2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 978-1-6654-9998-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/RTAS54340.2022.00020

FA2: Fast, Accurate Autoscaling for Serving Deep Learning Inference with SLA Guarantees

Kamran Razavi[†], Manisha Luthra[†], Boris Koldehofe[†,‡], Max Mühlhäuser[†], Lin Wang[†,§]

[†]Technische Universität Darmstadt
[‡]University of Groningen
[§]Vrije Universiteit Amsterdam

***Abstract*—Deep learning (DL) inference has become an essential building block in modern intelligent applications. Due to the high computational intensity of DL, it is critical to scale DL inference serving systems in response to fluctuating workloads to achieve resource efficiency. Meanwhile, intelligent applications often require strict service level agreements (SLAs), which need to be guaranteed when the system is scaled. The problem is complex and has been tackled only in simple scenarios so far.**

This paper describes FA2, a fast and accurate autoscaler concept for DL inference serving systems. In contrast to related works, FA2 adopts a general, contrived two-phase approach. Specifically, it starts by capturing the autoscaling challenges in a comprehensive graph-based model. Then, FA2 applies targeted graph transformation and makes autoscaling decisions with an efficient algorithm based on dynamic programming. We implemented FA2 and built and evaluated a prototype. Compared with state-of-the-art autoscaling solutions, our experiments showed FA2 to achieve significant resource reduction (19% under CPUs and 25% under GPUs, on average) in combination with low SLA violations (less than 1.5%). FA2 performed close to the theoretical optimum, matching exactly the optimal decisions (with the least required resources) in 96.8% of all the cases in our evaluation.

I. Introduction

With the rapid advancements of deep learning (DL) techniques, DL inference has become a popular component in various modern intelligent applications and services [1]–[4]. Some applications involve a single deep neural network (DNN) for inference tasks like object recognition or natural language understanding. Others, such as digital assistant services (e.g., Amazon Alexa), involve a more complex chain of DNNs for inference tasks, including speech recognition, question interpretation, and text-to-speech to serve user requests [2], [5]. Most of these applications are mission-critical or user-interactive, imposing strict service-level agreements (SLAs) on the inference latency, e.g., the end-to-end latency should be bounded by a deadline [1], [3], [6].

One critical concern in provisioning applications with DL inference is on *resource efficiency*. Resource efficiency is essential simply because DNNs typically require intensive computation, which imposes prohibitive costs and stringent deployment constraints when the resources are limited, e.g., in the edge environment [7]. Ideally, the amount of resources assigned to each of the DNNs in an application should be just right, matching the real-time workload (measured in requests per second, RPS) of the application while guaranteeing the SLA. We refer to this problem as "resource autoscaling," which has become a critical challenge in building efficient DL inference serving systems [2], [5].

Resource autoscaling is a non-trivial problem in general and has been heavily explored in various contexts, including stream processing [8]–[10], serverless computing [11], [12], and microservices [13]–[16]. The unique properties of DL inference serving systems make the situation even worse. In particular, we identify the following factors in DL inference serving systems, which, when combined, add new challenges and significantly exacerbate existing ones: (a) The DNNs for an application are orchestrated with data **dependencies** specified by a dataflow graph. Thus, the resource scaling decision for one DNN may affect that for other (downstream) DNNs due to workload changes [1], [2]. (b) Inference requests may follow **uncertain** execution paths, where, depending on the output of a DNN, requests may be forwarded to different succeeding DNNs (thus different paths) for further processing [5]. (c) DL inference serving systems typically require strict **SLA guarantee** on the end-to-end latency over all possible execution paths specified in the dataflow graph [1], [3]. (d) Request **batching** is widely used for DNNs to improve resource utilization by trading processing time for throughput [1], [6], [17]. Thus, the batch size for one DNN may affect the scaling decision of others and also the end-to-end latency. These factors, when combined, make existing autoscalers inapplicable or inefficient for DL inference serving systems, calling for new solutions.

In this paper, we present a comprehensive study of resource autoscaling for DL inference serving systems and present FA2—a fast, accurate resource autoscaler tailored for efficient provisioning of DL inference-based applications. Given an application with a set of DNNs orchestrated with a dataflow graph, FA2 makes collective resource scaling decisions, including both the number of instances and the corresponding batch size for each of the involved DNNs adaptively. Our goal is to minimize the total amount of resources occupied by all the DNNs of an application while ensuring the SLAs of all the execution paths in the application.

