Resource Allocation with Service Affinity in Large-Scale Cloud Environments

Zuzhi Chen*, Fuxin Jiang*, Binbin Chen*, Yu Li*, Yunkai Zhang†, Chao Huang*, Rui Yang*,
Fan Jiang*, Jianjun Chen*, Wu Xiang*, Guozhu Cheng*, Rui Shi*, Ning Ma‡, Wei Zhang§, Tieying Zhang*✉
*ByteDance Inc. †University of California, Berkeley
‡Xi'an Jiaotong University §South China University of Technology
*{chenzuzhi, jiangfuxin, chenbinbin.1996, liyu.xjtu1998, huangchao.thss15, yangrui.emma, jiangfan.2017,
jianjun.chen, xiangwu, chengguozhu, shirui, tieying.zhang}@bytedance.com, †yunkai_zhang@berkeley.edu,
‡maning@xjtu.edu.cn , §zw2020@scut.edu.cn

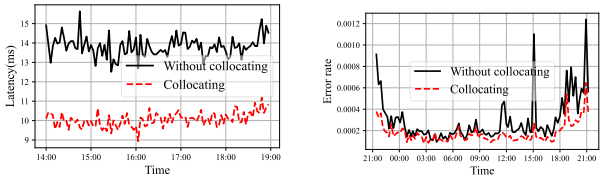
***Abstract*—Containerization has garnered substantial favor among cloud service providers. Nevertheless, the notable network overhead incurred between containers has prompted concerns within the community. In cloud resource scheduling, collocating service containers that frequently communicate to the same machine – termed "service affinity" – is instrumental in enhancing application performance. In response to this concern, we present a solution that harnesses service affinity and collocates containers to enhance the overall system performance and stability. To maximize the benefits of collocating containers, it is necessary to calculate a new schedule that optimally and efficiently maximizes service affinity, especially within the expansive domain of industry-scale cloud environments. In pursuit of this, we leverage the skewness property of affinity and machine learning to fuse solver-based algorithms, thereby assuring both quality and efficiency for problems at scale. Our methodology encompasses the partitioning of a given task into discrete subproblems, with a keen focus on resolving the most critical ones. Via a graph neural network classifier, we assign each subproblem to be solved independently using methods based on off-the-shelf solvers in our algorithm pool – namely, MIP-based, or column generation. This strategic approach enables the efficient computation of a schedule for a cloud cluster that fully optimizes the overall service affinity. We further propose a heuristic algorithm to compute executable container migration plans for practical use, facilitating the transition to the new placement where service affinity is well optimized. Our solution has been deployed in our large-scale production environment, covering over a million cores within ByteDance. Through the successful real-world production deployment, our approach exhibits an average improvement in end-to-end latency by 23.75% and a reduction in request error rates by 24.09% compared to the original system.**

***Index Terms*—resource allocation, service affinity, optimization, solver**

I. Introduction

Containerization has been widely adopted in cloud computing due to its scalability, lightweight nature, fault-isolation, and various other advantages [1]–[4]. One of the most popular applications of containerization is the microservice architecture, widely embraced by major Internet companies. In microservice architecture, applications often consist of dozens to hundreds of containerized services, each encompassing a specific functionality. These services are highly interconnected and engage in frequent data exchange. Despite the many benefits of containerization, one primary concern that troubles many companies in containerization is the substantial network overhead incurred due to the frequent remote calls between services. Furthermore, a recent trend has witnessed a shift from merely containerizing computing components to also migrating database and storage components into containers [5], [6]. These include NoSQL databases such as caching components like Redis [7], [8] and message queues like Kafka [9]. These data systems find extensive use in various data-intensive applications, where minimizing latency is paramount for enhancing user experiences. To mitigate this issue, one solution is to optimize the scheduling of containers to maximize the traffic that can communicate within the local machine, which can greatly reduce network overhead and enhance service performance.

The scheduling of containers[1] to physical machines is a fundamental resource allocation challenge in cloud computing [10]–[12]. Consider two services in a microservice cluster, when the collocation of their containers on the same machine can yield benefits, we refer to these services as having an affinity relation. Collocating containers with an affinity relation yields multiple advantages, including the potential for network bandwidth savings [10], [13], enhanced service performance [10], [14], and reduced resource consumption [15], [16].



(a) End-to-end latency (b) Stability - error rate

Fig. 1: Comparison of collocating and without collocating two containers with a service affinity relation.

Fig. 1(a) and (b) demonstrate the benefits to service performance and stability by collocating service containers with an affinity relation on the same machine. By leveraging inter-process communication (IPC) between collocated containers

✉ Corresponding Author

[1]For convenience, in this paper, the term *container* is extended to refer to any object that can be assigned to machines in the cloud, like "Container" in Yarn and "Pod" in Kubernetes.

instead of remote procedure calls (RPC) over the network, we can significantly reduce network latency associated with network I/O, minimize data transfer overhead between different hosts, and lower request error rates related to network congestion, packet loss, or connectivity issues.

Why Not Consolidation? We define **consolidation** as the process of merging several services into one, while we define **collocation** as the deployment of containers from different services on the same machine. Network efficiency is undoubtedly higher in consolidation, as it entirely eliminates network costs. However, extensively adopting consolidation is often infeasible in practice for the following reasons:

- Microservices with different programming languages and compilation environments cannot be easily consolidated. To merge different microservices into a single entity requires substantial effort to deal with various compatibility, environment requirements, and service level agreement (SLA) challenges, which is very time-consuming.
- Consolidation undermines the advantages of microservice architecture. It intensifies the coupling between services, therefore, limiting the flexibility that containerization could offer in disaster recovery, independent scaling and upgrades, and other operational aspects.

On the other hand, collocation does not encounter such complexities. It retains the independence of microservices and only involves the infrastructure team to implement.

In this paper, we propose the Resource Allocation with Service Affinity (RASA), which is a constrained optimization problem that aims to find a container-to-machine mapping that maximizes an objective function accurately characterizing the overall utility of collocating containers. Section II defines this objective function as the total gained affinity.

Despite the aforementioned benefits, collocation comes with associated challenges that need to be addressed before implementing it in practice. There are two main challenges:

- The first challenge lies in mathematically defining the problem and accurately modeling the concept of affinity to optimize it effectively. Prior studies [10], [11], [13] simply treat affinity as a Boolean relation, overlooking its full potential for optimization. In our work, we propose a model that approximates affinity based on traffic and represents the affinity relation as a graph. By transforming the optimization of affinity into a scheduling problem, we can effectively enhance the network.
- The second challenge is how to efficiently solve the proposed scheduling problem on a large-scale cloud environment. In the context of an optimization algorithm, "solution quality" pertains to the objective value of the solution, while "time efficiency" relates to the algorithm's running time. The optimization of a schedule that maximizes actual benefits necessitates that the algorithm ensures both solution quality and time efficiency. However, in practical scenarios, such as those encountered at ByteDance, where a single cluster can comprise thousands of services and machines, efficient heuristics often yield solutions with low overall affinity, while sophisticated solver-based algorithms may take days or even weeks to produce high-quality solutions. Consequently, ensuring both quality and efficiency in large-scale RASA problems poses a significant challenge.

In this paper, we present a comprehensive solution for optimizing the affinity of a large-scale containerized cluster. Our solution revolves around consistently optimizing container placement within the given cluster. Our contributions are summarized as follows:

- **Present the problem definition and mathematical formulation of RASA.** We formally define and mathematically formulate the problem of optimizing service affinity (termed RASA). This includes defining service affinity precisely, framing it as a scheduling problem with typical constraints, and introducing the objective function.
- **Propose a novel algorithm to efficiently optimize the schedule of industrial-scale clusters[2].** Our algorithm is a three-phase approach. In the first phase, we analyze service affinities as a graph, utilizing affinity skewness and graph partitioning techniques to identify key subproblems. This significantly reduces the scale of the problems at hand. In the second phase, we employ graph learning to select the appropriate solver-based approach to strike a balance between quality and efficiency, enabling us to address industrial-scale clusters previously deemed intractable. In the final phase, we propose a migration path algorithm to calculate the orders of container deletions and creations necessary for transitioning the container's placement to align with the new mapping.
- **Show advantages of our solution via extensive evaluations in both experimental and production environments**. In our experiments, our algorithm, on average, not only outperforms the state-of-the-art by 17.66% in terms of the optimization objective function, known as total gained affinity, but also achieves this with much less computation time. This showcases that the RASA algorithm excels in both quality and efficiency. In our real-world evaluations, our solution is integrated with Kubernetes and deployed in production clusters with over one million cores, resulting in a 23.75% reduction in latency and a 24.09% decrease in error requests. This demonstrates that our solution effectively enhances service performance and cluster stability.

