Kairos: Building Cost-Efficient Machine Learning Inference Systems with Heterogeneous Cloud Resources

Baolin Li
Northeastern University

Siddharth Samsi
MIT

Vijay Gadepally
MIT

Devesh Tiwari
Northeastern University

ABSTRACT

Online inference is becoming a key service product for many businesses, deployed in cloud platforms to meet customer demands. Despite their revenue-generation capability, these services need to operate under tight Quality-of-Service (QoS) and cost budget constraints. This paper introduces Kairos [1], a novel runtime framework that maximizes the query throughput while meeting QoS target and a cost budget. Kairos designs and implements novel techniques to build a pool of heterogeneous compute hardware without online exploration overhead, and distribute inference queries optimally at runtime. Our evaluation using industry-grade machine learning (ML) models shows that Kairos yields up to 2× the throughput of an optimal homogeneous solution, and outperforms state-of-the-art schemes by up to 70%, despite advantageous implementations of the competing schemes to ignore their exploration overhead.

KEYWORDS

Machine Learning; Inference Systems; Heterogeneous Hardware.

1 INTRODUCTION

As machine learning (ML) models are becoming widely adopted in commercial services, the service providers will utilize cloud computing resources to serve their customers, and online inference has become a highly critical application for both on-premise and public cloud computing platforms [1–3]. As a result, an increasing amount of research effort is dedicated to improving the capability of cloud systems for inference workloads [4–9]. Serving ML inference is particularly challenging because they pose additional constraints and objectives beyond meeting latency deadlines. For example, business service providers can utilize the pay-as-you-go model to rent cloud computing instances, but they seek the following desirable objectives: (1) meet the quality-of-service target (QoS constraint, e.g., 99% of queries finish within 100ms); (2) efficient under a fixed cost budget; (3) process as many queries as possible per time unit (i.e., high query throughput).

Cloud platforms provide a wide range of virtual machines (VMs), and each comes with different hardware types (e.g., different CPU, GPU, and memory). While there have been previous attempts at providing partial solutions to exploit hardware heterogeneity in datacenter [10–12], edge [13, 14], and cloud [15–17], we lack a complete solution to achieve all the desirable properties (Sec. 2). In particular, prior schemes do not consider the full aspects of inference serving: *heterogeneous resource allocation and intelligent query distribution among allocated hardware resources.*

[1]Kairos has been accepted at the 32nd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)

Note that a heterogeneous pool of cloud compute instances (a mixture of GPUs and CPUs) appear naturally more promising for inference serving as they provide the opportunity to balance the trade-off between cost and performance (QoS target). More powerful and expensive instances can be used toward satisfying strict QoS targets for larger queries. Less powerful and relatively less expensive instances can be used for executing smaller queries that will not violate their QoS on such instances, and thereby, provide a chance to reduce the overall cost of the query serving system. Consequently, many prior techniques have opportunistically taken advantage of hardware heterogeneity to improve query throughput or meet QoS target [10, 11, 13, 14, 17]. However, none of them provide a systematic methodology to efficiently optimize the heterogeneous configuration (i.e., determine the number of GPUs and CPUs of different types). **Therefore, while prior works are heterogeneity-aware, they do not proactively optimize the hardware heterogeneity under a cost budget.**

In fact, we show that some heterogeneous configurations can perform significantly worse than an otherwise cost- and QoS-equivalent homogeneous configuration (Sec. 4). Determining a heterogeneous configuration requires online evaluation of multiple potential candidates. Unfortunately, this approach is not suitable when the query load changes or other system parameters change, since it requires invoking the exploration process frequently and potentially evaluating configurations that are worse than homogeneous configurations. This has been the main hindrance for the community to exploit heterogeneous computing hardware. **Kairos breaks this limitation and designs novel techniques to take full advantage of hardware heterogeneity while meeting QoS constraints under a given cost budget.**

Summary of Contributions. *We design and implement Kairos, a novel runtime framework to maximize throughput under cost budget and QoS constraints for machine learning inference tasks.* Kairos breaks away from searching the complex and vast configuration space of heterogeneous hardware. Instead, Kairos devises two techniques to quickly find a high-throughput heterogeneous configuration without exploring.

First, Kairos designs an efficient query-distribution mechanism to distribute queries among different cloud computing instances for any given heterogeneous configuration to maximize throughput – formulating this as a bipartite matching problem and solving it efficiently. Second, Kairos approximates the upper bound of the throughput that a heterogeneous configuration can provide at the best. Then, Kairos uses the similarity in top-ranked heterogeneous

Table 1: Overview of related works and KAIROS.

	Inference QoS	Throughput	Cost	Query Mapping	Proactive in Heterogeneity	No Online Exploration	Miscellaneous Notes
Paragon [10]	✘	✔	✘	✔	✘	✘	Requires prior data for training
TetriSched [11]	✘	✘	✘	✔	✘	✔	Supports user-based reservation
S^3DNN [13]	✔	✔	✘	✔	✘	✔	Uses supervised CUDA stream
DART [14]	✔	✔	✘	✔	✘	✘	Profiles layers and applies parallelism
Scrooge [15]	✔	✔	✔	✘	✘	✘	Chain execution of media applications
Ribbon [16]	✔	✔	✔	✘	✔	✘	Bayesian Optimization for allocation
DeepRecSys [17]	✔	✔	✘	✔	✘	✘	Schedules using profiled threshold
Clockwork [18]	✔	✔	✘	✔	✘	✔	Consolidates latency for predictability
KAIROS	✔	✔	✔	✔	✔	✔	Full heterogeneity support

configurations to pick the most promising heterogeneous configuration without online evaluation. Our evaluation confirms that KAIROS's configuration choice is often the near-optimal configuration across different machine learning models in production, where the optimal configuration is determined via exhaustive offline search of all heterogeneous configurations.

We have leveraged industry-grade deep learning models to drive the evaluation of KAIROS's effectiveness [17] – although we note that KAIROS's design is generic and not tuned for particular kinds of ML models. Our evaluation shows that compared to the optimal homogeneous configuration, KAIROS is able to significantly increase the throughput (by up to 2×) under the same QoS target and cost budget. KAIROS outperforms the state-of-the-art schemes in this area (Ribbon, DeepRecSys, and Clockwork [16–18]) by up to 70%, despite advantageous implementations of those competing schemes by ignoring the exploration overheads and improving the query distribution technique. Our proposed solution, KAIROS, is publicly available as an open-source package at *https://doi.org/10.5281/zenodo.7888058*.

2 RELATED WORK

Table 1 lists the relevant works in exploiting heterogeneous hardware and inference serving. Overall, KAIROS is the only work that satisfies all the desirable properties (table header from left to right): (i) meets QoS for inference queries; (ii) has service throughput requirement; (iii) is aware of heterogeneous hardware cost; (iv) intelligently distributes (or maps) queries among resources; (v) proactively allocates and optimizes heterogeneous resources; and (vi) does not need prior knowledge to train a model or perform online exploration. While some previous works are heterogeneity-aware (i.e., can efficiently use available heterogeneous hardware), they do not proactively configure the heterogeneity to optimize other aspects: query throughput, QoS, and cost budget.

Latency-critical applications are commonly studied in large-scale datacenter and cloud systems [19–23]. Previous works such as Paragon [10] and TetriSched [11] have focused on optimizing heterogeneous resource utilization [24–27], but their resource heterogeneity is pre-determined and sub-optimal, and their target applications are long-running jobs in datacenters, which is different from online inference tasks. Some other previous works have relied on tuning by expertise [28–31], prior profiling [32–35], or historical training data from similar applications [36–40], and cannot be used to solve the KAIROS problem.

Existing ML inference frameworks [1, 4, 8, 13, 14, 18, 41–47] are not suitable for exploiting heterogeneous hardware optimally and may require extensive profiling, KAIROS addresses this limitation. For example, S^3DNN and DART are heterogeneity-aware deep neural network (DNN) inference frameworks [13, 14], but their hardware heterogeneity is pre-determined. INFaaS [47] selects one particular hardware type from a pool of devices depending on the user application, but unlike KAIROS, it does not explore serving the model using different hardware simultaneously. Media application frameworks such as Llama [46] and Scrooge [15] allocate different hardware for different stages of the media application inference, but each query is assigned to the same sequence of hardware types, they do not distribute queries to heterogeneous resources like KAIROS and are not suitable for general purpose applications.