To this end, we first present a graph-based model to capture all the aforementioned factors holistically. In particular, we model the processing time and the worst-case queuing delay at all DNNs explicitly and take both into account when calculating the end-to-end delay for all the execution paths.

978-1-6654-9998-9/22/$31.00 ©2022 IEEE
DOI 10.1109/RTAS54340.2022.00020

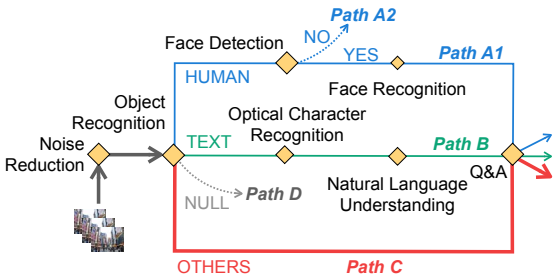

Fig. 1. The execution graph of an example modern intelligent application taking images as input. Depending on the object recognition result, different execution paths will be taken for each inference request.

This design choice is critical in guaranteeing SLAs on all the execution paths with dependency and uncertainty. Through targeted graph transformation, we relax the problem and present an efficient resource autoscaling algorithm based on dynamic programming. The proposed algorithm runs fast and generates accurate scaling decisions. In contrast to existing solutions where simple heuristics are used for scaling decision-making [2], [5], our design is principled and achieves almost optimal performance.

In short, this paper makes the following contributions. After presenting the background for DL inference serving systems and identifying the challenges (§II), we

- present the design of FA2, including its overall architecture and components (§III);
- introduce a comprehensive graph-based model to capture the resource scaling problem in DL inference serving systems and present our scaling algorithm based on graph transformation and dynamic programming (§IV);
- build a system prototype for FA2 (§V) and evaluate it with synthetic and real-world workload traces (§VI). Overall, FA2 outperforms the state-of-the-art autoscalers, achieves significant resource reductions (19% under CPUs and 25% under GPUs), while showing only slightly over 1% SLA-violation rates, meaning that the latency at the 99-th perentile can mostly be guaranteed. Our results also reveal that FA2 matches the theoretical optimum scaling decisions in around 96% of the cases.

§VII summarizes related work. §VIII draws final conclusions.

II. Background and Motivation

This section presents the background on deep learning (DL) inference and discusses the resource autoscaling problem in DL inference serving systems. We then identify the challenges in efficient autoscaling and motivate a new autoscaler design.

A. Deep Learning Inference Serving

With the fast advancement of DL techniques, a variety of modern applications such as intelligent personal assistants, augmented reality (AR), and autonomous driving adopt deep neural networks (DNNs) as a fundamental building block [1], [18], [19]. Typically, DL is used for inference tasks such as object detection and recognition and voice recognition, where input data in the form of images or voice recordings is sent to DNNs, producing predictions for the input. For sophisticated applications, multiple DNNs may be involved in the process where all these DNNs are chained or orchestrated into a complex dataflow graph, represented by a direct acyclic graph (DAG), to process the input data step by step. Each input to the system spawns an *inference request* which needs to be handled by a subset of the DNNs specified in the dataflow graph sequentially [2]. Figure 1 depicts the dataflow graph of an example modern intelligent application.

One salient feature of these DL-based applications is being timed critical since they are mostly either user-interactive (e.g., personal assistant and AR) or mission-critical (e.g., autonomous driving) [2], [4], [18], [20]. It is often required that the end-to-end latency in serving each inference request by the system has to meet a certain threshold dictated by the application for the request to be useful. The DL inference serving system provisioning such applications thus needs to provide strict service-level agreements (SLAs) where the tail inference latency has to be bounded by the threshold to guarantee the overall usability of the application [1], [3].

B. Resource Autoscaling

Resource scaling concerns dynamically adjusting the computing resources assigned to applications according to their changing workload. The goal is to improve resource efficiency while meeting the resource demands of the application. Resource scaling has been extensively explored in cloud-based systems, including stream processing [8]–[10], serverless computing [11], [12], and microservices [13]–[16].