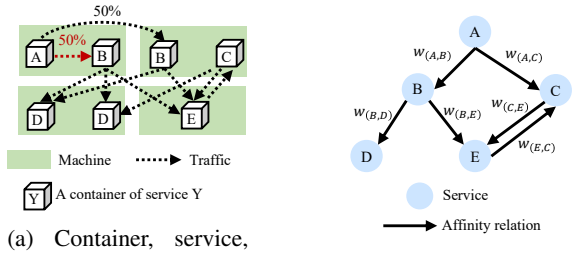
II. Problem Formulation

A. The Basics

Given a cluster, assume there are N services and M machines. Let $\mathcal{S}$ and $\mathcal{M}$ represent the sets of services and machines, respectively. To meet the SLA (Service Level Agreement), each service $s \in \mathcal{S}$ needs to instantiate d_s **homogeneous** containers in this cluster. Fig. 2(a) illustrates the fundamental concepts of services, containers, and machines. It is important to note that the concept of affinity discussed here pertains to the service-to-service level rather than the service-to-machine level. In concise terms, this concept is referred

[2]The source code of our RASA algorithm and the datasets are available in the GitHub repository [17].

to as "service affinity." This paper focuses on service affinity resulting from frequent data communication between services. Fig. 2(b) illustrates the affinity relations between services.



(a) Container, service, and machine

(b) An affinity graph

Fig. 2: Illustration of key concepts.

B. Modelling the Affinity

To describe the complex affinity topology among services, we define affinity graphs. To quantize the utility we obtain from service affinity, we introduce the concept of total affinity vs. gained affinity. Affinity is an abstract concept. In this paper, since affinity arises from frequent data communication between services, we can intuitively use the volume of traffic between two services to equate the affinity between them, aiding reader comprehension.

An **affinity graph** is a weighted undirected graph $G = \langle V, E \rangle$, where each vertex of V represents a service, as illustrated in Fig. 2(b). If $(u, v) \in E$, then the services u and v have an affinity relation.

The **weight of an edge** between two services describes the degree of their affinity. A higher weight between two services indicates that collocating their containers results in more benefits. The values of weights should be explicitly designed for the corresponding scenarios. Once again, in this paper, the affinity we focus on arises from frequent data communication between services. Therefore, in our real-world experiments, we have a metrics monitoring system to track the volume of traffic between any two services within a given cluster, and we utilize this volume of traffic as the weight of the edge between the services.

We call the total weight of an affinity graph G as **the total affinity**. For simplicity, we normalize the total affinity to 1.0. As a comparison, we quantize the realized utility from a given mapping of containers to machines as **the gained affinity**. As a concrete example, in this paper, since the affinity we consider is defined to encourage frequent data communication among services, utility is the amount of traffic that can be localized on each machine. In our case, we consider the gained affinity as *the maximum amount of traffic that can be shared within the same machine under a traffic load balancing* [18]. Here, we present its formal definition.

Definition 1 (The Gained Affinity): Consider a machine $m \in \mathcal{M}$. The affinity between services s and s' is $w_{s,s'}$ and the number of containers that the two services scheduled on machine m is $x_{s,m}$ and $x_{s',m}$ respectively, then *the gained affinity of services s and s' on machine m* is

$$a_{s,s',m} = w_{s,s'} \cdot \min\left\{\frac{x_{s,m}}{d_s}, \frac{x_{s',m}}{d_{s'}}\right\}. \quad (1)$$

The overall gained affinity is the sum of all $a_{s,s',m}$, $\forall\left(s, s'\right) \in E$ and $m \in \mathcal{M}$, where $\mathcal{M}$ is the set of all machines.

The gained affinity between service s and s' is denoted as $\sum_{m \in \mathcal{N}} a_{s,s',m}$, which signifies the maximum ratio of traffic between s and s' that can be transferred within the same machine. For instance, in Fig. 2(a), consider s as Service A and s' as Service B. The gained affinity between Service A and B is 50%, indicating that up to 50% of the traffic can be transferred within the same machine (marked with a red dashed line). We refer to this as localized traffic. The more traffic between Service A and Service B that is localized, the more requests can contribute to reducing latency and error rates between these two services. To optimize service affinity, it is natural to consider the overall gained affinity as the objective function for optimization since it quantifies the quantity of actual benefits.

In practice, there is flexibility to finetune affinity for better optimization. For instance, the cluster manager can set up multiple priority levels and ask each microservice developer to specify the priority of network performance for their services. If the priority is high, then we can assign a higher weight to the traffic as the affinity of their services. Otherwise, we assign a lower weight to the traffic as the affinity of their services towards other services.

C. Formulation of RASA

RASA is an optimization problem involving the scheduling of containers to machines in order to maximize a chosen utility function while satisfying various constraints. Let x be a matrix of size $N \times M$, where $x_{s,m}$ denotes the number of service s's containers assigned to machine m. The RASA problem is to maximize the utility via optimizing x (Notations are summarized in Tab. I):

$$\text{max.} \quad \sum_{(s,s') \in E} \sum_{g \in \mathcal{F}} a_{s,s',g} \quad (2)$$

$$\text{s.t.} \quad \sum_{m \in \mathcal{M}} x_{s,m} = d_s, \qquad \forall s \in \mathcal{S} \quad (3)$$

$$\sum_{s \in \mathcal{S}} x_{s,m} \cdot R^S_{r,s} \le R^M_{r,m}, \qquad \forall r \in \mathcal{R}, m \in \mathcal{M} \quad (4)$$

$$\sum_{s \in A_k} x_{s,m} \le h_k, \qquad \forall A_k \in \mathcal{A}, m \in \mathcal{M} \quad (5)$$

$$b_{s,m} \cdot d_s \ge x_{s,m}, \qquad \forall s \in \mathcal{S}, m \in \mathcal{M} \quad (6)$$

$$w_{s,s'} \cdot \frac{x_{s,m}}{d_s} \ge a_{s,s',g}, \quad \forall\left(s, s'\right) \in E, m \in \mathcal{M} \quad (7)$$

$$w_{s,s'} \cdot \frac{x_{s',m}}{d_{s'}} \ge a_{s,s',g}, \quad \forall\left(s, s'\right) \in E, m \in \mathcal{M} \quad (8)$$

$$x_{s,m} \in \mathbb{N}, \qquad \forall s \in \mathcal{S}, m \in \mathcal{M}. \quad (9)$$

Here, $\{a, x\}$ are the decision variables, and $\left\{r, R^S, R^M, b, d, h, w\right\}$ are the given parameters. Equation (9) restricts x to be non-negative integers.

Optimization Objective. To maximize the actual benefits obtained from collocating containers, we will use the overall gained affinity, defined in Definition 1, as the optimization objective function, as it quantifies the utility from collocation.

Constraints. A container can only be placed on a machine if it satisfies various scheduling constraints. In practical scenarios, we consider the following constraints:

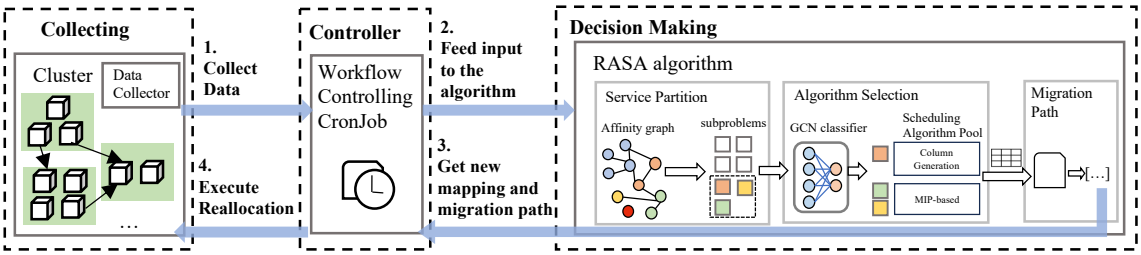


Fig. 3: Workflow of the entire system.

TABLE I: Summary of notations

Notation	Description
$\mathcal{S}$	Set of all services, where $N = \lvert\mathcal{S}\rvert$
$\mathcal{M}$	Set of all machines, where $M = \lvert\mathcal{M}\rvert$
$\mathcal{R}$	Set of all resource types
$\mathcal{A}$	Set of anti-affinity sets
$x_{s,m}$	Number of containers service s places on machine m
d_s	Number of containers for service s
$R^S_{r,s}$	Requested type r resource of each container for service s
$R^M_{r,m}$	Total type r resource of machine m
$b_{s,m}$	$b_{s,m} = 1$ if machine m can host containers of service s; 0 otherwise
h_k	The maximum number of containers of anti-affinity $A_k \in \mathcal{A}$ that a single machine can host
$w_{s,s'}$	Weight of edge (s, s') in affinity graph
$a_{s,s',g}$	Gained affinity of s and s' on machine group g

- *SLA constraints* concern the need to create a sufficient number of service instances (i.e., d_s) for each service s to adhere to the service level agreements (SLAs). The SLA constraint corresponds to (3) in the above formulation, where d_s is a constant predefined by users.
- *Resource constraints* require that if we want to place a container on a machine, then the requested resources of the container must not exceed the machine's available resources. In practical applications, multiple resource types need to be considered, including CPU, memory, network, and disk. We use $\mathcal{R}$ to represent the list of resource types. For $r \in \mathcal{R}$, we use $R^S_{r,s}$ to denote the requested r-th resource of the container for service s and $R^M_{r,m}$ to denote the total r-th resource capacity of machine m. The resource constraints correspond to (4) in the formulation, which prevents the total requested resources of all containers hosted on each machine from exceeding the total available resources of that machine.
- *Anti-affinity constraints* state that given a set of services $A_k \in \mathcal{A}$, for any machine, the number of containers from the service set A_k should not exceed a certain predefined threshold, denoted as h_k. Anti-affinity constraints prevent too many containers with a certain feature from being concentrated on a single machine. These constraints are often designed for the purposes of disaster control, fault tolerance, isolation, and security. Note that if A_k consists of only one service s, the constraints prevent service s from placing too many containers on one machine. This is often referred to as service-to-machine anti-affinity. We use the set $\mathcal{A}$ to represent all anti-affinity sets, and the anti-affinity constraints are expressed by (5).
- *Schedulable constraints* determine whether the containers of a service s can be hosted by a machine m. For a given cluster, we use a binary matrix $b_{N\times M}$ to represent the schedulable relations, i.e., the machine m can host the container of service s if and only if $b_{s,m} = 1$. Schedulable constraints are commonly related to compatibility issues in practical applications. For example, if machine m does not support the IPv4 network stack, but the service s relies on the IPv4 protocol for communication, the deployment of the container for that service is not allowed on that machine, and we will set $b_{s,m} = 0$. We abstract all these compatibility requirements as schedulable constraints and formulate them as (6).