Ribbon [16] optimizes the serving cost by exploring different heterogeneous configurations, but compared to KAIROS, it still incurs Bayesian Optimization exploration overhead and does not exploit the heterogeneity by intelligently distributing the queries. DeepRecSys [17] explores heterogeneity between GPUs and CPUs when serving online queries. However, it does not explore the potential of different CPU/GPU ratios under a cost budget. It uses a hill-climbing algorithm to find an optimal threshold for query distribution, but it incurs tuning overhead as the threshold is different for each heterogeneous configuration. Clockwork [18] consolidates design choices in a top-down manner for deterministic inference latencies, but its central controller does not exploit heterogeneous hardware like KAIROS. Compared to all previous work, KAIROS delivers a full suite of heterogeneity support for cloud service and considers all key metrics (QoS, throughput, and cost).

3 BACKGROUND

Machine learning inference service. When machine learning models are trained into maturity, they will get deployed in production to provide ML inference service. The service users can submit inference requests through provided interfaces (e.g., HTTP request), then get a response. The inference pipeline can have multiple stages (e.g., data pre-processing, model prediction, post-processing), and they are typically packaged into a container image along with the software dependencies. On the cloud, the inference service provider can then allocate a set of compute instances and use a resource manager like Kubernetes to deploy the service. In this work, we focus on discussing the potential of using a heterogeneous resource instance allocation – how to efficiently distribute the inference queries and find a good heterogeneous configuration quickly.

Inference serving with QoS constraints and cost budget. The inference service has a QoS target, requiring the tail latency (e.g., 99^{th} percentile) of queries to be within a limit for a better user experience. For flexibility reasons and the pay-as-you-go model, businesses rent computing power from the cloud computing provider to meet the QoS target, but they also have a budget constraint. Each compute instance type, rented from the cloud, is associated with a price ($\$/hr$). Given a cost budget, one can only allocate a limited number of instances to serve as many queries as possible – that is, maximize the query throughput. The query throughput is defined as queries served per second (QPS). Since QoS cannot be violated, we use the **allowable throughput**, which is the maximum throughput the allocated instances can serve without causing QoS violation. In this work, we use *allowable throughput*, *throughput*, and *QPS* inter-changeably. All of them hold the implicit condition that QoS is satisfied.

4 MOTIVATION

In this section, we first provide experimental evidence to demonstrate that a heterogeneous configuration (a configuration can be a mixture of a few GPU instances, a few instances of CPU type A, and a few instances of type B) *can be* better than a homogeneous configuration *under the same cost budget* while respecting QoS. But, it is not always true – and any heterogeneous configuration is not superior by simply the virtue of heterogeneity.

First, we note that given a certain cost budget, one can choose to allocate the most cost-effective instances that can meet the QoS for all queries. We denote such instance type as **base instance**, and such strategy as **homogeneous serving** or homogeneous configuration. However, since inference queries have highly diverse batch sizes (or query sizes) [4, 16, 17], even though a cheaper but higher throughput-per-cost instance type cannot meet the QoS (so it cannot serve standalone as the allowable throughput is 0), it can still meet QoS for some smaller queries (queries with smaller batch sizes) due to the lower latency. Another choice is to replace some base instances with such cheaper instances (denoted as **auxiliary instances**), we denote this as **heterogeneous serving** or heterogeneous configuration. Unlike the base instance which comes from the optimal homogeneous instance, multiple types of auxiliary instances can be used for more flexibility and higher potential.

Are heterogeneous configurations always better? In Fig. 1, we compare the throughput of homogeneous serving against three different heterogeneous configurations on a Meta production model RM2 [2] under a fixed cost budget (dashed line). All configurations shown here respect the QoS target. We use three AWS EC2 instance types denoted as G1 for base instance, and C1, C2 for auxiliary instances (details in Sec. 7). The $(4, 0, 0)$ homogeneous configuration

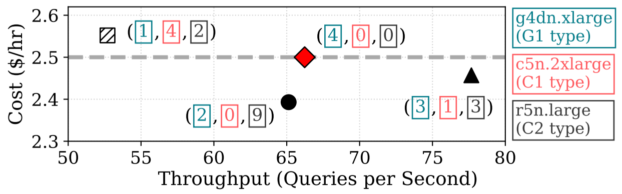

Figure 1: Different heterogeneous configurations versus the best homogeneous one. The number indicates the instance count of each type.

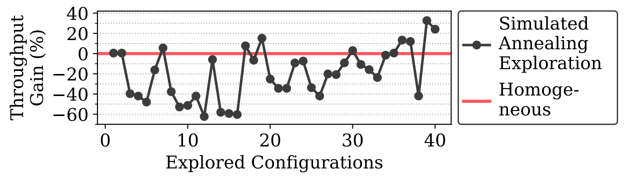

Figure 2: Throughput improvement over homogeneous when exploring using simulated annealing.

still has some unused budget for 70% of one G1, so we proportionally scale its throughput and cost up till the budget to give it an advantage. We observe that heterogeneous outperforms homogeneous as $(3, 1, 3)$ has 15% higher throughput than $(4, 0, 0)$. However, heterogeneity is not always necessarily better (e.g., $(2, 0, 9)$ and $(1, 4, 2)$). Especially for $(1, 4, 2)$, it indicates that simply raising the budget is not an ideal approach to gain throughput. *Therefore, being only heterogeneity aware is not sufficient (like previous work). But, how do we find an optimal configuration like* $(3, 1, 3)$*?*

Finding a high-performing heterogeneous configuration is expensive. This is because the search space of possible heterogeneous configurations is large, especially when there are more instance types, the space becomes high-dimensional and each instance type may have multiple instances. Second, evaluating the throughput of a new configuration is *expensive* and *time-consuming* because it requires service reconfiguration, just allocating new cloud instances would take significant time (tens of seconds). Also, during the online search of configurations, each explored configuration may not yield enough throughput to sustain all the queries - lower throughput than the homogeneous setting. Fig. 2 shows the limitation of heterogeneous serving during online exploration using simulated annealing [48]. Although we have pre-filtered out configurations that yield less than 20 QPS, the majority of explored configurations (about 70%) are still worse than the homogeneous serving marked as the red line. QoS violations will occur frequently if the allowable throughput is below the target level. *High cost of exploring and evaluating has prohibited previous works from finding a better heterogeneous configuration. Kairos breaks this limitation by providing an approximate method to quickly determine a promising configuration without any online evaluation.*

Exploiting heterogeneity via intelligent query distribution is the key to higher throughput. Next, we show that only finding a high-performing heterogeneous is not sufficient. Distributing diverse queries among heterogeneous instances is key to unlocking higher throughput. In previous results (Fig. 1 and 2), we used

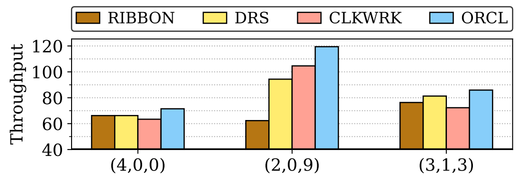

Figure 3: Heterogeneous configuration performance varies with query distribution mechanism.

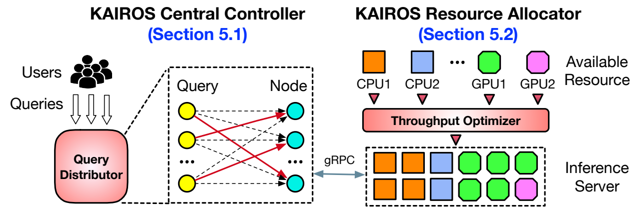

Figure 4: Kairos Overview.

Ribbon's [16] mechanism to schedule the arrived query on the best instance available. To demonstrate the impact of query distribution strategies, we employ two more complicated query distribution schemes denoted as DRS [17], CLKWRK [18], and an oracle scheme (ORCL) (details in Sec. 7) in Fig. 3. We make two observations. First, all state-of-the-art schemes Ribbon [16], DRS [17] and CLKWRK [16] are sub-optimal. Second, one is not necessarily better than the other and leaves scope for more improvement. *Kairos exploits this opportunity and bridges the gap to the oracle scheme by designing a new intelligent query distribution mechanism for heterogeneous serving.*

5 KAIROS DESIGN

In this section, we provide design details of Kairos as a complete heterogeneous serving solution, illustrated in Fig. 4. The queries submitted by users will be distributed to the processing nodes, which consist of heterogeneous compute instances built by a throughput optimizer. The first design target of Kairos is to efficiently distribute all the arrived queries of an ML inference service to different instances in a particular heterogeneous configuration (Sec. 5.1). Since finding a promising heterogeneous configuration within a cost budget is also a challenging problem by itself, this is Kairos's second design component (Sec. 5.2).

5.1 Kairos Query Distribution Mechanism

Overview. We introduce the query distribution mechanism as the first design component of Kairos. The key objective is to intelligently distribute queries of different batch sizes to different instances so that the throughput is maximized. In Sec. 8, we confirm that Kairos's query-distribution mechanism is indeed key to its overall effectiveness and works across different heterogeneous configurations.