There are generally two types of resource scaling mechanisms: *horizontal* scaling and *vertical* scaling. Horizontal scaling constructs a base instance (with a fixed amount of resources) built into a virtual machine or container and decides the number of instances required to support the real-time workload (e.g., throughput) at runtime. Vertical scaling adjusts the performance of every single instance by changing the resource allocation for the instance. Most data-intensive computing systems adopt horizontal scaling considering its simplicity [8]. Horizontal scaling is also beneficial to DL inference serving, as confirmed in Figure 2, where we show the latency-throughput comparison for an object detection model (Inception V2) under three different resource configurations. We observe that given the same total amount of resources (four CPU cores here), the configuration with the smallest instance provides higher throughout under the same latency. This is mainly because, even with batching, DL inference cannot be fully parallelized to take full advantage of the big instances. Therefore, we focus on horizontal scaling in this paper.

Apart from the scaling mechanism, it is critical to answering questions like *when* and *how* to scale, and these are typically handled by a scaling controller known as the resource autoscaler [8], [21]–[23]. Through conventional monitoring tools, symptoms of under- and over-provisioning can be detected, and whether to make a change is decided. The resource autoscaler then identifies the causes of the symptoms (e.g., bottleneck or underutilized components) and performs

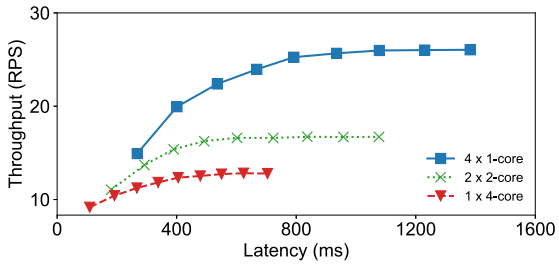

Fig. 2. Performance comparison between horizontal and vertical scaling policies with respect to throughput and latency on the object detection (OBJD) model in Table II. Each data point represents the result obtained with a batch size in the range of $[1, 9]$ respectively.

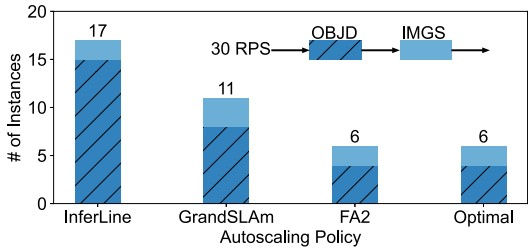

Fig. 3. Number of instances required by different autoscaling policies. InferLine neglects the combinatorial nature of the models and iterates on all models one by one and GrandSLAm uses a predefined stack allocation for DNN models, both resulting in need of extra resources (3x and 2x, respectively) compared with the optimal resources provided by Gruboi.

scaling actions accordingly. Usually, the detection of suboptimal provisioning is based on performance metrics such as CPU/memory utilization and backpressure or congestion [9], [21]. Resource autoscalers then make scaling decisions using simple threshold-based heuristics, leveraging control-theoretic models, or adopting complex queuing theory models based on workload prediction [23], [24].

C. Autoscaling Challenges in DL Inference Serving

Designing an efficient resource autoscaler for DL inference serving systems is non-trivial. In particular, we identify the following challenges, all of which combined distinguish the autoscaling problem in DL inference serving systems from those studied in other systems such as stream processing [8].

- **Dependency:** Modern DL inference systems typically involve multiple DNNs orchestrated with a DAG [2]. The edges in the DAG indicate the data dependencies between the DNNs, leading to the tight coupling of the scaling decisions for different DNNs. Hence, the scaling decisions for all the DNNs need to be coordinated holistically.
- **Uncertainty:** Depending on the output of the preceding DNN, an inference request may follow different execution paths in the dataflow graph [5]. This brings significant uncertainty in the system workload and renders prediction models based on queuing theory inaccurate.
- **SLA guarantee:** DL inference requests need to be processed within a certain amount of time in order to be useful, which refers to SLA guarantees [1], [2], [5]. It is particularly challenging to guarantee SLA in DL inference serving systems since requests following different execution paths may be specified with different SLAs, and due to uncertainty, the type (thus the corresponding SLA) of a request cannot be known a priori.
- **Batching:** DL inference serving systems typically adopt request batching, which is effective in improving resource utilization [6], [17], [19], as shown in Figure 2. In essence, we trade off latency for throughput with SLA guaranteed. Changing the batch size of one DNN leads to changes in the throughput and latency, affecting other DNNs due to dependency and SLA guarantee.