III. System Overview

A. System Design

In this section, we provide an overview of our system, which comprises three primary components, as illustrated in Fig. 3:

- *Data Collector*. For each cluster, a data collection program gathers information at a given moment. This includes the service list, machine list, current container deployments, and traffic metrics. This data forms the cluster state, serving as input for our RASA algorithm.
- *Workflow Controlling Periodic Task (CronJob)*. The CronJob is responsible for orchestrating the workflow of the entire system. It triggers data collection and the RASA algorithm, and manages container reallocation operations.
- *RASA algorithm*. Our algorithm, referred to as the RASA algorithm, plays a central role. It determines the new mapping of containers to machines to maximize service affinity and computes the necessary container migrations to transition to the new cluster state.

The workflow for fully optimizing the cluster is as follows:

- Firstly, the CronJob triggers the data collection module of the cluster, obtaining a cluster state that includes service information, machine details, and traffic data.
- Second, the CronJob triggers the decision-making program and feeds the cluster state to the RASA algorithm.
- Third, the RASA algorithm calculates a new container-to-machine mapping and migration plan, which includes instructions for deleting and creating containers to align with the new mapping, and then returns them.
- Lastly, the CronJob reallocates the containers according to the migration plan.

Following the aforementioned process, a full optimization of the cluster is completed. However, in practice, the cluster's state may change for various reasons, such as application updates or user modifications. After these changes, the overall gained affinity may no longer be satisfactory. To address this, we continuously optimize the cluster by configuring the

CronJob to run every half an hour. This approach ensures that the overall gained affinity remains consistently high, allowing us to maximize the benefits of collocation.

B. Trade-Offs in Other Metrics

Our approach may trade off other cluster features, like load balancing, as we reallocate containers to optimize service affinity. These trade-offs are inevitable in order to optimize network performance. However, in practice, we can effectively manage the compromises resulting from our approach.

First, the extent of side effects our approach has on the overall cluster is not significant. As discussed in Section IV-B2, a few services account for the majority of the cluster's traffic, and we focus optimization and reallocation efforts on containers associated with these services. In practice, we observe that in each execution, less than 5% of the total containers are relocated. This minimal proportion of containers has negligible impacts on the overall cluster.

Second, we have implemented extra mechanisms to prevent extreme cases on other metrics:

- *Resource utilization*: The load balance is maintained by the default scheduler of the cluster, and mild imbalance is acceptable. Even if our approach causes highly skewed loads on some machines, we have a rollback mechanism that rolls back the reallocation and utilizes the default scheduler to schedule these containers on skewed machines. Furthermore, to prevent these containers from causing excessive imbalances again and churn, we will tag them as unschedulable for three days.
- *Churn*: Churn refers to the rate of container movements. The trade-off in churn is small for the following reasons: i) The maximum number of moved containers is relatively small. ii) We focus on optimizing stateless services in the cluster, which have a negligible moving cost. iii) In practice, the half-hourly CronJob will only dry-run if the gained affinity does not show a significant improvement (i.e., an improvement of over 3%) in the new schedule. Therefore, in practice, the real execution of the reallocation happens only a few times a day.

IV. RASA Algorithm

The RASA algorithm is the core component of our entire solution, as it determines the mapping of containers to machines, thereby directly influencing the actual benefits we can obtain. In this section, we will provide an overview of our RASA algorithm and then dive into the finer details.

A. Algorithm Overview

We devised the mathematical programming formulation for RASA in Section II-C. A common approach is to employ off-the-shelf solvers to solve the formulation and obtain promising results; we refer to this as the solver-based approach. However, while the solver-based approach can achieve optimal optimization quality, it often suffers from an exponential time complexity [19]–[22] that is unacceptable for large-scale problems. To address this, we propose the RASA algorithm.

The RASA algorithm in Fig. 3 shows an overview of our algorithm. We maintain a *scheduling algorithm pool* consisting of two solver-based methods - namely column generation and MIP-based algorithms. The first step is *service partitioning*, where a multi-stage partitioning algorithm splits the input data and produces several subproblems. To achieve better optimality, the second step is *algorithm selection*, where a GCN-based classifier selects the most appropriate algorithm from the pool for each subproblem. From here, each subproblem is solved independently with its selected algorithm, and then, we combine the solution of each subproblem into an overall placement of containers. Finally, a heuristic algorithm is employed to calculate a migration path comprising batches of delete and create commands. This path facilitates the transition of the current mapping to the new mapping while ensuring compliance with SLA and resource requirements.

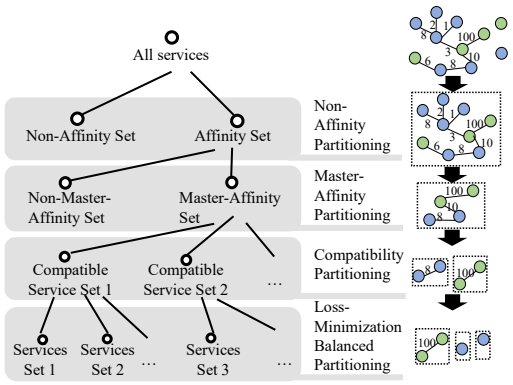


Fig. 4: Workflow of the multi-stage service partitioning with subproblem representations (each leaf node) and an example.

B. Service Partitioning

At ByteDance, each cluster comprises hundreds or even thousands of services. Computing an optimal solution to optimize the overall gained affinity of such large-scale clusters can be time-consuming. Partitioning is a commonly used technique to deal with this. The most prevalent way is equal-partitioning, which divides the problem into homogeneous subproblems [23]–[26]. However, this method is not optimal for problems with skewed properties, where certain subproblems are more important and require greater attention, while others are less significant and can be ignored. To deal with this, we propose a multi-stage service partitioning technique, where sets of services are iteratively partitioned into more disjoint sets, each representing a subproblem. The procedure of multi-stage partitioning can be represented by a hierarchy tree, as illustrated in Fig. 4. Now, we dive into the features we consider and the algorithm we use at each stage.

1) Non-Affinity Partitioning: The first stage is partitioning the original service set into two disjoint sets, the *affinity set* and the *non-affinity set*. Services with no affinity relations with other services belong to the non-affinity set. The services in the non-affinity set can never contribute to the gained affinity, so collocating containers of these services is not necessary.

2) Master-Affinity Partitioning: The second stage is to set apart the *non-master services* from the affinity set. We define the *total affinity of a service* s as $T(s)$, where $T(s) = \sum_{s' \in N(s)} w_{s,s'}$, where $N(s)$ is the neighborhood of vertex s. Without loss of generality, we assume that the services

are indexed in the order of decreasing total affinity. Given $\alpha \in [0,1]$, we call the top $\lfloor \alpha N \rfloor$ services with the largest total affinity as the *master services*, and call its complement as the *non-master services*. The master services set and the non-master services set are disjoint. In Section V-B, we will explain how we determine the value of $\lfloor \alpha N \rfloor$ for the master partitioning step in our real-world deployment.

Note that the master services are only a small subset of all services while taking up a large portion of the total affinity in several practical cases. A power-law function often approximates this skewed distribution of affinity.

Assumption 4.1: The total affinity of the s^{th} service satisfies $T(s) \propto \frac{1}{s^\beta}$ for some constant $\beta > 1$, for all $s = 1, \cdots, N$.

Prior works provide both empirical [3], [27], [28] and theoretical [29]–[31] evidences that Assumption 4.1 holds in network applications, and is further confirmed by our practical cluster data as shown in Fig. 5. With this assumption, we prove the following lemma [3].

Lemma 1: Under Assumption 4.1 with a power of $\beta > 1$, for any $\epsilon \in (0,1]$, let $\gamma = (\beta - 1)(1 - \epsilon)$. Then the total affinity of the last $N - O\left(\ln^{1-\epsilon} N\right)$ services is bounded by $O\left(\frac{1}{\ln^\gamma N}\right)$.

Given any $\epsilon \in (0,1]$. If we let $\alpha = O(\frac{\ln^{1-\epsilon} N}{N})$, Lemma 1 implies that scheduling only the top $O(\ln^{1-\epsilon} N)$ services leads to just a small loss in the objective, which is $o(1)$. In other words, the set of non-master services can only contribute minimal affinity to the gained affinity. Thus we can ignore these services to reduce the time complexity greatly.