We start with mathematical formulation to maximize the throughput for a given configuration. The key intuition is to distribute the queries in a way that maximizes the available time in all instances in the future. This maximizes the likelihood of serving more queries in the future – a higher throughput. We show that this problem can be transformed and mapped to a min-cost bipartite matching problem, which Kairos solves to find an efficient query-distribution plan without knowledge of future query arrivals.

Table 2: Query distribution optimizer parameters.

List	Description
$L_{i,j}$	Time needed to finish serving Q_i on instance I_j from t_0.
m	Number of queries at time t_0.
n	Number of instances in the configuration.
C_j	Heterogeneity coefficient for instance I_j.
T_{qos}	QoS target latency.
W_i	Query Q_i's time spent waiting in queue before t_0.
$P_{i,j}$	Query-to-instance pairing/assignment matrix.

Mathematical formulation of query distribution for throughput maximization. Our problem objective is: given a number of queries to be served at the current time t_0, maximize future availability of instances until a future time instance. This is equivalent to minimizing the total resource usage since unused resources can be used to process future queries, indirectly maximizing throughput at the current time.

Suppose at t_0, there are m queries in the serving queue, denoted as $Q_1, Q_2, ..., Q_m$, and n compute instances in the heterogeneous configuration, denoted as $I_1, I_2, ..., I_n$. Table 2 summarizes the parameters used in our mathematical formulation. If distribute query Q_i to instance I_j, the query completion time from t_0 ($L_{i,j}$) includes the serving latency (varies for different i, j) and if there is a query currently being served at I_j, the remaining time till I_j can serve Q_i. For all queries and instances, this time can be represented by an $m \times n$ matrix L. Namely, $L_{i,j}$ represents the I_j instance resource usage (measured by time) if scheduled to serve query i.

It is important to note that the equal wall-clock usage time on different instance types in a heterogeneous configuration do not hold the same value. That is, one second of GPU is not equivalent to one second of CPU. To account for this, Kairos employs a heterogeneity coefficient C_j for each instance type j.

Definition 1. *We define $C_j \in (0, 1]$ as the heterogeneity coefficient for instance j. It is used to represent the relative importance of instance j compared to other instances in a heterogeneous system. It is calculated as the ratio of the largest query latency between I_j and the base instance type.*

The heterogeneity coefficient helps Kairos weight resources differently, which aligns with previous task scheduling algorithms for heterogeneous processors [49, 50]. To determine C_j, we first set the coefficient of the base instance type (e.g., lowest latency instance) to 1 as a normalization point, then calculate C_j as the latency ratio. We find that using the largest query the system can serve to measure the latency ratio works well. For example, if the largest query has latency 100ms on instance I_1, 200ms on I_2 and 500ms on I_3, then $C_1 = 1$, $C_2 = 0.5$, $C_3 = 0.2$. In our system, we limit the maximum batch size of a query to 1000 because of QoS constraints. Intuitively, larger queries are more compute demanding and more prone to violate QoS, thus, they are more suitable for evaluating the relative importance of instances in a heterogeneous system. With the introduction of heterogeneity coefficient, the **revised usage time** for instance j can be expressed as $C_j L_{i,j}$.

This usage can be calculated for every query/instance pair. Since the time is relative to the base instance, we can directly sum the usage up across all instances, and the sum is the aggregated resource usage. To minimize this, we need to carefully select which Q_i to be served on which I_j. We define these optimization variables as an $m \times n$ binary matrix P:

$$P_{i,j} = \begin{cases} 1 & \text{if query } Q_i \text{ is served by instance } I_j, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Then, we express the minimization objective function as:

$$f(P) = \sum_{i=1}^{m} \sum_{j=1}^{n} C_j L_{i,j} P_{i,j} \tag{2}$$

Before optimizing this objective function, we first make an observation that this formulation maps closely itself to the linear-sum assignment problem, or in graph theory, a min-cost bipartite matching problem [51]. Therefore, we leverage the theory of bipartite matching to formulate and solve this as a bipartite matching optimization.

A bipartite graph has two disjoint and independent sets of vertices U and V. In our case, U contains the queries as vertices, V contains the instances as vertices. An edge is available for all query-instance pairs, and a cost is associated with each edge. A typical min-cost bipartite matching would have the same number of elements in U and V, the elements are one-to-one matched with the total cost minimized. However, in our situation, there is no guarantee about the number of queries. If there are fewer queries than instances, the matching is valid when all queries are matched to a unique instance, and when there are fewer instances than queries, the matching is valid when all instances are matched to a unique query. The cost of each edge between Q_i and I_j corresponds to $C_j L_{i,j}$ in Eq. 2.

Before solving this bipartite matching problem to maximize the throughput, we note that processed queries can only count towards throughput when served under QoS, otherwise KAIROS's idea of heterogeneous serving becomes meaningless: one can simply find the instance type with the highest throughput-to-cost ratio and do homogeneous serving. To be QoS-aware, we add an inequality constraint:

$$(L_{i,j} + W_i) P_{i,j} \le T_{qos} \tag{3}$$

This constraint states that if serving Q_i with I_j, the sum of query completion time on I_j and queue wait time before t_0 should be less than the QoS target. We need to consider the query wait time W_i because not all queries are guaranteed to be scheduled to an instance (e.g., more queries than instances), they may need to wait in a queue until more resources become available and restart another round of query distribution. Considering W_i in Eq. 3 avoids starvation of unscheduled queries when new queries continuously arrive.

In summary, KAIROS formulates an optimization problem and designs its objective function and constraints for throughput maximization under the QoS target. The key design principle is respecting the fact that different queries have different speedups from one instance type to another (e.g., queries with larger batch sizes have higher speedups from CPU to GPU). By prioritizing higher speedup queries on more powerful instances, KAIROS minimizes resource usage and prepares maximized slack time for future queries. This is reflected in Eq. 2, where the L matrix implicitly contains this information. Fig. 5 visualizes this effect with a 2-instance example. By efficiently distributing current queries without future information,

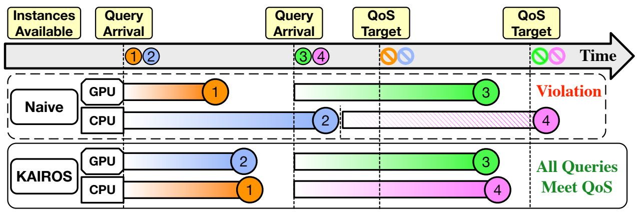

Figure 5: Higher-speedup queries should be prioritized on more powerful instances to create more slack time. Queries 1 to 4 (arrived in order) are represented in different colors.

KAIROS leaves more slack time for the future, thus it can process all 4 queries while a naive scheme (e.g., FCFS) can only process 3 queries (shaded query does not count towards throughput due to QoS violation). The superior distribution scheme gives KAIROS 33% higher throughput (4 queries vs. 3 queries processed in time) than the naive scheme, despite the same hardware.

Putting everything together, the query-distribution problem can be formulated as follows:

$$\min_{P} \sum_{i=1}^{m} \sum_{j=1}^{n} C_j (L_{i,j}) P_{i,j} \tag{4}$$

$$\text{s.t. } \forall i, j, \quad (L_{i,j} + W_i) P_{i,j} \le T_{qos}, \tag{5}$$

$$\forall i, j, \quad \sum_{i=1}^{m} P_{i,j} \le 1, \sum_{j=1}^{n} P_{i,j} \le 1, \tag{6}$$

$$\sum_{i=1}^{m} \sum_{j=1}^{n} P_{i,j} \ge min\{m, n\} \tag{7}$$

where $i \in \{1, 2, ..., m\}$ and $j \in \{1, 2, ..., n\}$. Eq. 6 indicates one-one mapping, and Eq. 7 guarantees when there are more instances than queries, every query gets mapped to an instance; when there are more queries than instances, every instance receives a query.

We note that KAIROS's formulated problem is not a strict min-cost bipartite matching problem as discussed in traditional bipartite matching literature because of the QoS constraint in Eq. 5. Therefore, to guarantee feasibility, KAIROS integrates this constraint into the objective function by modifying the L matrix with a condition. If serving Q_i on I_j does not violate the QoS, $L_{i,j}$ is unchanged. If it violates QoS, then $L_{i,j}$ is penalized by a large quantity (e.g., 10× of the QoS target). Consequently, KAIROS achieves min-cost solutions that avoid QoS-violating Q_i-I_j matching. With this constraint integration, the new L matrix becomes:

$$L_{i,j} = \begin{cases} L_{i,j} & \text{if Eq. 5 is true,} \\ 10 \cdot T_{qos} & \text{otherwise.} \end{cases} \tag{8}$$

Then, the Eq. 5 constraint is removed, and the problem with updated parameter L becomes a strict min-cost bipartite matching problem. KAIROS solves this problem using the Jonker-Volgenant algorithm [52] which is a variant of the widely used Hungarian algorithm [53], but more efficient in practice [54].