Existing autoscaler designs for DL inference serving systems are mainly based on decoupling the batch size and the autoscaling decision-making problems using simple heuristics. A popular approach is called "split-and-conquer" where we split the SLA over the DNNs following a proportional policy and make scaling decisions locally for each DNN, as done in GrandSLAm [1] and InferLine [2]. Despite convergence issues already noted in stream processing engines [8], such an approach is conservative and can lead to significant resource over-provisioning [5]. To confirm this observation, we perform an experiment using a simple DL inference serving pipeline consisting of two DNNs for object detection (OBJD) and image segmentation (IMGS), respectively. The results are shown in Figure 3. With a steady input of 30 RPS, GrandSLAm and InferLine require almost $2\times$ and $3\times$ of the optimal number of instances (obtained by the Gurobi solver), respectively. However, the exploration space for the optimal solution is exponentially large, and thus exhaustive search is not practical. In the rest of this paper, we will adopt a principled approach to tackle the autoscaling problem in DL inference serving systems. We will show how FA2 achieves (almost) optimal scaling decisions while being computationally efficient.

III. FA2 System Design

In this section, we describe the design of FA2—fast, accurate autoscaling for DL inference serving with SLA guarantee. We first provide an overview of the system architecture and then discuss the system's major components.

System overview. An overview of the FA2 architecture is depicted in Figure 4. Our system consists of three main components (monitor, optimizer, and controller). The *monitor* keeps monitoring statistics about the distribution of request arrivals, i.e., the average number of requests that have followed each execution path in the execution graph. The *optimizer* takes the application execution graph with specified SLAs over execution paths as well as the request distribution from the monitor as input and makes scaling decisions (i.e., number of instances and batch size for every DNN in the execution graph) by solving an optimization problem. Afterward, the *controller* informs the DL inference serving system and reconfigures the system according to the scaling decision. The DL inference serving system deploys the DNN instances, which serve re-

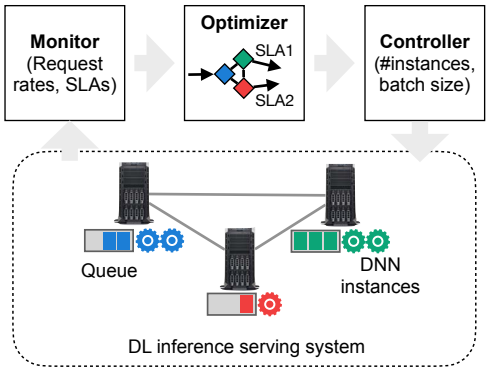

Fig. 4. An overview of the FA2 architecture. The monitoring service collects the metric data from the DL inference serving system. The optimizer makes scaling decisions for the DNNs. The controller enforces the scaling decisions by configuring the inference serving system.

quests from a central queue preceding all the DNN instances of the same type at each server. The system runs periodically to adapt to request rate changes as well as possible request distribution drifts. Similar to DS2 [8] for stream processing (which is not applicable to DL inference serving due to the challenges we have identified), FA2 aims to make holistic scaling decisions for all the DNNs in a single shot and satisfies the SASO properties (i.e., stability, accuracy, short settling time, and not overshooting) in control theory [25].

Monitor. The monitor pulls two types of metrics from the DL inference serving system in a predefined time interval: (a) the sequence of DNNs (i.e., the execution path) each inference request has traversed while in the system, and (b) the end-to-end processing time of each inference request following the execution path. The former is used to calculate the average number of requests that have followed a specific execution path in the past, while the latter is used to decide if a scaling operation is necessary depending on the ratio of SLA violations for the served requests.

Optimizer. The optimizer aims to generate the scaling decision—the optimal configuration (including the number of instances per DNN and the corresponding batch size for each DNN) to achieve the highest resource efficiency while respecting all the SLAs in the system using the metrics reported from the monitor. To this end, the optimizer first builds a graph-based model incorporating both the request processing and queuing delays. Then, it performs graph transformation to simplify the graph-based model and provides an efficient algorithm based on dynamic programming to calculate the scaling decision. We will elaborate them in §IV. Finally, the optimizer passes the scaling decision to the controller to enforce the system configuration in the system.