Fig. 5: Fitting exponential and power law distributions to the total affinity distribution of 40 services in a production cluster.

3) Compatibility Partitioning: The third stage is to isolate services with different compatibility requirements, as defined by the matrix b. A machine m is compatible with a service s if and only if machine m can host the container of service s. Since services with no intersecting compatible machines can never be placed together, their containers can be scheduled separately with no loss in the objective. Compatibility partitioning is, in fact, the decomposition of the compatibility matrix b. An example is if $b = \left(\begin{smallmatrix} A & 0 \\ 0 & B \end{smallmatrix}\right)$, then compatibility partitioning will divide the service into two disjoint service sets, with A and B as their compatibility matrices, respectively. This stage will partition the master services into several even smaller disjoint service sets, without hurting optimality.

4) Loss-Minimization Balanced Partitioning: We ended up with several service sets after the previous three stages. However, the scale of each set could still be massive. Thus, for each large service set $\mathcal{S}_l$ at this stage, we further seek a balanced disjoint partition of the service set while minimizing the total affinity (weight) between different subsets. We refer to this as loss-minimization balanced partitioning. Here, "loss-minimization" means that the total affinity between services from different subsets is minimized. "Balanced" means that the number of services in different subsets is close. Specifically, we consider a partition balanced if the number of services in the largest subset does not exceed twice the number of services in the smallest subset after the partitioning.

To achieve this, we propose a heuristic that partitions the service set into balanced sets while minimizing the loss of affinity. Given a service set S_l and its affinity graph G_l, we follow the process below for $|E|$ times ($|E|$ is the edge number of the graph G_l), generating a new partition each time:

i) Randomly sample h services from S_l.
ii) For each of the h services in the affinity graph G_l, apply the breadth-first search algorithm.
iii) For each service s that is not among the h sampled services, if it is firstly visited by s' from the h services, then in this partition, s and s' will be placed in the same subset. This process results in h disjoint service subsets from S_l, forming a partition of S_l.

After the above process, we obtain $|E|$ ways of partitioning the service set S_l. We first filter out those not satisfying the balanced condition among these partitions. Then, we select the partition that minimizes the loss of affinity between different subsets as the final partition for the service set S_l. The resulting partition demonstrates a clear balanced feature, while the loss-minimization aspect stems from the intuition that each subset in a good partition contains a set of services within a neighborhood in G_l. In large-scale industrial scenarios, this heuristic algorithm excels due to its simplicity and parallelizable nature, enabling efficient performance without significant loss of affinities after the partitioning process.

5) Summary of Service Partitioning: To summarize, referring to Fig. 4, the services from the non-affinity set and non-master-affinity set only contain a minimal amount of total affinity according to the analysis in IV-B1 and IV-B2, and are therefore deemed as *trivial* services. Conversely, the services from the descendant set of the master-affinity set are considered as *crucial* services, as they encompass most of the overall affinities. Note that under Assumption 4.1, crucial services are relatively small in scale.

To construct subproblems, we need first to ignore the trivial services. Our method is to construct a new machine set, i.e., for machine m with a total resource R_m^M, if a container of trivial services s is initially hosted by machine m, then we construct a new machine with a new total resource $R_m^M - R_s^S$. After we construct the new machine set, for each type of machine specification, a specific number of machines with that specification are assigned to each crucial service set, proportional to the ratio of requested resources by that service set relative to the total requested resources by all crucial service sets. Each crucial service set and its assigned machines form a new subproblem. For trivial services, we do not need to do any further operations. It is important to note that our algorithm may not be able to successfully deploy all

[3]The full proof can be found in [18].

containers in each subproblem. However, a small number of failed deployments is considered acceptable, as they will be managed by the default scheduler in the cluster.

C. Scheduling Algorithm Pool

After the service partitioning step, we end up with several subproblems. Meanwhile, our scheduling algorithm pool possesses two algorithms tailored for different types of problems, which are *MIP-based algorithm* and *column generation algorithm*. For each algorithm, we briefly describe how it works, its characteristics, and the features of its target subproblems.

1) MIP-Based Algorithm (MIP): RASA can naturally be formulated as a mixed integer programming (MIP), as shown in Expressions (2) - (9). The MIP-based algorithm for solving RASA feeds its MIP formulation into an off-the-shelf mathematical programming solver [32], [33] directly. Note that the principles behind these solvers could be quite complicated and are thus out of the scope of this paper[4].

Characteristics: MIP-based algorithm guarantees an optimal solution (within a tolerance) but has a runtime exponential to the input size, rendering it only acceptable for small-scale problems but impractical for industry-scale applications.

Targets: If a subproblem is relatively small in scale yet has a significant total affinity, employing the MIP-based algorithm is a favorable option.

2) Column Generation Algorithm (CG): To illustrate the principles of the column generation algorithm, we introduce the concepts of *patterns* and the *cutting stock formulation* of RASA. A pattern $p \in \mathbb{N}^N$ represents a feasible placement of service containers on a machine, satisfying resource, anti-affinity, and schedulable constraints. The cutting stock formulation is an equivalent formulation of the MIP formulation (Expressions (2) - (9)) of RASA. In this formulation, decision variables determine the pattern used by each machine. Given a pattern set $\mathcal{P}_m$ of machine m, which consists of feasible patterns on machine m, let $p^{(l)} = \left[p_1^{(l)}, p_2^{(l)}, \ldots, p_N^{(l)}\right] \in \mathcal{P}_m$ be the l^{th} pattern of machine m, and $y_{m,l}$ be a binary decision variable denoting whether the container placement on machine m follows the pattern $p^{(l)}$ or not.

Algorithm 1 presents the framework of the column generation algorithm[5]. In each iteration of the *while* loop, `SolveCuttingStock` solves the cutting stock formulation. Note that `SolveCuttingStock` relaxes the integer constraints of decision variables and produces a fractional solution y for time efficiency. Then, `GenPattern` solves the formulation for generating feasible patterns and produces new patterns P' for the next iteration. The algorithm repeats this process until no more patterns with negative reduced cost are found[6], or the runtime exceeds the time-out parameter t_{max} (i.e., `IsTerminate`). The final step is to `Round` y to obtain an integral solution x.

In summary, the column generation algorithm solves RASA by iteratively generating patterns and solving the cutting stock formulation. The pattern generation process aims to improve currently obtained patterns and generate high-quality patterns with a high gained service affinity. The cutting stock formulation generates the final solution by utilizing a small set of patterns. These techniques effectively reduce the problem size compared to solving the original MIP formulation without significantly compromising optimality.

Algorithm 1: Column Generation for RASA

Input : Parameters of Expressions (2) - (9): $\{d, R^s, R^m, b, h, w\}$, t_{max}
Output: Scheduling decision $x \in \mathbb{N}^{N \times M}$
begin
  `// Initialize patterns, let` $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_M\}$
1 $\mathcal{P}_m \leftarrow \mathsf{diag}(b_{1,m}, \ldots, b_{N,m}), \forall m \in \mathcal{M}$
2 **while** `IsTerminate` $\left(y, \mathcal{P}', t_{max}\right)$ **do**
    `// Solve the cutting stock formulation`
3    $y \leftarrow$ `SolveCuttingStock` $(\mathcal{P}, d)$
    `// Generate new patterns`
4    $\mathcal{P}' \leftarrow$ `GenPattern` $(d, R^s, R^m, b, h, \mathcal{P})$
5    $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}'$
6 $x \leftarrow$ `Round` $(y, d, R^s, R^m, b, h, w, \mathcal{P})$
7 **return** x

Characteristics: The column generation algorithm can solve large-scale MIPs and often performs efficiently in practice [37], [39]–[41]. Thus, we consider column generation to have a sub-optimal optimization quality and an acceptable computation time.

Targets: For a subproblem of a medium scale with non-negligible total affinity, the column generation algorithm is a promising option since it strikes a good balance between efficiency and optimality.

3) Summary of Algorithm Pool: Considering the NP-hard nature of the RASA problem [18], the worst-case time complexity for both algorithms in our pool is exponential to the input size [18], [21]. However, in practice, these algorithms exhibit varying levels of efficiency and solution quality for different subproblems, depending on factors such as problem scale and affinity structure. By considering the features of each subproblem, assigning the most appropriate algorithm to it can ensure that the algorithm achieves both efficiency and quality.

D. Algorithm Selection

With a set of subproblems after partitioning and the two scheduling algorithms, the next step is to select an appropriate algorithm for each subproblem. The algorithm selection takes a subproblem as input and uses a graph learning model to select the more appropriate algorithm between CG and MIP.

1) Graph Classifier: Empirically, selecting between CG and MIP should factor in the finer graphical structure of the subproblem. In this case, simple heuristics would fail since it is practically infeasible to design rules that can capture all the structural information of the input. Motivated by a surge of interest in graph learning [42]–[45] in recent years, we propose a classifier based on graph convolution network (GCN) to select the appropriate algorithm.

[4]For more details, we refer readers to [34], [35] and [36].

[5]The omitted formulations of cutting stock and pattern generation can be found in our GitHub repository [17].

[6]For more details, please refer to [37] and [38].