Remarks on assumptions and overhead. We note that KAIROS requires constructing the parameter matrix L, which requires predicting the query latency of certain batch sizes on different instance types. Fortunately, ML inference is a fully deterministic process without conditional branching, thus the latency is highly predictable [18]. Because the query includes a batch of requests, KAIROS makes sure an instance serves one query at a time without any resource contention. Thus, the end-to-end latency has a very low variance ($< 0.5\%$ of mean). Previous work has observed that inference latency can be accurately predicted with simple features such as request batch size [55]. We have observed similar trends in our experiments as inference latency is highly correlated with query batch size: the Pearson correlation coefficient [56] between latency and batch size is greater than 0.99 for all models and instance types in Sec. 7. As a result, KAIROS can model this completely online with a handful of queries without requiring any prior knowledge or instrumentation. KAIROS starts with a linear model but does not rely on the model accuracy because it will quickly transition into a lookup table after processing more queries. All our evaluation results include this overhead from learning the query latencies online. In practice, as a noise safeguard, we replace T_{qos} with ξT_{qos} in Eq. 5, and setting ξ to be 0.98 such that the completion time predicted to be within 2% range of the QoS target is considered a violation.

5.2 KAIROS Throughput Estimation

Next, we discuss how KAIROS quickly reaches a good configuration from the vast search space with a cost budget constraint as the second part of its design component. Calculating the cost is easy but evaluating the throughput of a configuration is expensive and causes delays in finding a good configuration (Sec. 4), prohibiting the system from promptly responding to load changes. KAIROS takes a different approach to approximate the actual throughput using an upper bound. Classical approximation algorithms for unrelated machines [57, 58] are not applicable to our application scenario of serving online inference queries using KAIROS's query distribution mechanism. We have also explored other options such as queuing theory [59, 60] to analytically calculate the actual throughput. However, due to the dynamic service time (varying batch size), the heterogeneity in hardware, and unconventional queue discipline (Sec. 5.1), we cannot fit the problem into a classical $M/M/c$ queue framework. Therefore, we take KAIROS's approximation approach to avoid expensive evaluations.

Designing an application-specific approximation strategy is challenging [61]. This is also true for KAIROS's throughput approximation due to QoS restrictions and the complex interactions between queries and heterogeneous instances. KAIROS tackles this challenge with a method to calculate a throughput upper bound for a given configuration. With this method, KAIROS first finds promising candidates with high upper bounds from search space under cost budget without any evaluation, then performs aggregation to output a final configuration. To explain, we first demonstrate the intuition and calculation for the throughput upper bound given a simple heterogeneous configuration – that is, one instance of base type (e.g., GPU) and one instance of auxiliary type (e.g., CPU). Then, we show how to extend this method to cases where each instance type can have multiple instances. Eventually, we extend it to cases where one can have multiple different types of auxiliary instances.

DEFINITION 2. *For an inference service, given a particular allocation of hardware resources, the throughput QPS varies with the query distribution algorithm λ. The allowable throughput can be represented*

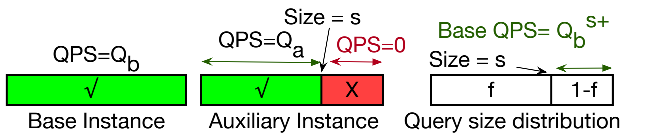

Figure 6: Upper bound calculation parameters. The auxiliary instance cannot serve larger queries due to QoS.

as a function $Q(\lambda)$. We define the throughput upper bound QPS_{max} as a number that satisfies $\forall\lambda$, $Q(\lambda) < QPS_{max}$.

Essentially, the upper bound of a particular hardware allocation is a throughput that cannot be exceeded no matter how the system distributes the query. A throughput number may be the upper bound for a hardware allocation, but if the allocation changes, it may not be the upper bound anymore.

We estimate the throughput upper bound as follows: KAIROS makes a simple observation that upper bound estimation is akin to estimating the maximum possible throughput in an unrealistic scenario where all queries are available to us at the beginning, and we can control when each query should arrive – then there is no need to worry about latency interactions with queuing. Recall that KAIROS's query distribution mechanism efficiently accounts for the practical case when we have no control over when queries will arrive: queries may wait in the queue, and instances may be idle waiting for queries. Compared to the practical case, our upper bound calculation ensures queries do not miss QoS by waiting in the queue and instances do not waste idle cycles that could have served more queries.

Using the one-base-one-auxiliary simplification example, the intuition is to determine which instance type is the bottleneck given a mixture of queries of various batch sizes, and then, the bottleneck instance type dictates the maximum possible throughput we can achieve. Formally, let Q_b and Q_a denote the *standalone* throughput (QPS) achieved by the base and auxiliary instance type, respectively. Note that the standalone auxiliary instance cannot satisfy QoS for all queries. Therefore, Q_a refers to the throughput achieved when serving only queries that do not violate QoS (i.e., queries smaller than size, say s). Queries larger than size s will then have to be served by the base instance. However, the base instance throughput when serving larger-than-s-size queries will be lower than Q_b because larger batches require longer processing time – we use s+ to represent queries larger than size s (that cannot be consumed by auxiliary instance) and denote their allowable throughput running on base instances by Q_b^{s+}.

We can then partition the query mix into two fractions: f for queries smaller than batch size s, and $1 - f$ for queries larger than s. If the auxiliary instance is fully occupied with smaller size queries, it implies that $\frac{1-f}{f} \times Q_a$ of the throughput needs to be executed on the base instance type, represented by the ratio between s+ queries and the Q_a queries times the Q_a throughput. Sending larger queries to base and small queries to auxiliary instances also aligns well with the design principle of Sec. 5.1. However, recall that all the queries offloaded to base instance are of sizes larger than s, and hence, the base instance can serve them only at the rate of Q_b^{s+}. Fig. 6 shows a visual representation of the mathematical formulation above.

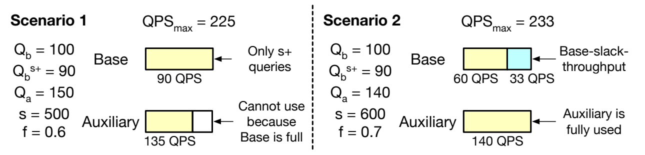

Figure 7: Example of how Kairos's upper bound calculation works. Scenario 1 represents when the base instance is the bottleneck and Scenario 2 represents when the auxiliary instance is the bottleneck.

Note that there are only two potential outcomes that dictate the maximum throughput we can achieve: (1) the base instance is the bottleneck, and (2) the auxiliary instance is the bottleneck.

If $Q_b^{s+} \leq \frac{1-f}{f} \times Q_a$, that means *the base instance is the bottleneck*: the queries offloaded from auxiliary instance cannot be fully consumed by the base instance, thus the auxiliary instance cannot serve queries at the rate Q_a. If the maximum throughput is denoted as QPS_{max}, then, a $1-f$ fraction of all the queries (or queries larger than size s) will be served by the base instance at the rate of Q_b^{s+}. Hence, the maximum throughput can be estimated as:

$$QPS_{max} = \frac{Q_b^{s+}}{1-f} \tag{9}$$

If $Q_b^{s+} > \frac{1-f}{f} \times Q_a$, that means *the auxiliary instance is the bottleneck and the base instance has some slack left to serve more queries.* If the maximum throughput is denoted as QPS_{max}, then, f fraction of these queries (queries smaller than size s) will be served by the auxiliary instance at the rate of Q_a. So, the QPS_{max} is equal to $\frac{Q_a}{f}$. However, recall that the base instance still has some slack available to serve more queries. Intuitively, the slack ratio at the base instance is equal to the difference between Q_b^{s+} and $\frac{1-f}{f} \times Q_a$, divided by its total capability Q_b^{s+}. Multiplying this slack ratio by the base throughput, we get the base slack throughput (extra queries the base can serve when the auxiliary is the bottleneck) as:

$$\text{Base-slack-throughput} = \left(\frac{Q_b^{s+} - \frac{1-f}{f} \times Q_a}{Q_b^{s+}}\right) \times Q_b \tag{10}$$

Therefore, the maximum possible throughput is:

$$QPS_{max} = \frac{Q_a}{f} + \left(\frac{Q_b^{s+} - \frac{1-f}{f} \times Q_a}{Q_b^{s+}}\right) \times Q_b \tag{11}$$

Fig. 7 uses two example scenarios 1 and 2 to demonstrate the upper bound calculations. In Scenario 1, the auxiliary instance can only process queries smaller than size 500, and the base has to spend all its capacity processing the Q_b^{s+} queries, Kairos uses Eq. 9 to calculate QPS_{max}. In Scenario 2, the base instance still has some slack capacity left after processing the Q_b^{s+} queries, and Kairos uses Eq. 11 to calculate QPS_{max}.