Controller. The controller is responsible for reconfiguring the system according to the scaling decision generated by the optimizer. This reconfiguration includes the batch size and the number of instances for each DNN in the inference serving system. To this end, the controller first compares the new configuration from the optimizer with the current system configuration. If both configurations are the same, no reconfiguration will be needed. Otherwise, the controller sends the new batch size information to the queues at each DNN and brings up/down DNN instances based on the difference between the two configurations. Note that if the only change in the new configuration is the batch size, the system can be reconfigured immediately without any delay.

TABLE I
NOTATIONS

Symbol	Description
G	Application's dataflow graph (a DAG)
S	Set of registered DNNs
s	A DNN model from set S
P	Set of possible execution paths
p	An execution path in set P
n_s	Number of instances for DNN s
b_s	Batch size of DNN s
Int_s	Computation intensity of DNN s
$d_s(b_s)$	Processing time for DNN s with batch size b_s
$q_s(b_s)$	Max. queuing time at s with batch size b_s
$l_s(b_s)$	Total time spent at DNN s, i.e., $d_s(b_s) + q_s(b_s)$
$h_s(b_s)$	Throughput of s with batch size b_s
SLA_p	Service-level agreement for path $p \in P$
λ_p	Request rate for execution path $p \in P$
λ_s	Request rate at DNN s
$OPT(s,t)$	Optimal resources consumed by s and all its successors under time budget t

IV. Autoscaling Problem and Algorithm

In this section, we focus on the optimization problem that the optimizer needs to solve to produce accurate scaling decisions. In particular, we adopt a principled approach and model the system comprehensively. We first model the processing time of DNNs and then provide a model for the worst-case queuing delay at the queue preceding the DNN instances of the same type. With these models, we formulate the autoscaling problem with an integer program. We then propose an efficient algorithm to solve the problem based on graph transformation and dynamic programming. Table I summarizes the notations we use in the paper.

A. DNN Performance Modeling

To facilitate decision making at the optimizer, FA2 requires to know the performance, i.e., throughput $h(b)$ and latency $d(b)$ with respect to the batch size b, of each DNN instance. Prior work has demonstrated that the performance of DL models is quite predictable, especially of those for deep learning inference [1], [3], [19]. We follow the same line and use profiling data and robust regressions [26] to build models for all the DNNs in the system. Other more sophisticated performance modeling methods [27] can also be employed. Such performance models can be built offline and be reused throughout the lifecycle of the DNNs as long as the size of the DNN instance stays the same, which is true since only horizontal scaling will be considered.

In contrast to existing work [1], [19] which suggests a linear relationship between batch size and latency, we apply some slight changes that improve the prediction accuracy

considering that a larger batch size could potentially better utilize non-shareable resources such as CPU caches other than the computing units. In particular, we use a second-order quadratic polynomial $d(b) = \alpha b^2 + \beta b + \gamma$ for latency prediction under a given amount of resources, where α, β, and γ are parameters and they will be fitted with profiling data. The throughput of a DNN instance is directly given by $h(b) = b/d(b)$. Our evaluation with 100K inferences for each DNN latency profiling (see Figure 6) confirms that the quadratic model is more accurate than a linear model with a much smaller mean squared error.

B. Queuing Delay Modeling

The queuing delay at a given DNN can be affected by the following factors: the average request arrival rate λ reported by the monitor, the batch size b, processing time of the DNN instance $d(b)$, and the number of DNN instances n. For a request at a specific DNN, the worst-case queuing delay can be captured in two scenarios: (a) Assume a DNN instance is idle and waiting to serve requests. When a request arrives, it has to wait for another $b-1$ requests to come until a batch can be formed and served by the DNN instance. In this case, the first request has to be queued for $(b-1)/\lambda$ time before it can be processed. (b) Assume a DNN instance has been assigned a batch for processing, and following a round-robin policy, the next batch for the same instance containing the $(nb+1)$-th through $(nb+b)$-th requests have arrived. Thus, the batch needs to wait for the DNN instance to be freed from processing the previous batch before it can be processed, even after the arrival of the last request of the new batch for the DNN instance, with a queuing time of $d(b) - (nb+1)/\lambda$ (the worst case happens to the first request in the new batch). By combining the above two cases, the worst-case queuing latency at a DNN with batch size b and n instances is given by

$$q(b,n) = \max\left(\frac{b-1}{\lambda}, d(b) - \frac{nb+1}{\lambda}\right). \tag{1}$$

Meanwhile, the total throughput $n \cdot h(b)$ of all the n instances for the same DNN has to match the arrival rate, i.e., $n \cdot h(b) = \lambda$. With some simple manipulation we can obtain $d(b) = nb/\lambda$. Plugging this equation in the above queuing latency formula, we observe that the second term is always negative. Therefore, the worst-case queuing delay can be simplified as

$$q(b) = \frac{b-1}{\lambda}. \tag{2}$$

We will use this equation for modeling the worst-case queuing delay in the problem formulation and our algorithm.