For the subproblem k with a service set $\mathcal{S}_k \subseteq \mathcal{S}$, let $G[\mathcal{S}_k] = \langle \mathcal{S}_k, E_k \rangle$ be the sub-graph induced by $\mathcal{S}_k$ in the affinity graph G and F_k be a matrix with a size of $N \times 2$. Let $[r_s, d_s]$ be the s^{th} row of F_k, which represents the resource demand and the containers number of service s. We define $\hat{G}_k = \langle \mathcal{S}_k, E_k, F_k \rangle$ as the *feature graph* of subproblem k, which is used to select an algorithm for the subproblem.

Definition 2 (Graph Classification): Given a set of feature graphs $\mathcal{G} = \left\{\hat{G}_1, \hat{G}_2, \ldots, \hat{G}_K\right\}$ and their labels $\{\ell_1, \ell_2, \ldots, \ell_K\}$, where $\ell_k \in \mathcal{L} = \{\text{CG}, \text{MIP}\}$, $\forall k \in [K]$. We need to learn a function $f : \mathcal{G} \to \mathcal{L}$, so that $f\left(\hat{G}_k\right)$ approximates ℓ_k, $\forall k \in \{1, 2, \ldots, K\}$.

We propose to parameterize f with the following GCN model: given subproblem's feature graph $\hat{G}$ as input, it is first processed by a two-layer GCN with ReLU as the activation function. Then, graph readout is applied to get a hidden vector. Lastly, a linear layer with the softmax function calculates the probability of selecting each label based on the hidden vector.

To learn f, we obtain our train set by randomly sampling 1000 subproblems and their corresponding feature graphs from four real clusters[7]. To label a subproblem, we attempt each subproblem with the two candidate algorithms and choose the one that returns better objective within one-minute time limit.

E. Migration Path

After the algorithm selection step, we have solved all subproblems and obtained a new mapping of containers to machines. However, transitioning to this new mapping necessitates the reallocation of a portion of containers within the cluster. Container reallocation involves two steps: deleting the container from its original machine and then creating it on the target machine. This reallocation process is subject to two specific requirements:

- SLA constraints, which can be temporarily relaxed, allow each service s to maintain at least 75% of its containers alive during reallocation.
- Satisfying the resource constraints in Section II-C.

The first requirement restricts us from deleting all containers at once and subsequently creating new ones. The second requirement necessitates the deletion of some containers first to free up resources before new ones can be created. To effectively employ the RASA algorithm in practical scenarios, we must determine the optimal sequence for deleting and creating containers while ensuring compliance with the aforementioned two requirements during the reallocation. This problem is known as the *migration path problem*.

Algorithm 2 details the process of computing a migration path using the original and new mappings of containers to machines. The migration path consists of a list of command sets containing commands to delete or create containers on specific machines. For instance, $(delete, c_1, m_2)$ refers to delete the container c_1 on machine m_2.

In each iteration, the algorithm generates two command sets: one for deleting containers (l_{delete}) and another for creating containers (l_{create}). These sets are devised iteratively until the containers match the new mapping. The choice of containers to delete or create on each machine depends on the *offline ratio* (off_s) of each service $s \in \mathcal{S}$ which refers to the proportion of containers from service s that have been deleted and not yet recreated, relative to the total number of containers for service s. More specifically, the `SelectDelete` function on machine m, will firstly filter out all containers on machine m that need to be migrated, then select the one with the lowest offline ratio. Similarly, the `SelectCreate` function on machine m, filters containers that meet the following criteria: 1) they are scheduled to machine m in the new mapping, 2) they have been deleted but not yet created, and 3) their requested resource does not exceed the available resource on machine m. It then selects a container whose service has the highest offline ratio. These offline-ratio-based heuristics ensure that SLA constraints are maintained during reallocation.

Algorithm 2: Compute a Migration Path

Input : An original mapping $x_{curr} \in \mathbb{N}^{N \times M}$ and a new mapping $x_{new} \in \mathbb{N}^{N \times M}$
Output: Migration paths p
begin
1 $x_{curr} \leftarrow x_{orig}$
2 $migration\_path \leftarrow [\,]$
3 **while** $x_{curr} \neq x_{new}$ **do**
`// Get a list of containers to delete`
4 $l_{delete} \leftarrow [\,]$
for *each* $m \in \mathcal{M}$ **do**
5 $l_{delete} \leftarrow l_{delete} + \left[\left(delete, \texttt{SelectDelete}\left(m, x_{curr}, x_{new}\right), m\right)\right]$
6 $p \leftarrow p + [l_{delete}]$
`// Get a list of containers to create`
7 $l_{create} \leftarrow [\,]$
for *each* $m \in \mathcal{M}$ **do**
8 $l_{create} \leftarrow l_{create} + \left[\left(create, \texttt{SelectCreate}\left(m, x_{curr}, x_{new}\right), m\right)\right]$
9 $p \leftarrow p + [l_{create}]$
10 **return** p

Each set of the final migration path list contains a series of delete or create commands, which can be executed in parallel on different machines. However, it is important to note that the commands in the i-th set can only be executed after completing all the commands in the $i-1$-th set.

F. Running Example

Our approach begins with service partitioning to reduce the number of services that need to be considered in optimization. Fig. 4 shows our four-partitioning process, where we illustrate the properties of affinity relations in a cluster and show how we can leverage them to reduce unnecessary computations.

Initially, we perform non-affinity partitioning to identify services lacking affinity with other services. These services do not need to be reallocated for collocation. For the remaining services, we observe a common property of affinity in real-world scenarios, which is skewness. This means that a few

[7]Denoted as T1 - T4, which are different from the testing datasets M1 - M4 in Section V.

services contribute significantly to the overall affinity, while the remaining majority have minimal impact and can be disregarded. Based on this observation, we further divide the services into two subsets. Next, we perform compatibility partitioning, exemplified by IPv4 and IPv6 support. If one service requires IPv4-compatible machines and another requires IPv6 support, they cannot be deployed on the same machine. Attempting to collocate them is unnecessary; thus, we separate them into two subproblems. In some cases, even after the previous steps, a subproblem still contains a large number of services. To address this, we introduce a final partitioning step that partitions a service set into multiple smaller services while minimizing the affinity between different service sets.

After partitioning, we obtain multiple subproblems. We need to solve each subproblem independently. Two common algorithms, MIP-based and column-generation, are used to solve the formulations. These two algorithms each have their own advantages and are suitable for different problem structures, so we train a classifier model to select the best algorithm for each subproblem. Comparison experiments in Section V-C validate the effectiveness of this selection approach. The solutions of all subproblems are combined to form the final solution, in which some containers are moved to other machines in the cluster. Finally, we reallocated these containers accordingly.

V. Evaluations

We begin by describing the experiment setup in Section V-A. Then, we delve into examining the effectiveness of service partitioning and algorithm selection in Sections V-B and V-C, respectively. Next, we provide the results of gained affinity and running times in Sections V-D and V-E. Finally, in Section V-F, we present the benefits of the deployed solution in the production environment at ByteDance.

Note that from Section V-B to V-E, we evaluate the algorithms designed for solving the optimization problem of RASA, the primary criteria for comparison are the optimization objective (total gained affinity) and algorithm running times. Therefore, we only ran different algorithms and ablation experiments in simulation without executing any deployments. On the other hand, in Section V-F, to assess the overall effectiveness of our approach, we deployed our solution in a production environment and presented results related to end-to-end latency and request error rate.

A. Experimental Setup

The experiment is to validate that the RASA algorithm can efficiently compute a map of containers to machines, improving the cluster's overall gained affinity. We introduce the datasets and baselines first:

Datasets. We conduct experiments on four microservice clusters containing services, machines, and traffic (or affinity) data. All these data are collected from real traces of microservice clusters at ByteDance. Tab. II summarizes the dataset.

Baselines. We compare the following algorithms[8]:

- POP: An algorithm in [23] to efficiently solve granular resource allocation problems. However, since our problem involves services with interactions, it is not considered granular, and therefore, POP is not applicable. Nevertheless, we include POP as a baseline method since it represents one of the state-of-the-art approaches for solving large-scale optimization problems using solvers.
- K8s+: An online algorithm in [14] that simulates the Kubernetes scheduling processing - filter and score. We use a scoring function that considers service affinity.
- ApplSci19: An *extension* of the offline heuristic algorithm in [46], which is based on the min-weight graph partitioning and heuristic packing techniques.
- RASA: The full approach we proposed in Section IV.
- Original: Original assignments from the model in ByteDance production combine the idea of first-fit with the K8S's filter and score process.

[8]We used Gurobi 9.5 [32] as the solver for all the solver-relevant algorithms.

TABLE II: Scales of Experimental Datasets

Cluster Name	#Service	#Container	#Machine
Microservice Cluster 1 (M1)	5,904	25,640	977
Microservice Cluster 2 (M2)	10,180	152,833	5,284
Microservice Cluster 3 (M3)	547	3,485	96
Microservice Cluster 4 (M4)	10,682	113,261	4,365

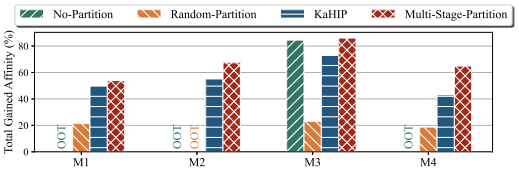


Fig. 6: Comparison of the gained affinity of different partitioning algorithms under a one-minute time-out.