Kairos's approach of query mixture partition by batch size s would be over-optimistic if the partitions have strong temporal locality (e.g., if all large queries arrive together before small queries, then auxiliary instance cannot contribute). However, no such cyclic behavior has been observed, and due to the law of large numbers, this upper bound is reasonable over the long term for a large number of query mixes. We can next extend this method to the case when each instance type has multiple nodes. If the base instance has u nodes and the auxiliary instance has v nodes, then, Eq. 9 and 11 can be written as:

$$QPS_{max} = \frac{uQ_b^{s+}}{1-f} \tag{12}$$

$$QPS_{max} = \frac{vQ_a}{f} + \left(\frac{uQ_b^{s+} - \frac{1-f}{f} \times vQ_a}{uQ_b^{s+}}\right) \times uQ_b \tag{13}$$

The next step is to extend this upper bound estimation for multiple types of auxiliary instances. Multiple types of auxiliary instances are more challenging since each new auxiliary instance has its own QoS-respecting region and throughput in that region. Fortunately, Kairos is not concerned with modeling the accurate throughput, instead, it only cares about the upper bound estimation of the throughput. Therefore, it makes a relatively simple approximation that additional auxiliary instance types have the same QoS-respecting region as the type with maximum s size and f fraction. This essentially makes the upper bound estimation more optimistic since some weaker auxiliary instances are assumed to meet QoS for batch sizes larger than their limit. As our evaluation confirms (Sec. 8.5), even though this approximation results in a higher upper bound, configurations still follow similar order as the actual throughput (a higher upper bound is likely to indicate higher throughput). With this simplification, the n-auxiliary-instance-type general case upper bound can be written. First, we define an intermediate variable:

$$C = \frac{\sum_{i=1}^{n} v^i Q_a^i (1-f')}{f'} \tag{14}$$

where $f' = max(f^1, f^2, ..., f^n)$. This corresponds to $\frac{1-f}{f} \times Q_a$ that is used to compare with Q_b^{s+}. The superscript i indicates auxiliary instance type i. Q_a^i is the instance type i throughput for queries with batch size smaller than the maximum s of all types. We have the final upper bound formula as:

$$QPS_{max} = \begin{cases} \frac{uQ_b^{s+}}{1-f'} & \text{if } uQ_b^{s+} \leq C, \\ \frac{\sum_{i=1}^{n} v^i Q_a^i}{f'} + \left(\frac{uQ_b^{s+} - C}{uQ_b^{s+}}\right) uQ_b & \text{otherwise.} \end{cases} \tag{15}$$

Finally, now that we have the formula, we can calculate the upper bound for all configurations within the cost budget. We describe the last step that Kairos performs to reach the final configuration without evaluations. To this end, Kairos can trivially pick the configuration with the highest upper bound from its approximation method. However, Kairos recognizes that a higher upper bound does not necessarily mean a higher throughput. To address this, Kairos applies a similarity-based method to pick a configuration from the highest upper bound configurations. Kairos first checks if the top-3 upper bound configurations have the same base instance number. If true, Kairos picks the highest upper bound configuration. Otherwise, for each configuration with a top-10 upper bound, Kairos calculates its squared Euclidean distance to the other 9 configurations, sums them up, and picks the one with the least distance sum. Such metric is commonly used in clustering analysis [62], it is equivalent to considering all configurations to form a cluster, then setting the cluster centroid as the configuration that has the least sum-of-squared error (SSE). The intuition is that there should be a region for the high throughput configurations, and the distance-based method lands Kairos in such a region. We find Euclidean distance to be a reasonable similarity metric, other metrics such as cosine similarity do not reflect the locality of the promising region. As our evaluation confirms (Sec. 8), Kairos is able to find a good configuration for all workloads, and the process does not evaluate any configuration online.

Remarks on assumptions and overhead. Kairos's upper bound based throughput estimation has a one-time warmup phase consisting of two major steps. Firstly, it needs to compute the upper bound for all combinations of instance numbers of each type under cost budget. Fortunately, this calculation is quick using the formula in Eq. 15: for an order of 1000-configuration search space, all upper bounds can be calculated and ranked within 2 seconds, negligible compared to even one evaluation (tens of seconds for instance allocation). Secondly, Kairos's estimation implicitly assumes that it can obtain information on the batch size distribution (fraction f of batch sizes smaller than s). This is done via query monitoring to keep track of a number of most recent queries (e.g., 10000 queries), and does not require extra profiling. In addition, to provide robustness, we demonstrate that Kairos adapts when the batch size distribution changes and continues to be effective (Sec. 8.4).

Upper-bound-assisted search algorithm. We also develop Kairos+, a variation of Kairos that uses a minimum number of online evaluations to quickly find the optimal (Algorithm 1). The search process is guided by the estimated upper bounds and is shown to be outperforming any other traditional search space exploration (Sec. 8). The intuition to greedily start from high upper bound configurations as these configurations have better potential than others, and after evaluating a number of such instances, the current best throughput will likely be high enough to filter out a large number of configurations whose upper bounds are lower. Another pruning mechanism in the algorithm is sub-configuration pruning. If configuration $\boldsymbol{x}_1$ can add more instances to become $\boldsymbol{x}_2$, we define $\boldsymbol{x}_1$ to be a sub-configuration of $\boldsymbol{x}_2$. Every time a configuration is evaluated, all of its sub-configurations get pruned away from the search space since these sub-configurations will not have higher throughput than the evaluated one. We note that upper bounds that are tight to the actual throughput are especially beneficial to Kairos+ since more configurations can be pruned away.

Algorithm 1: Kairos+'s pruning-based algorithm for quickly finding optimal configuration.

```
$UBs \leftarrow$ Sort all $QPS_{max}$ high to low
$curr\_best = 0$ // Highest throughput so far
$best\_config = None$
$configs \leftarrow$ list of all configs within cost budget
$\boldsymbol{x} \leftarrow$ variable representing one configuration
foreach $UB(\boldsymbol{x})$ in $UBs$ do
    if $\boldsymbol{x} \in configs$ then
        $eval = f(\boldsymbol{x})$ // Actual QPS evaluation.
        if $eval > curr\_best$ then
            $curr\_best = eval$
            $best\_config = \boldsymbol{x}$
            Filter all $\boldsymbol{c}$ out of $configs$ that satisfies
              $UB(\boldsymbol{c}) \leq curr\_best$
        end
        Prune away all sub-configs. of $\boldsymbol{x}$ from $configs$
    end
end
return $curr\_best$, $best\_config$
```

6 IMPLEMENTATION

Kairos and Kairos+ are implemented as a cloud inference server, similar to frameworks such as NVIDIA Triton [63]. Every allocated compute instance hosts a copy of the model, only one query consisting of the batched requests is served by one model copy at a time. If the model is hosted on a CPU instance, all the CPU cores will be used. The query distribution mechanism (Sec. 5.1) resides in a central controller, which performs the optimization to decide the query-instance mapping. It acts as a client and sends the optimized inference requests to individual instances (as servers) through the gRPC protocol [64]. Compared to a traditional load balancer, the central controller is aware of the server heterogeneity and tries to avoid QoS violations. When queries arrive, the controller estimates the query latency and uses its estimated server remaining time to construct the $\boldsymbol{L}$ matrix for Eq. 4 and 5.

Notice that the central controller needs to solve the optimization problem in real time because the solving time is added to the inference latency. We implement this using the `scipy.optimize` package to solve the bipartite matching problem in polynomial time. We experimentally confirmed that the sum of network delay and algorithm runtime for a large 20-query-20-instance matching is within 0.05ms, and even for hundreds of queries arriving concurrently the overhead is well within 1ms, negligible compared to QoS which is typically tens to hundreds of milliseconds. Thereby, Kairos ensures that its controller does not become the bottleneck or add significant latency. Furthermore, according to the theory from POP [65], inference service frameworks like Kairos can scale to extremely large systems by dividing the system into multiple sub-systems and running a Kairos instance on each sub-system.

7 METHODOLOGY

Models and QoS constraints. We use industry-scale machine learning service models to drive the evaluation of Kairos' effectiveness. Such ML models are widely used in online services and have several advantages: (1) wide interest from systems research [16, 66, 67] (2) large customer demand and wide deployment in industry [68, 69] (3) representative public trace and artifacts for reproducibility and comparison [2, 17, 70]. Table 3 lists the models and QoS as 99th tail latency target based on their service specifications.