C. Problem Formulation

Based on the DNN performance model and the queuing delay model we have introduced, we now provide a formal description for the resource autoscaling problem. We denote the dataflow graph of the application by $G = (S, E)$ where node-set S represents the set of DNNs and edge-set E represents the data dependencies between the DNNs. On the dataflow graph, the application specifies a set of execution paths denoted by set P. For each path $p \in P$, $S_p \subseteq S$ denotes the set of DNNs on p, and SLA_p denotes the SLA specified on the end-to-end latency of this path. The aggregate request arrival rate for the application is denoted by λ, which may change over time. Due to the uncertainty property of DL inference serving systems as discussed in Section II, SLA-aware request scheduling (e.g., priority-based scheduling or earliest deadline first) at a DNN is not possible since it is unknown which execution path will be taken by each request before the request leaves the system.

The monitoring system continuously reports to the controller the execution path that has been taken by each of the inference requests in a given period in the past. From such information, we derive the average number of requests served by each of the execution paths $p \in P$, denoted by λ_p. To ensure system stability, the aggregate throughput of all the instances for a DNN should be no less than the expected request rate, i.e., for any DNN $s \in S$, $h_s(b_s) \cdot n_s \geq \sum_{s \in p: p \in P} \lambda_p$. Such a constraint ensures that all the DNNs are sufficiently provisioned. As a result, queuing of inference requests at each DNN will be under control.

The optimization problem is to decide n_s and b_s for all $s \in S$ such that under a given workload, none of the SLAs specified by the execution paths are violated. The goal is to minimize the aggregate amount of resources used for all the DNNs. The problem can be formulated with the following integer program (IP):

$$\begin{aligned} \min \quad & \sum_{s \in S} n_s + \delta(\sum_{s \in S} b_s) \\ \text{subject to} \quad & \sum_{s \in S_p} d_s(b_s) + q_s(b_s) \leq SLA_p, \forall p \in P, \\ & h_s(b_s) \cdot n_s \geq \sum_{s \in p: p \in P} \lambda_p, \forall s \in S, \\ & b_s, n_s \in \mathbb{Z}^+, \forall s \in S. \end{aligned} \tag{3}$$

In the objective function, in addition to the total number of instances, we introduce a small penalty term $\delta(\cdot)$ on the total batch sizes. This penalty ensures we will use the minimal possible batch sizes under the optimal number of instances due to the fact that a larger batch size without enough queries in the system not only increases the processing and queuing delay (which can be valuable for other DNNs) but also does not increase the system's utilization. The first constraint ensures that all the SLAs will be satisfied. We omit the network latency as we assume high-throughput, low-latency network links are available in data centers. Our model is capable of incorporating network latency. The second constraint enforces the stability of the system. The number of instances and the batch size should both be positive integers, as shown in the last line. The objective is to minimize the resources, i.e., the total number of DNN instances used by the application. Each DNN can have a different batch size and number of instances. Thus, the solution space for the IP increases exponentially for every new DNN added to the system, making it hard to explore the space to find the optimal solution exhaustively. For example, for just 10 models with the maximum batch size of 128 and the

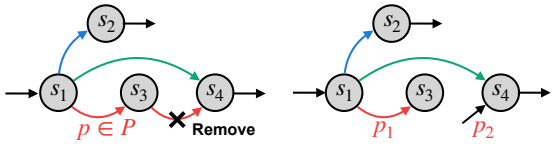

Fig. 5. Graph transformation example: (left) the original dataflow graph, (right) the transformed graph containing only egress aggregators.

maximum number of instances of 8, the feasible space could be as large as $(128 \times 8)^{10} = 1024^{10} \approx 10^{30}$). To address this issue, we provide an efficient algorithm to solve the IP based on graph transformation and dynamic programming, as detailed in the following sections.