B. Comparison on Service Partitioning

Different Partitioning Algorithms. To demonstrate the effectiveness of the service partitioning step, we compare different service partitioning algorithms:

- No-Partition: The approach considers the entire problem without partitioning the services.
- Random-Partition: The approach that uniformly random partitions the services set in the service partitioning step.
- KaHIP: The approach that adopts KaHIP [47], [48], which is the state-of-the-art for min-weight balanced graph cut problem, to partition the services.
- Multi-Stage-Partition: Our service partitioning algorithm as described in Section IV-B, which adopts a multi-stage service partitioning technique.

Fig. 6 shows the gained affinities under different service partitioning methods. On average, our method outperforms Random-Partition and KaHIP by 52.25% and 12.69%. For No-Partition, the program succeeds only for one small-scale cluster (M3). Our Multi-Stage-Partition outperforms all the other methods. Furthermore, we evaluated the optimality loss and time overhead associated with our Multi-Stage-Partition method. The results [18] show that the loss is generally below 12%, and the time overhead constitutes less than 10% of the total execution time of the RASA algorithm.

Different Master Ratio. Master affinity partitioning plays a crucial role in reducing computation time, so it is worth further analyzing its properties. Fig. 7 illustrate how the gained affinity and the total affinity of master services vary with different master ratios under the one-minute time-out constraint.

Our analysis and Lemma 1 indicate that the master ratio α should be set to $O((\ln^{1-\epsilon} N)/N)$. In practice, we empirically set the master ratio $\alpha = 45 \cdot (\ln^{0.66} N)/N$. Fig. 7 also plots the master ratios we use in our algorithm. Our chosen master ratio is close to the optimal value for all clusters.

In general, as the master ratio increases, the total affinity of master services quickly approaches 1.0, and the gained affinity increases to a peak before either plateauing for small and medium-scale clusters or decreasing for large-scale clusters. The decrease in gained affinity is due to the limited time frame of 1 minute for solving large-scale clusters, which is insufficient for our algorithm to explore the search space and obtain a good solution fully.

C. Comparison on Algorithm Selection

To demonstrate the effectiveness of the algorithm selection step, we compare different algorithm selection settings.

- CG: Our approach except that the graph classifier is replaced by labeling every subproblem with CG.
- MIP: Our approach except that the graph classifier is replaced by labeling every subproblem with MIP.
- HEURISTIC: An empirical heuristic that calculates the average container number of all services and the average machine number of all machine types. If the former is greater than the latter, then we select CG. Otherwise, we select MIP.
- MLP-BASED: The approach that takes the mean value of each feature for all services and then processes it by a multi-layer perception (MLP) [49]. This method completely ignores the topology of the affinity graph.
- GCN-BASED: The approach described in Section IV-D which adopts the GCN-based algorithm selection.

Fig. 8 shows the gained affinities under different algorithm selection methods. Except GCN-BASED, no method can achieve the best gained affinity across all clusters. We find that exclusively selecting either CG or MIP does not yield the best optimization results, as CG outperforms MIP in certain clusters and less effectively in others. HEURISTIC is derived from our practical experience, which works well for clusters except for M2 and M4. The reason is that M2 and M4 are large clusters with complex features in graph structures and feature matrices, and the rule is not enough to capture all these features. Similarly, the MLP-BASED approach, which ignores the affinity topology compared to GCN-BASED and fails to identify the best choice for clusters like M1 and M3. To summarize, GCN-BASED is a general algorithm selection approach that works well across different datasets.

D. Optimization Quality Results

To get a fair comparison of the optimization quality, we compare POP, APPLSCI19, K8S+ and RASA under the exact time-out requirements. Considering the service level objective (SLO) in practical scenarios, we set the time-out to one minute. If an algorithm cannot produce a result in one minute, we will mark it as OOT (Out of Time).

Fig. 9 illustrates the gained affinity of different algorithms. For the Microservice dataset, RASA improves the gained affinity by more than $13.83\times$ compared to the ORIGINAL schedule on average. RASA also outperforms POP, K8S+ and APPLSCI19 by 54.91%, 54.69% and 17.66% on average, respectively. In summary, RASA achieves the best optimization quality for all clusters under a one-minute time-out.

POP fails due to its random partitioning, which causes significant loss. ORIGINAL and K8S+ also perform suboptimally since they are both online heuristic algorithms with limited ability to optimize schedules. As for APPLSCI19, though the graph partitioning performs well, the heuristic packing after each partitioning frequently fails since the original algorithm can only deal with one machine size. The heuristic packing did not consider problems with multiple machine types.

E. Efficiency Results

We now investigate the runtime of different algorithms from two aspects: (1) the minimum possible runtime, and (2) the solution quality under the same time-out constraint. For reference, from our practical experience, an algorithm that produces a schedule with a runtime (consider the 95th percentile or p95) under 60 seconds is practically valuable. Those with a runtime under 5 minutes could cause minor errors when scheduling due to the nonnegligible changes in the cluster snapshot, while those with a runtime over 5 minutes are considered impractical.

For RASA and POP, the algorithm iteratively refines its solution until it converges to optimality. If we halt the program in the middle of execution, the algorithm can still return the current best result. Thus, by setting a time-out parameter, we can control the max runtime of RASA and POP. This allows us to plot how the optimization quality changes over runtime for RASA and POP. In contrast, manipulating the max runtime is rather difficult for APPLSCI19 and K8S+ since these two algorithms do not produce any feasible solutions unless the algorithm fully executes.

Fig. 10 reveals the relations between the optimization quality and the runtime for RASA and POP. A point closer to the top-left is desired, meaning its quality is superior, and runtime is shorter. As shown, RASA outperforms the other algorithms in terms of both quality and efficiency. Note that the improvement of solution over time is not significant for both RASA and POP, but for completely opposite reasons. For RASA, partitioning is able to separate out small subproblems that have high affinity. Employing a minimum time limit can already achieve satisfactory results, and increasing the time limit further does not yield significant improvements. However, for POP, the subproblems after partitioning remain large in scale, resulting in inferior solutions compared to RASA. Increasing the time limit leads to only marginal improvements since the search space is too large.

F. Performance Improvements in Production

The solution described in Section III has been deployed in production at ByteDance for several critical clusters, covering resources totaling more than a million CPU cores. In this section, we present the results observed from our deployed solution in production environments. In the production setting, we adopted an altered RPC framework that allows collocated

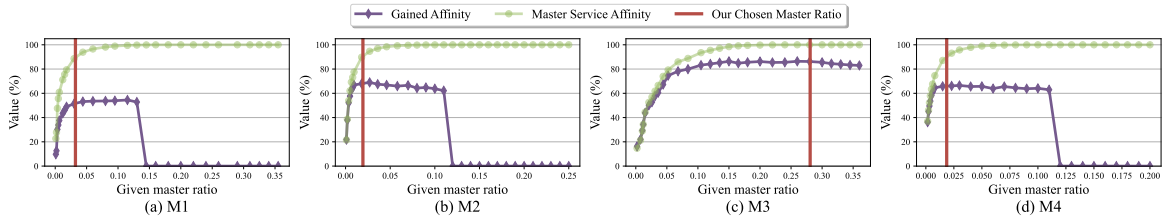


Fig. 7: Under one-minute time-out constraint, the gained affinity and the total affinity of master services under different given master ratios, and the value of our chosen master ratios.

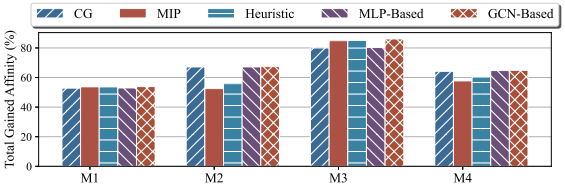


Fig. 8: Comparison of the gained affinity of different algorithm selections under a one-minute time-out.

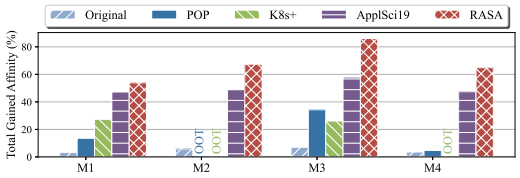


Fig. 9: Gained affinity comparisons of different algorithms for RASA under a one-minute time-out.

containers to communicate through inter-process communication (IPC) instead of using the network.

To validate the effectiveness of improving service performance and stability through the deployment of RASA, we present the results of end-to-end latency and request error rate (all metrics are normalized with a maximum value of 1.0) for both the WITH RASA and WITHOUT RASA cases. WITH RASA refers to containers whose placement is optimized with the RASA algorithm, where more of them are collocated. In contrast, WITHOUT RASA refers to the containers without optimizing service affinity, which is essentially the ORIGINAL algorithm that we described in Section V-A. To show the optimization upper bound that could be achieved, we also present results of ONLY COLLOCATED. For a service pair, certain requests between them are routed between collocated containers on the same machine. ONLY COLLOCATED solely considers the latency and error rate of these collocated containers, providing a close approximation to the scenario where all containers are collocated on a single machine.

