACM ASPLOS*, vol. 49, no. 4, pp. 127–144, 2014.
[45] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.
[46] YARN Capacity Scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
[47] YARN Fair Scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
[48] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proc. of USENIX OSDI*, 2014, pp. 285–300.
[49] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of Eurosys*, 2013, pp. 351–364.
[50] A. Pi, W. Chen, X. Zhou, and M. Ji, "Profiling distributed systems in lightweight virtualized environments with logs and resource metrics," in *Proc. of ACM HPDC*, 2018.
[51] X. Zhou, X. Peng, T. Xie, and J. Sun, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proc. of ESEC/FSE*, 2019.
[52] C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using targeted fault injection," *IEEE TPDS*, vol. 28, no. 2, pp. 503–516, 2016.
[53] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proc. of EuroSys*, 2010.
[54] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proc. of ACM ASPLOS*, 2010.
[55] Systemtap. [Online]. Available: https://sourceware.org/systemtap/
[56] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proc. of ACM SOSP*, 2015, pp. 378–393.
[57] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. of IEEE/ACM MICRO*, 2011.
[58] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proc. of ACM ASPLOS*, 2017.
[59] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *Proc. of USENIX ATC*, 2015.
[60] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *Proc. of ACM ASPLOS*, 2014.
[61] J. Zhu, R. Yang, C. Hu, T. Wo, S. Xue, J. Ouyang, and J. Xu, "Perphon: A ml-based agent for workload co-location via performance prediction and resource inference," in *Proc. of ACM SoCC*, 2019, pp. 478–478.
[62] C. Liu, Y. Shang, L. Duan, S. Chen, C. Liu, and J. Chen, "Optimizing workload category for adaptive workload prediction in service clouds," in *Proc. of ICSOC*, 2015.
[63] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proc. of ACM SOSP*, 2017.
[64] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer, "Prionn: Predicting runtime and io using neural networks," in *Proc. of ACM ICPP*, 2018, pp. 1–12.
[65] Q. Yang, C. Peng, H. Zhao, Y. Yu, Y. Zhou, Z. Wang, and S. Du, "A new method based on psr and ea-gmdh for host load prediction in cloud computing system," *The Journal of Supercomputing*, vol. 68, no. 3, pp. 1402–1417, 2014.

Jianyong is now an assistant professor with Department of Computing at North China Electric Power University. He received PhD degree from Beihang University in 2022 and previously a software engineer in Baidu Group. His research interests include distributed systems, and data center resource management.

Renyu Yang is an EPSRC-funded Research Fellow with the University of Leeds, UK. He was previously with Alibaba Group China and Edgetic Ltd. UK, having industrial experience in building large-scale resource scheduling systems. His research interests include reliable resource management, distributed systems and applied machine learning. He is a member of IEEE.



Xiaoyang Sun is a PhD student of the Distributed Systems and Services Group at the University of Leeds. He participated in research internships in Alibaba Group Inc., working on resource management and task scheduling on the large-scale clusters, accelerating pre-trained models in the resource-limited environment. His primary research focuses on system optimization for deep learning workflows on heterogeneous resources.

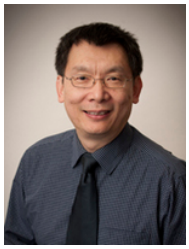
Tianyu Wo is an Associate Professor with the School of Software at Beihang University. He received his BEng and PhD Degrees both in computer science from Beihang University in 2001 and 2008 respectively. His current research interests include distributed systems, network operation systems and IoV systems. He is a member of IEEE.

Chunming Hu is a Professor and Dean of the School of Software, Beihang University. He received PhD degree from Beihang University in 2006. His current research interests include distributed systems, system virtualization, data management and processing systems.

Hao Peng is currently an assistant professor with Beijing Advanced Innovation Center for Big Data and Brain Computing in Beihang University, and School of Cyber Science and Technology in Beihang University. His research interests include representation learning, text mining and social network mining.

Junqing Xiao is currently a software engineer with Alibaba Group. He obtained MSc degree from Beihang University in 2018. His research interests include distributed systems and data center resource management.

Albert Y. Zomaya is the Chair Professor of High Performance Computing & Networking in the School of Computer Science, Sydney University and the Director of the Center for Distributed and High Performance Computing. To date, he has published more than 600 scientific papers and articles and is (co-)author/editor of 30+ books. As a sought-after speaker, he has delivered more than 250 keynote addresses, invited seminars, and media briefings. His research interests span several areas in parallel and distributed computing and complex systems. He served as EiC of IEEE Trans. on Computers for two terms (2011-2014) and the Founding EiC of the IEEE Trans. on Sustainable Computing (2016-2020). He is currently the EiC of the ACM Computing Surveys. He is a Fellow of AAAS, IEEE and IET. Also, he is an Elected Fellow of the Royal Society of New South Wales and an Elected Foreign Member of Academia Europaea.



Jie Xu is the Chair Professor of Computing at University of Leeds, the leader for a Research Peak of Excellence at Leeds, Director of UK EPSRC WRG e-Science Centre, Executive Board Member of UK Computing Research Committee (UKCRC), and Chief Scientist of BDBC, Beihang University, China. He has worked in the field of dependable distributed computing for over 30 years. He is a Steering/Executive Committee member for numerous IEEE conferences including SRDS, ISORC, HASE, SOSE and is a co-founder for IEEE IC2E, DAPPS, JCC, etc. He has led or co-led many research projects to the value of over $30M, and published in excess of 400 academic papers, book chapters and edited books. He is a Fellow of the Alan Turing Institute.