D. Graph Transformation

Our autoscaling algorithm consists of two parts: graph transformation and dynamic programming. We now describe why and how we transform the graph model and elaborate on the dynamic programming design in the next section.

The autoscaling problem is challenging due to the dataflow dependency, combined with SLA guarantee and batching, as discussed in Section II. This problem is illustrated in Figure 5 (left), where two execution paths (green and red-colored edges) share DNNs s_1 and s_4 in a fork-join fashion, leading to a deadlock situation in scaling these DNNs if we look at these DNNs one by one. In this example, if we decide the instance numbers and batch sizes for s_1 and s_4 from the green-colored path, the configurations may be sub-optimal or even infeasible for the same DL models in the red-colored path with a different SLA and more DNNs on the path. More specifically, if an application's dataflow graph meets the following condition, the problem can be solved efficiently: The dataflow graph does not contain both ingress and egress aggregators at more than one DNN. Here, an ingress aggregator represents a DNN that receives requests from multiple preceding DNNs, and an egress aggregator represents a DNN that sends requests to multiple succeeding DNNs. If we can avoid having both types of aggregators in the dataflow graph, we can eliminate the dependency issue explained in the above example. This property significantly reduces the complexity of the autoscaling problem. Without loss of generality, we choose to avoid ingress aggregators.

To ensure the above property, we apply graph transformations to remove a subset of edges from the dataflow graph. To this end, we define a new metric called *sharing degree* for each edge, which captures the number of execution paths on which the edge is being used. The calculation of the sharing degree for each edge can be done by traversing through all the execution paths. After that, we loop through all the ingress aggregator DNNs and pick the edge with the lowest sharing degree to remove. Since removing the edge from the dataflow graph requires splitting the execution paths that use this edge to be partitioned into two parts, choosing the ones with the lowest sharing degree leads to the least number of execution paths we need to partition. To partition an execution path, we need to split the SLA specified for that execution path into two parts, and then the two path segments can be treated as entirely independent execution paths with their own SLAs. We decide to split the SLA proportionally for the two path segments based on a metric called *intensity* (denoted by Int), which characterizes how intensive the computation is for the DNNs on a path segment. Other optimizations considering the queuing delay of the DNNs can also be applied. We compute the intensity of a DNN by averaging the processing time of the DNN over a set of batch sizes (here, we use batch sizes in $[1, 16]$ as the processing time with a larger batch size may violate SLAs solely). For a path p to be split, we assume p_1 is the first segment before the edge to be removed, and p_2 is the other segment. The SLA for the first segment can thus be calculated as

$$SLA_{p_1} = \frac{\sum_{s \in p_1} Int_s}{\sum_{s \in p} Int_s} \cdot SLA_p. \tag{4}$$

The SLA for p_2 can be computed analogously. We repeat the above procedure until no ingress aggregators can be found in the dataflow graph.

Finally, the number of requests to a DNN is directly calculated as the sum of its preceding DNNs on the original dataflow graph (not the one after transformation). To this end, we adopt a topological sort algorithm [28] to sort all the DNNs in the original dataflow graph and calculate the workload for each execution path in the transformed graph. The new graph with the workload information now can be handled with dynamic programming to obtain the optimal scaling decision, as we will explain in the next section.

E. Scaling with Dynamic Programming

Now, we focus on how to solve the resource scaling problem with dynamic programming on the transformed graph. We denote by $OPT(s, t)$ the optimal solution, i.e., the minimum resource consumption, when we consider only DNN s and all its successors in the transformed graph, given a time budget of t. We consider two cases here:

Case 1: A DNN without successors. In this case, time t can be allocated to the DNN entirely, and the optimal solution is achieved with the maximum batch size that can be handled within time t:

$$OPT(s, t) = \min_{b: l_s(b) \le t} \left(\lceil \frac{\lambda_s}{h_s(b)} \rceil \right). \tag{5}$$