These models are chosen because they represent a wide range of ML-based applications, the internal architectures are also highly diverse across models [2, 70]. For example, NCF is a light-weighted model with limited embedding tables, but the RM2 model is dominated by large embedding tables, while the MT-WND model has

Table 3: Models and QoS targets.

Model	Description	Application	QoS
NCF [71]	Collaborative Filtering	Movie recommendation	5 ms
RM2 [2]	Meta's recommendation model class 2	High-accuracy social media posts ranking	350 ms
WND [72]	Google Wide and Deep recommender system	Google App Store	25 ms
MT-WND [73]	Multi-Task Wide and Deep, predicts multiple metrics in parallel	YouTube video recommendation	25 ms
DIEN [74]	Alibaba Deep Interest Evolution Network	E-commerce	35 ms

Table 4: Different instance types used in heterogeneous pool.

Instance Type	Instance Class	Price ($/hr)
g4dn.xlarge	GPU Accelerated Computing	0.526
c5n.2xlarge	Compute Optimized CPU	0.432
r5n.large	Memory Optimized CPU	0.149
t3.xlarge	General Purpose CPU	0.1664

large parallel DNN predictors for abstract features. The QoS constraints for these models cover a wide range and are selected strictly based on real applications [17, 72, 74]. For example, Alibaba's e-commerce service requires tens of milliseconds of response time while Meta's social media platform requires hundreds of milliseconds.

An important feature of model queries is that they arrive in different batch sizes (Sec. 4). Our evaluation is driven by the production trace of real query batches from Meta [17]. The query inter-arrival is generated from a Poisson process generating 100s of queries per second, which has been commonly used in various online inference serving studies [2, 75–77]. To evaluate Kairos's response to load change and sensitivity, we also use Gaussian distributed batch sizes because Gaussian distribution is another commonly used distribution for online services [78].

Computing hardware types and cost. We use different hardware types provided by Amazon Elastic Compute Cloud (EC2) for our evaluation. The compute instances are categorized into four classes: general purpose, compute optimized, memory optimized, and accelerated, and represent different cost points. We select an instance type of each class to form the heterogeneous pool, Table 4 summarizes the compute instance types. These instance types are selected to collectively cover a wide spectrum of performance and cost points, as they come from different representative compute-memory-accelerator classes.

These instance types create a 4-dimensional search space, each heterogeneous pool corresponds to a certain number of instances of each type. We select the instance size (indicated by `.xlarge` characters) so that all types have 16GB of memory allocation to host the model. We use the `g4dn.xlarge` GPU instance type as the base instance type as only this instance type can meet QoS for all batch sizes. The other instance types (e.g., c5n.2xlarge) are considered auxiliary instance types. `g4dn.xlarge` instance type uses one NVIDIA T4 GPU. Compared to all other GPU-accelerated instance types in EC2, `g4dn` has the best inference performance – similar to `p3.2xlarge` instance type (NVIDIA V100 GPU). But, `p3` has nearly 6× higher cost than `g4dn`, therefore, we use `g4dn` as the most cost-effective base instance type to be used in a homogeneous pool. The cost budget is set to 2.5$/*hr* by default for Kairos evaluation - this budget is chosen purely to ensure the cost incurred during the evaluation period remains within a reasonable limit while demonstrating the value of the idea. As our evaluation confirms, Kairos is not sensitive to the cost budget (e.g., 10$/*hr* budget) and is not tuned to work specifically only for certain cost budget caps. Kairos respects the budget cap as a constraint – consistent with our design goal.

Metrics. Our main evaluation metric remains the throughput (QPS) under QoS (Sec. 4). To find this allowable throughput, we gradually increase the arrival rate of queries, until the QoS is violated. In addition, since evaluating a configuration in the search space is expensive, we also compare the number of iterations required to find the optimal configuration for a particular scheme. This part is only applicable to competing techniques (non-Kairos solutions) since Kairos does not require online exploration and evaluation.

Competing query distribution techniques. We evaluate Kairos against three recent ML inference serving schemes.

Ribbon. This scheme focuses only on hardware allocation, therefore it has a simple first-come-first-serve (FCFS) query distribution policy which prefers instances of the base type when multiple instances are available. However, it uses Bayesian optimization to allocate a near-optimal set of heterogeneous hardware on the cloud, which is similar to the problem Kairos tries to solve in Sec. 5.2.

DRS. This scheme represents the scheme used in a related work – DeepRecSys system [17]. It uses a static batch size threshold to decide whether to serve a query on the base instance (if large than a threshold) or on the auxiliary instance (otherwise). To determine the threshold, a hill-climbing sweep is used by DeepRecSys system [17] to find the threshold that yields the highest throughput.

CLKWRK. This is a QoS-aware query distribution scheme inspired by Clockwork [18]. It has a focus on latency consolidation, but we have used its central controller for comparison with Kairos. This scheme monitors all hardware availability timings and accurately predicts query latency. A query is guaranteed to be served within its latency target unless none of the instances can meet the QoS target. Each instance maintains an individual FCFS query queue. There is a central controller to keep track of all hardware and queue timings and send new queries to the instance queues.

ORCL. Oracle scheme is a practically infeasible scheme only for reference to understand the limits of performance. Oracle always has higher throughput than other schemes because it knows the future query arrival patterns. It creates a sequence of queries according to batch size distribution and sorts all queries by the batch size. Whenever a base instance is available, it serves the next largest query. For auxiliary instances, it is the next smallest query. There is no wait time for queries, and queries never run on instances that violate QoS. The throughput on serving the query sequence is recorded, and among all possible heterogeneous configurations, the largest throughput is used as the Oracle throughput.

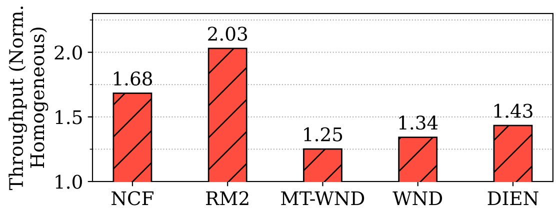

Figure 8: Kairos yields higher throughput compared to the homogeneous configuration.

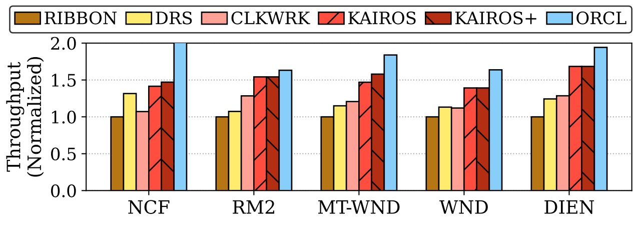

Figure 9: Throughput comparison of Kairos and Kairos+ against other competing schemes.

Recall that **Kairos** determines the near-optimal heterogeneous configuration using its upper-bound-assisted method without any online evaluation - unlike other competing schemes. **Kairos+** is an online variation of Kairos for optimal configuration, but the number of online evaluations is limited because it is guided by Kairos's upper bound method, as our evaluation discusses next.

8 KAIROS: EXPERIMENTAL EVALUATION

First, we quantitatively evaluate Kairos's performance compared to other competing schemes and state-of-the-art approaches. We explain why Kairos works effectively, and what the key contributing factors toward its superiority over existing methods are. Finally, we demonstrate that Kairos performs effectively even under varying constraints and parameters (e.g., cost budget, QoS, and batch size distribution).

8.1 Comparison with Best Homogeneous

First, our experimental results (Fig. 8) confirm that Kairos provides significant allowable throughput improvement compared to the most competitive base instance homogeneous configuration under the same QoS constraints and budget (Sec. 7). The number of instances in the homogeneous pool is determined by the maximum number of nodes that can fit within the cost budget, this is the optimal homogeneous resource configuration. Note that the cost budget is not a multiple of `g4dn.xlarge` price. To compensate for the wasted budget, we scale up the homogeneous throughput proportionally. *But, to evaluate conservatively, we allow the budget slack of Kairos to be wasted. Despite this, Kairos is still able to provide up to* $2\times$ *throughput (i.e., RM2 model) and more than* $1.25\times$ *in all cases.*

8.2 Comparison with State-of-the-Art

Next, our results show that Kairos outperforms competing schemes of Ribbon, DRS, and CLKWRK in Fig. 9. Recall that the competing techniques DRS and CLKWRK only focus on distributing the queries to a set of instances following a certain strategy. They do not observe that one can build a heterogeneous configuration that is more performant than a homogeneous configuration under a cost budget. To provide these techniques an advantage, we provide each scheme with the best heterogeneous configuration obtained via Oracle search and compare the throughput against Kairos. Fig. 9 shows the allowable throughput for different schemes.