Fig. 11 and 12 demonstrate the improvements in end-to-end latency and request error rate achieved by the RASA algorithm for four critical business service pairs in production. Each subplot represents the average metrics of all containers of the service pair with the RASA algorithm optimizing (solid line) and without the RASA algorithm optimizing (blue dashed line) its placement. The relative improvements in latency range from 16.77% to 72.16%, and the relative improvements in error rate range from 13.27% to 64.42%. These improvements are due to the RASA algorithm enabling more containers to be collocated, allowing more requests to benefit from collocation, and resulting in improved average latency and error rates.

Fig. 13 shows the improvements of all services that we have currently optimized in a cluster. We utilize a weighted metric encompassing all the services considered in our RASA algorithm. The weight assigned to each service pair in the metric is based on its queries per second (QPS) relative to the total QPS of all services. The WITH RASA weighted latency and error rate demonstrates a significant 23.75% improvement and a 24.09% reduction compared to the WITHOUT RASA, respectively. This further validates that with the optimization of the RASA algorithm and more containers collocated, we achieve greater improvements in overall latency and error rate.

Furthermore, for the four individual service pairs as well as for the overall cluster, the average absolute gap of WITH RASA to ONLY COLLOCATED is less than 10% for both latency and error rate. These results show that the affinity has been sufficiently optimized.

VI. RELATED WORK

Resource allocation for clusters. Resource allocation for clusters has been studied extensively in the past years. Most of the previous research focuses on heuristic algorithms [50]–[59]. An example is Eigen [59], which proposes a hierarchical resource management system and three resource optimization algorithms based on heuristics to improve resource allocation ratio without hurting resource availability.

More recently, adopting solvers in resource allocation gained popularity due to its capability of producing high-quality solutions [10]–[12], [23], [60]–[63]. [10] uses MIP to model the scheduling of containers for long-running applications and adopt an ILP-based solver to solve the MIP. [12] studies the stochastic bin packing problem derived from container scheduling, where they reformulate the problem and employ a solver-based cutting stock approach. The most recent work [64] studies the tenant placement problem in a Database-as-a-Service cluster with a focus on minimizing the probability of failovers. The authors proposed mathematical programming models to address this problem and utilized solvers to solve it. The results demonstrate significant advantages of this approach over previous state-of-the-arts. Due to the poor efficiency of solvers, only a small portion of solver-based research focuses on large-scale clusters, and they are often equipped with some acceleration techniques to meet the efficiency requirements. For instance, RAS [11] uses MIP to model the capacity reservation problem of large-scale clusters. Multi-phase solving and variable aggregation techniques are applied to meet the SLO of

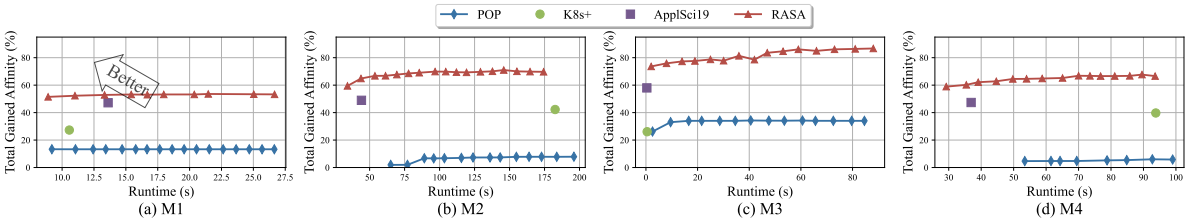


Fig. 10: The optimization quality (concerning total gained affinity) under different runtimes.

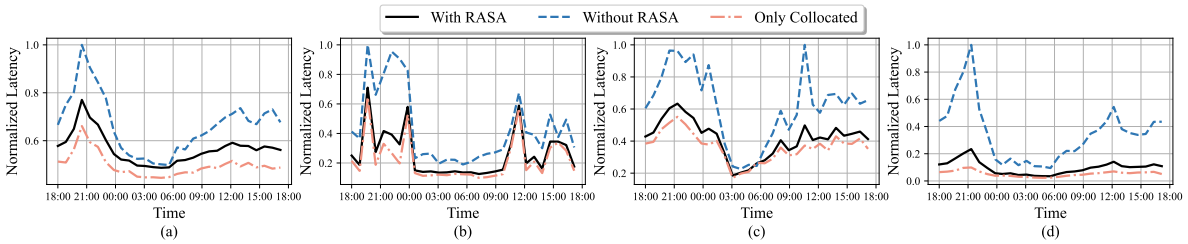


Fig. 11: Comparison of (normalized) end-to-end latency for four critical service pairs in production.

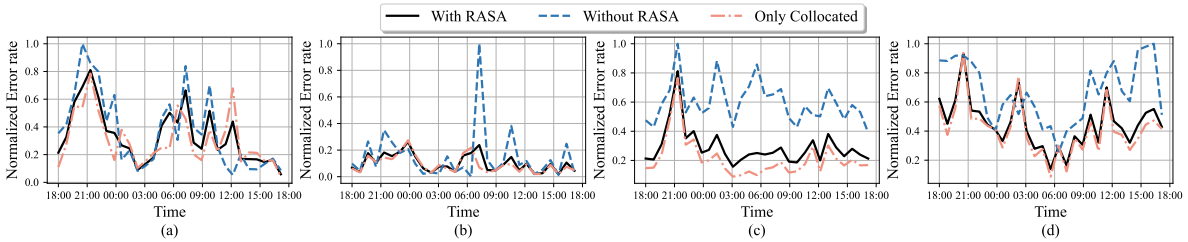


Fig. 12: Comparison of (normalized) request error rate for four critical service pairs in production.

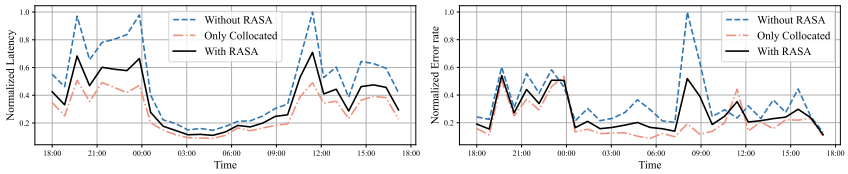


Fig. 13: Comparison of weighted end-to-end latency and error rate for considered services in production.

solving within one hour. POP [23] proposes a general solution for granular resource allocation problems that can produce high-quality solutions efficiently.

Resource allocation with service affinity. Previous works on service affinity in the cloud often refer to collocating the containers of services or virtual machines to the same machine or the same group of machines to minimize cross-group network communication. For example, as [65] studies the virtual machine placement problem and [11] studies capacity reservation problem in the cloud, they both take reducing the cross-datacenter traffic into account. Moreover, [10], [13], [46], [66] study the container scheduling, where they instead consider minimizing the inter-machine traffic between containers. Besides that, a popular container orchestration system, Kubernetes, also provides an affinity feature [14], [67], which allows the developer to customize their affinity requirements.

VII. Conclusion

Service affinity can greatly improve stability and enhance system performance. However, such a topic with tremendous business and environmental impact remains largely understudied. To fill this gap, we present a formulation of this problem as Resource Allocation with Service Affinity (RASA).

On top of it, we propose a novel approach that utilizes a multi-stage partitioning technique to divide a given task into several subproblems. With a GCN classifier, each subproblem, based on its scale and impact on the objective, is assigned to be solved by an algorithm from the candidate pool. Notably, we show that exact algorithms only need to be applied to a small fraction of services with top affinities to guarantee asymptotic optimality. We further propose a heuristic algorithm to compute a migration path that can be directly executed, transitioning to the new placement where service affinity is well optimized. Experimental results show that our algorithm achieves both quality and efficiency on real traces, and the successful large-scale production deployment shows that our solution significantly improves stability and service performance. With the trend of moving data systems to containers, our solution can improve the performances of these data systems.

In general, adopting solvers for large-scale problems is rare due to efficiency concerns, despite their enormous potential. In future works, we aim to explore more high-quality-high-efficiency solver-based algorithms in databases, cloud computing, and distributed systems.

REFERENCES

[1] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*. ACM, 2019, pp. 3–18.

[2] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *ICSA Workshops*. IEEE Computer Society, 2017, pp. 243–246.

[3] H. Aragon, S. Braganza, E. F. Boza, J. Parrales, and C. L. Abad, "Workload characterization of a software-as-a-service web application implemented with a microservices architecture," in *WWW (Companion Volume)*. ACM, 2019, pp. 746–750.

[4] L. Baresi and M. Garriga, "Microservices: The evolution and extinction of web services?" in *Microservices, Science and Engineering*. Springer, 2020, pp. 3–28.

[5] "DOCKER: Persist the db," https://docs.docker.com/get-started/05_persisting_data, 2024.

[6] "KUBERNETES: Persistent volumes," https://kubernetes.io/docs/concepts/storage/persistent-volumes, 2024.

[7] "AWS: Amazon memorydb for redis," https://aws.amazon.com/cn/memorydb/, 2024.

[8] "REDIS: Run redis stack on docker," https://redis.io/docs/stack/get-started/install/docker, 2024.

[9] "APACHE KAFKA: Quick start - docker," https://developer.confluent.io/quickstart/kafka-docker, 2024.

[10] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *EuroSys*. ACM, 2018, pp. 4:1–4:13.

[11] A. Newell, D. Skarlatos, J. Fan, P. Kumar, M. Khutornenko, M. Pundir, Y. Zhang, M. Zhang, Y. Liu, L. Le, B. Daugherty, A. Samudra, P. Baid, J. Kneeland, I. Kabiljo, D. Shchukin, A. Rodrigues, S. Michelson, B. Christensen, K. Veeraraghavan, and C. Tang, "RAS: continuously optimized region-wide datacenter resource allocation," in *SOSP*. ACM, 2021, pp. 505–520.

[12] J. Yan, Y. Lu, L. Chen, S. Qin, Y. Fang, Q. Lin, T. Moscibroda, S. Rajmohan, and D. Zhang, "Solving the batch stochastic bin packing problem in cloud: A chance-constrained optimization approach," in *KDD*. ACM, 2022, pp. 2169–2179.

[13] C. Mommessin, R. Yang, N. V. Shakhlevich, X. Sun, S. Kumar, J. Xiao, and J. Xu, "Affinity-aware resource provisioning for long-running applications in shared clusters," *CoRR*, vol. abs/2208.12738, 2022.

[14] "KUBERNETES: Assigning pods to nodes," https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node, 2024.

[15] S. Sudevalayam and P. Kulkarni, "Affinity-aware modeling of cpu usage for provisioning virtualized applications," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 139–146.

[16] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration," in *2010 39th International Conference on Parallel Processing*, 2010, pp. 228–237.

[17] "Service-affinity-scheduling," https://github.com/bytedance/Service-Affinity-Scheduling, 2024.

[18] "Supplementary materials - resource allocation with service affinity in large-scale cloud envoronments," https://github.com/bytedance/Service-Affinity-Scheduling/blob/main/supplementary-materials/supplementary-materials.pdf, 2024.

[19] "SCIP: Solving constraint integer programs," https://www.scipopt.org/, 2024.

[20] "Branch and cut in cplex," https://www.ibm.com/docs/en/icos/12.10.0?topic=concepts-branch-cut-in-cplex, 2024.

[21] A. Basu, M. Conforti, M. D. Summa, and H. Jiang, "Complexity of branch-and-bound and cutting planes in mixed-integer optimization - II," *Comb.*, vol. 42, no. 6, pp. 971–996, 2022.

[22] J. E. Mitchell, "Integer programming: Branch and cut algorithms," in *Encyclopedia of Optimization*. Springer, 2009, pp. 1643–1650.

[23] D. Narayanan, F. Kazhamiaka, F. Abuzaid, P. Kraft, A. Agrawal, S. Kandula, S. P. Boyd, and M. Zaharia, "Solving large-scale granular resource allocation problems efficiently with POP," in *SOSP*. ACM, 2021, pp. 521–537.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.

[25] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *KDD*. ACM, 2014, pp. 1456–1465.

[26] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu, "Hierarchical, parameter-free community discovery," in *ECML/PKDD (2)*, ser. Lecture Notes in Computer Science, vol. 5212. Springer, 2008, pp. 170–187.

[27] R. Mayer and H. Jacobsen, "Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints," in *SIGMOD Conference*. ACM, 2021, pp. 1289–1302.

[28] Z. Wei, X. He, X. Xiao, S. Wang, Y. Liu, X. Du, and J. Wen, "Prsim: Sublinear time simrank computation on large power-law graphs," in *SIGMOD Conference*. ACM, 2019, pp. 1042–1059.

[29] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Random graph models for the web graph," in *FOCS*. IEEE Computer Society, 2000, pp. 57–65.

[30] L. A.Adamic and B. A.Huberman, "The nature of markets in the world wide web," *Quarterly Journal of Electronic Commerce*, vol. 1, no. 1, pp. 5–12, 2000.

[31] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[32] "GUROBI: Gurobi optimizer reference manual," https://www.gurobi.com, 2024.

[33] "GOOGLE OR-TOOLS," https://developers.google.com/optimization, 2024.

[34] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím, "IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG," *Constraints An Int. J.*, vol. 23, no. 2, pp. 210–250, 2018.

[35] W. E. Hart, J. Watson, and D. L. Woodruff, "Pyomo: Modeling and solving mathematical programs in python," *Math. Program. Comput.*, vol. 3, no. 3, pp. 219–260, 2011.

[36] G. L. Nemhauser, M. W. P. Savelsbergh, and G. Sigismondi, "Minto, a mixed integer optimizer," *Oper. Res. Lett.*, vol. 15, no. 1, pp. 47–58, 1994.

[37] M. E. Lübbecke and J. Desrosiers, "Selected topics in column generation," *Oper. Res.*, vol. 53, no. 6, pp. 1007–1023, 2005.

[38] K. Lin, M. Ehrgott, and A. Raith, "Integrating column generation in a method to compute a discrete representation of the non-dominated set of multi-objective linear programmes," *4OR*, vol. 15, no. 4, pp. 331–357, 2017.

[39] H. Dyckhoff, "A new linear programming approach to the cutting stock problem," *Oper. Res.*, vol. 29, no. 6, pp. 1092–1104, 1981.

[40] J. Gondzio, P. González-Brevis, and P. A. Munari, "New developments in the primal-dual column generation technique," *Eur. J. Oper. Res.*, vol. 224, no. 1, pp. 41–51, 2013.

[41] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," *Oper. Res.*, vol. 46, no. 3, pp. 316–329, 1998.

[42] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *ICLR*. OpenReview.net, 2019.

[43] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, 2022.

[44] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR (Poster)*. OpenReview.net, 2017.

[45] J. B. Lee, R. A. Rossi, and X. Kong, "Graph classification using structural attention," in *KDD*. ACM, 2018, pp. 1666–1674.

[46] Y. Hu, C. de Laat, and Z. Zhao, "Optimizing service placement for microservice architecture in clouds," *Applied Sciences*, vol. 9, no. 21, p. 4663, 2019.

[47] P. Sanders and C. Schulz, "Think locally, Act globally: Highly balanced graph partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[48] S. Schlag, C. Schulz, D. Seemaier, and D. Strash, "Scalable edge partitioning," in *Proceedings of the 21th Workshop on Algorithm Engineering and Experimentation (ALENEX)*. SIAM, 2019, pp. 211–225.

[49] M. Kubat, "Neural networks: A comprehensive foundation by simon haykin, macmillan, 1994, ISBN 0-02-352781-7," *Knowl. Eng. Rev.*, vol. 13, no. 4, pp. 409–412, 1999.

[50] W. Khallouli and J. Huang, "Cluster resource scheduling in cloud computing: literature review and research challenges," *J. Supercomput.*, vol. 78, no. 5, pp. 6898–6943, 2022.

[51] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*. ACM, 2013, pp. 77–88.

[52] A. Rahimikhanghah, M. Tajkey, B. Rezazadeh, and A. M. Rahmani, "Resource scheduling methods in cloud and fog computing environments: A systematic literature review," *Clust. Comput.*, vol. 25, no. 2, pp. 911–945, 2022.

[53] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang, "When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone," in *SOSP*. ACM, 2021, pp. 621–637.

[54] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *SOSP*. ACM, 2013, pp. 69–84.

[55] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *OSDI*. USENIX Association, 2014, pp. 301–316.

[56] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda, "Protean: VM allocation service at scale," in *OSDI*. USENIX Association, 2020, pp. 845–861.

[57] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated slos for enterprise clusters," in *OSDI*. USENIX Association, 2016, pp. 117–134.

[58] 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 *OSDI*. USENIX Association, 2014, pp. 285–300.

[59] J. Y. Li, J. Zhang, W. Zhou, Y. Liu, S. Zhang, Z. Xue, D. Xu, H. Fan, F. Zhou, and F. Li, "Eigen: End-to-end resource optimization for large-scale databases on the cloud," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3795–3807, sep 2023. [Online]. Available: https://doi.org/10.14778/3611540.3611565

[60] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: algebraic scheduling of mixed workloads in heterogeneous clouds," in *SoCC*. ACM, 2012, p. 25.

[61] L. Suresh, J. Loff, F. Kalim, S. A. Jyothi, N. Narodytska, L. Ryzhyk, S. Gamage, B. Oki, P. Jain, and M. Gasch, "Building scalable and flexible cluster managers using declarative programming," in *OSDI*. USENIX Association, 2020, pp. 827–844.

[62] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *SoCC*. ACM, 2014, pp. 2:1–2:14.

[63] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *EuroSys*. ACM, 2016, pp. 35:1–35:16.

[64] A. C. König, Y. Shan, K. Newatia, L. Marshall, and V. Narasayya, "Solver-in-the-loop cluster resource management for database-as-a-service," *Proc. VLDB Endow.*, vol. 16, no. 13, pp. 4254–4267, 2023.

[65] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM*. IEEE, 2010, pp. 1154–1162.

[66] Z. Wu, Y. Deng, H. Feng, Y. Zhou, G. Min, and Z. Zhang, "Blender: A container placement strategy by leveraging zipf-like distribution within containerized data centers," *IEEE Transactions on Network and Service Management*, 2021.

[67] "KUBERNETES: Production-grade container orchestration," https://kubernetes.io, 2024.