We make several observations. First, as expected, both DRS and CLKWRK outperform the Ribbon scheme because Ribbon uses a simple query distribution mechanism. Second, Kairos significantly outperforms all existing techniques and is close to the Oracle scheme. For all models, Kairos can provide around $1.5\times$ the throughput of Ribbon. Kairos's performance comes from its superior query distribution mechanism (Sec. 5) which focuses on exploiting the heterogeneity for maximum availability cycles on all resources combined, while carefully avoiding QoS violations. Note that DRS is ultimately limited by its threshold-based mechanism and misses the opportunity of utilizing different instance types for smaller queries. CLKWRK actively schedules queries on instances that do not violate QoS, thus it helps avoid more unnecessary QoS violations than Ribbon. However, unlike Kairos, it does not optimize on heterogeneous instances.

The most attractive aspect of Kairos's superior performance and design is that Kairos does not experimentally evaluate heterogeneous configurations online – unlike other methods. It uses its unique approximation method to determine a good heterogeneous configuration in one shot. *For our evaluation, we provided additional advantages to all competing schemes by allowing them to use optimal heterogeneous configuration – determined offline.* Even under this conservative evaluation, Kairos outperforms the DRS and CLKWRK schemes by up to 44% (Fig. 9).

Lastly, Kairos+ performs slightly better than Kairos. This is expected because Kairos+ employs a pruning-based online approach to find the optimal heterogeneous configuration. Nevertheless, Kairos still provides approximately the same throughput as Kairos+ without any online evaluation. Next, we discuss the impact of online configuration searches.

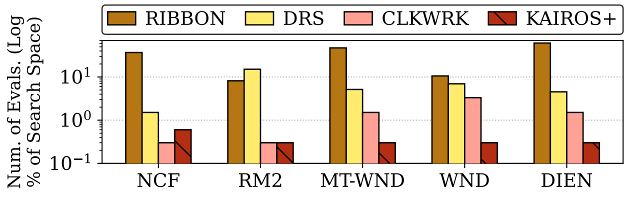

Figure 10: Kairos+ has low evaluation overhead.

8.3 Online Optimal Config. Exploration

Among all competing schemes, only Kairos and Oracle do not require online evaluation, others need to evaluate configurations online to find the optimal one for their query distribution mechanism. One may ask: how much overhead does Kairos save from avoiding online evaluation? In our previous result (Fig. 9), the impact of online evaluation is not included. Fig. 10 shows the number of evaluations each online technique requires to find its optimal configuration (on log scale). We observe that Ribbon and DRS often need to evaluate 5%-30% of the search space to reach the optimal configuration, and hence, the maximum throughput they can achieve is *delayed* by the length of this exploration period.

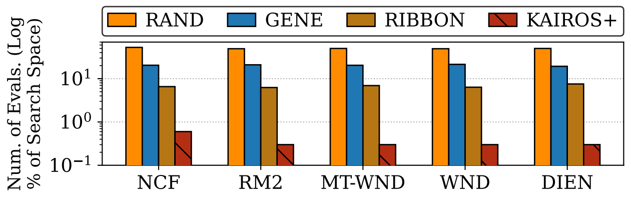

Figure 11: KAIROS+ versus competing search algorithms.

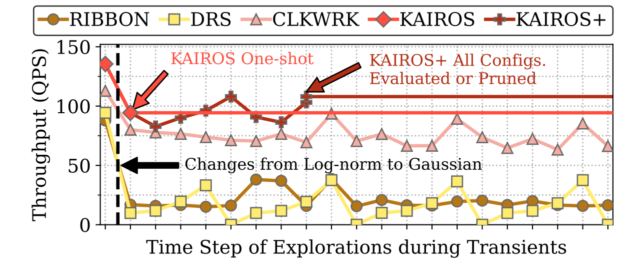

Figure 12: When query size probability distribution changes, the throughput of evaluated configurations.

For a fair comparison, in Fig. 10, we augment all competing techniques with the same online configuration exploration algorithm as KAIROS+. We note that KAIROS+ consistently evaluates less than 1% of the search space for all models, outperforming competing schemes despite using the same search algorithm. Similar trends are observed, even with other non-KAIROS+ online exploration algorithms such as genetic algorithm, simulated annealing, etc.

A related inquiry is: is calculating the throughput upper bound and leveraging it toward online exploration in KAIROS+ useful? Will any other alternative search methods such as Ribbon quickly find the same optimal configuration that KAIROS+ finds? We use random search (RAND), genetic algorithm [79] (GENE), and Ribbon's Bayesian Optimization as competing search algorithms and evaluate the number of evaluations compared to the KAIROS+ technique. We purposely provide these competing algorithms with the same sub-configuration pruning mechanism as KAIROS+ in Algorithm 1 to save some iterations. But, as Fig. 11 shows, competing methods still require significantly more online evaluations than KAIROS+. These results essentially decouple the effects of KAIROS's two components of query distribution and search space evaluation, showing that they both are critical for quickly achieving a configuration that performs well. The advantages of KAIROS as a technique that can quickly find a good configuration are further emphasized.

8.4 Timely Reaction to Load Changes

Previous results demonstrate that, besides a higher throughput, KAIROS's major benefit is from eliminating the time overhead to find a promising configuration. Fig. 12 provides further experimental demonstration to substantiate this. When the query load or its distribution changes, the optimal configuration changes. In Fig. 12, the query-size distribution changes from Log-normal to Gaussian (at the vertical dashed line) for a sample model (RM2), and the first 20 evaluated configurations of transient response are shown. All schemes respond to this change and restart the search process.

In Fig. 12, KAIROS reaches a near-optimal configuration in one shot. Without online evaluation, its throughput is 2× more than that of Ribbon and DRS. CLKWRK is the most competitive scheme, but it uses 9 evaluations to reach the same throughput as KAIROS. To find the optimal configuration, KAIROS+ performs upper-bound-assisted online search: by its 8^{th} evaluation, all possible configurations in the search space have been either evaluated or pruned by upper bound, and as expected, its final throughput is slightly higher than KAIROS. The throughput of KAIROS and KAIROS+ are within 15% of the Oracle (not shown for better figure readability).

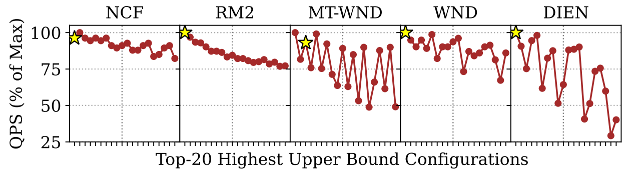

Figure 13: Actual throughput of the configs. sorted by upper bounds. KAIROS's selected configuration is marked by star.

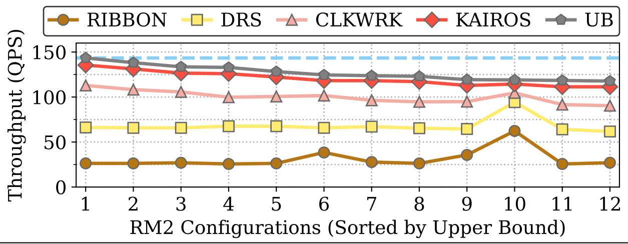

Figure 14: Impact of changing query distribution scheme. The dashed horizontal line indicates Oracle.

8.5 Source of KAIROS's Effectiveness

KAIROS's effectiveness comes from its upper-bound method that quickly finds a near-optimal heterogeneous configuration. Fig. 13 explains why it can find such configurations. In this figure, the top-20 highest upper bound configurations are shown with their actual throughput in red. The star corresponds to the configuration that KAIROS picks after applying its similarity-based criteria to choose the most promising one from the top 10 configurations. We make two key observations. Firstly, the actual optimal configuration is always among the top 10 candidates. Secondly, although the actual throughput does not follow the upper bounds in strict monotonic order (i.e., a higher upper bound always means higher throughput), they still follow the same trend, indicating the optimal configuration is among the highest upper bound ones. This explains why KAIROS can find a near-optimal configuration without online evaluations.

To further demonstrate the source of effectiveness of KAIROS, in Fig. 14, we pick the RM2 model and plot the experimental throughput for KAIROS's top upper bound configurations when changing the query distribution scheme to Ribbon, DRS, and CLKWRK. This setting is chosen to better understand how the query distribution mechanism and heterogeneous configuration search by KAIROS are co-designed, and replacing the query distribution mechanism with any other would result in a worse performance of configuration search. We make three observations: (i) The upper bound (UB) is lower than but close to the Oracle throughput (dashed), indicating that the upper bound is relatively tight and meaningful to be used in practice. (ii) The calculated (UB) and KAIROS's experimentally observed throughput are close to each other and follow the same trend.

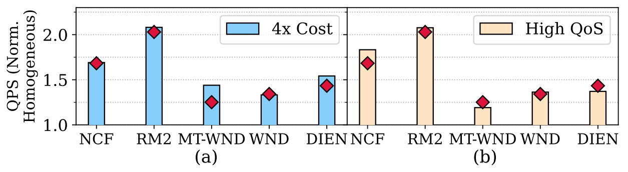

Figure 15: Kairos when: (a) budget is scaled by 4x; (b) using a higher QoS target. Red dot: Fig. 8 results.

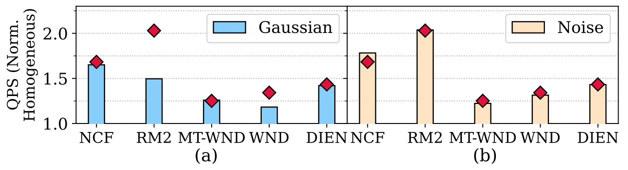

Figure 16: Kairos's effectiveness when: (a) batch size follows Gaussian distribution; (b) query latency noise presents.

This substantiates the design decision to approximate real throughput using the upper bound. (iii) Kairos's query-distribution mechanism is a key source of its effectiveness. If Kairos only chooses a configuration based on the upper bound without employing its query-distribution mechanism, the actual throughput will yield far from expectation (loose bound) – underscoring the effectiveness of Kairos's query-distribution mechanism.

8.6 Parameter Robustness Evaluation

Finally, we evaluate Kairos's robustness against different parameters. Fig. 15(a) shows that Kairos's heterogeneity approach offers a substantial improvement over homogeneous configurations when the cost budget scales. Notice that non-Kairos schemes would struggle more in finding a good configuration as the search space is increased by 4×. Similarly, in Fig. 15(b), when the QoS targets are set 20% higher, Kairos continues to offer similar improvements as before (Fig. 8 results are marked by the red dot).

Recall that the query batch size patterns may change over time, and Kairos can quickly respond to the new patterns without online evaluations (Sec. 8.4). In Fig. 16(a), we show that Kairos still yields significant benefits over homogeneous serving on Gaussian distributed batch sizes. Note that the Oracle could also yield lower improvements – hence, the relative improvement compared to the previous distribution (red dots) also decreases for some models. Lastly, since the query distribution scheme of Kairos makes a realistic assumption that the inference latency can be accurately predicted, we also show Kairos's effectiveness when this assumption is relaxed. In Fig. 16(b), we intentionally inject an additive Gaussian white noise with 5% variance in latency prediction to emulate performance variability in the cloud [80]. Our results suggest that Kairos is not sensitive to such noise that is common due to interference or transient hardware degradation, and continues to offer similar improvements.

9 CONCLUSION

Kairos demonstrates that a mixture of heterogeneous compute instances can be effectively utilized to maximize the inference query throughput under QoS and cost constraints. Kairos's upper-bound-based method eliminates the need for online configuration exploration, and Kairos intelligently distributes queries among heterogeneous instances. Our evaluation shows that Kairos significantly outperforms state-of-the art techniques. We expect that Kairos's two-pronged approach could potentially find interesting applicability in other computer system optimization problems.

ACKNOWLEDGMENTS

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001, and United States Air Force Research Laboratory Cooperative Agreement Number FA8750-19-2-1000. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering, or the United States Air Force. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1049–1062, 2019.
[2] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.
[3] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.
[4] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.
[5] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.
[6] Lin Ning and Xipeng Shen. Deep reuse: streamline cnn inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*, pages 438–448, 2019.
[7] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 493–506. IEEE, 2021.
[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 972–986. IEEE Computer Society, 2020.
[9] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 639–653, 2021.
[10] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.
[11] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
[12] Subho Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar Iyer. Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters. In *International Conference on Machine Learning*, pages 629–641. PMLR, 2020.
[13] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time*

and Embedded Technology and Applications Symposium (RTAS), pages 190–201. IEEE, 2018.

[14] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[15] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 624–638, 2021.

[16] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

[17] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[19] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[20] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[21] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[22] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 499–510, 2015.

[23] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[24] Cosmin Rusu, Alexandre Ferreira, Claudio Scordino, and Aaron Watson. Energy-efficient real-time heterogeneous server clusters. In *12th IEEE real-time and embedded technology and applications symposium (RTAS'06)*, pages 418–428. IEEE, 2006.

[25] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 409–420. IEEE, 2017.

[26] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 481–498, 2020.

[27] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224. IEEE, 2012.

[28] Kai Hwang, Xiaoying Bai, Yue Shi, Muyang Li, Wen-Guang Chen, and Yongwei Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on parallel and distributed systems*, 27(1):130–143, 2015.

[29] Ang Li, Xiaowei Yang, Ming Zhang, and S Kandula. Cloudcmp: Shopping for a cloud made easy. *HotCloud*, 10:1–7, 2010.

[30] Joel Scheuner and Philipp Leitner. A cloud benchmark suite combining micro and applications benchmarks. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 161–166, 2018.

[31] Mohammad Shahrad and David Wentzlaff. Availability knob: Flexible user-defined availability in the cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 42–56, 2016.

[32] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465, 2017.

[33] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 329–341, 2013.

[34] Seyedhamid Mashhadi Moghaddam, Sareh Fotuhi Piraghaj, Michael O'Sullivan, Cameron Walker, and Charles Unsworth. Energy-efficient and sla-aware virtual machine selection algorithm for dynamic resource allocation in cloud data centers. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 103–113. IEEE, 2018.

[35] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. Mocha: Multinode cost optimization in heterogeneous clouds with accelerators. 2021.

[36] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[37] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[38] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 937–954, 2020.

[39] Changyeon Jo, Youngsu Cho, and Bernhard Egger. A machine learning approach to live migration modeling. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 351–364, 2017.

[40] Young Geun Kim and Carole-Jean Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1082–1096. IEEE, 2020.

[41] Seulki Lee and Shahriar Nirjon. Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–29. IEEE, 2020.

[42] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 305–317. IEEE, 2019.

[43] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[44] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. {ALERT}: Accurate learning for energy and timeliness. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 353–369, 2020.

[45] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[46] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–17, 2021.

[47] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[48] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[49] Michael Orr and Oliver Sinnen. Optimal task scheduling for partially heterogeneous systems. *Parallel Computing*, 107:102815, 2021.

[50] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[51] Richard M Karp, Umesh V Vazirani, and Vijay V Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 352–358, 1990.

[52] Roy Jonker and Anton Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[53] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[54] David F Crouse. On implementing 2d rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1679–1696, 2016.

[55] Shuang Chen, Angela Jin, Christina Delimitrou, and José F Martínez. Retail: Opting for learning simplicity to enable qos-aware power management in the cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–168. IEEE, 2022.

[56] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[57] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46: 259–271, 1990.

[58] Yossi Azar and Amir Epstein. Convex programming for scheduling unrelated parallel machines. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 331–337, 2005.

[59] John P Lehoczky. Real-time queueing theory. In *17th IEEE Real-Time Systems Symposium*, pages 186–195. IEEE, 1996.

[60] Robert B Cooper. Queueing theory. In *Proceedings of the ACM'81 conference*, pages 119–122, 1981.

[61] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[62] Rena Nainggolan, Resianta Perangin-angin, Emma Simarmata, and Astuti Feriani Tarigan. Improved the performance of the k-means cluster using the sum of squared error (sse) optimized by using the elbow method. In *Journal of Physics: Conference Series*, volume 1361, page 012015. IOP Publishing, 2019.

[63] Nvidia triton inference server. URL https://docs.nvidia.com/deeplearning/triton-inference-server/.

[64] grpc. URL https://grpc.io/.

[65] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[66] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing deep learning recommender systems training on cpu cluster architectures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[67] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

[68] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[69] Andreas Argyriou, Miguel González-Fierro, and Le Zhang. Microsoft recommenders: Best practices for production-ready recommendation systems. In *Companion Proceedings of the Web Conference 2020*, pages 50–51, 2020.

[70] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 157–168. IEEE, 2020.

[71] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[72] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[73] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 43–51, 2019.

[74] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 5941–5948, 2019.

[75] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[76] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[77] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[78] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[79] Thomas Wortmann and Giacomo Nannicini. Black-box optimisation methods for architectural design. 2016.

[80] Jamie Ericson, Masoud Mohammadian, and Fabiana Santana. Analysis of performance variability in public cloud computing. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 308–314. IEEE, 2017.