Case 2: A DNN with successors. Let us denote by $M \subset S$ the set of successors of s. In this case, time t can be split into two parts: one part for the current DNN s and the other for all its succeeding path segments. The optimal solution consists of the resources consumed by s plus the sum of resources consumed by DNNs on all the succeeding paths of s. For each DNN s we denote by r_s the maximum time that can be spent on DNN, which is calculated as the minimum of the SLAs of paths that s is on, i.e.,

$$r_s = \min_{p: s \in p \wedge p \in P} SLA_p. \tag{6}$$

Algorithm 1: Dynamic Programming

input : graph G (after transformation), $\{SLA_p : p \in P\}$, $\{\lambda_s : s \in S\}$
output: $OPT(s_0, t_{max})$

1 $dp \leftarrow [|S|][t_{max}](\infty, 0)//(instance\_number, batch\_size)$
2 **for** s *in* $reversed(S)$ **do**
3 **for** t *in* $[1, t_{max}]$ **do**
4 **for** b *in* $[1, b_{max}]$ **do**
5 $q_s \leftarrow \frac{b_s - 1}{\lambda_s}$, $p_s \leftarrow d_s(b_s)$, $l_s \leftarrow p_s + q_s$
6 **if** $l_s > t$ **then**
7 **break**
8 **else if** s *has no successors* **then**
9 $dp[s][t] \leftarrow (\lceil \lambda_s / (b_s \cdot \lfloor t/p_s \rfloor \rceil, b_s)$
10 $sum \leftarrow \lceil \lambda_s / (b_s \cdot \lfloor t/p_s \rfloor \rceil$
11 **for** p *in* P **do**
12 **for** m *in* p **do**
13 **if** $(s, m) \in p$ ***and*** $t - l_s <= SLA_p$ **then**
14 $sum \leftarrow dp[m][t - l_s].in + sum$
15 **if** $sum < dp[s][t].in$ **then**
16 $dp[s][t] \leftarrow (sum, b)$

The optimal solution from DNN s given time budget t can be obtained following the recursive function:

$$OPT(s,t) = \min_{b:l_s(b) \le r_s} \left(\lceil \frac{\lambda_s}{h_s(b)} \rceil + \sum_{m \in M} OPT(m, t_m) \right) \tag{7}$$

where $t_m = \min\{SLA_p, t - l_s(b)\}$ if m is the first DNN on path p and $t' = t - l_s(b)$ otherwise. Assuming s_1 is the first DNN in the transformed graph after the topological sort, we add an artificial node s_0 preceding s_1 in the graph. The optimal resource consumption by the system is thus given by $OPT(s_0, t_{max})$ where $t_{max} = \max_{p \in P} SLA_p$ denotes the maximum time any execution path on the graph can spend.

The above recursive function leads to an algorithm based on dynamic programming, which leverages the optimal substructure of the problem and avoids redundant computation. The pseudo-code for our algorithm based on dynamic programming is listed in Algorithm 1. We define a matrix dp containing tuples of $(instance\_number, batch\_size)$. We initialize dp in line 1. Then, we iterate over all the DNNs in a reversed order of the topologically sorted dataflow graph, the possible time budget, and the batch size. For each DNN under the given time budget and batch size, we compute its processing time and queuing time to see if the time budget is possible. If so, we apply fill the dp matrix with Equation 5 (line 9) and Equation 7 (lines 10–14). Finally, we use backtrack in the filled dp matrix to obtain the optimal number of instances and batch size for each DNN. The time complexity of the proposed algorithm is dominated by the dynamic programming part, where we need to iterate over multiple dimensions to fill in the DP table. This time is calculated as $O(|S| \cdot t_{max} \cdot b_{max} \cdot |S|)$ where $|S|$ is the number of DNNs, t_{max} is the number of time slots (of 1 ms length) conditioned by the largest SLA, b_{max} is the maximum possible batch size. Also, the dominant data space needed in Algorithm 1 is for the DP table, an array of size $|S| \cdot t_{max}$ holding two-tuples of integers.

V. Implementation

We implemented FA2, including the monitor, optimizer, and the controller in Python. The source code of the FA2 runtime framework is available at [29]. The DNNs are implemented with TensorFlow [30] which is an open-source framework for machine learning. Each DNN instance is built with a Docker container [31] with a pre-specified amount of resources running the target DNN inside the container. FA2 leverages Kubernetes [32] to orchestrate and manage the containers in the system. In particular, FA2 sends control commands to Kubernetes for scaling in/out the DNNs and reconfiguring the batch size for the instances of each DNN.

Each DNN is built with two components: (a) a central queue in front of all the instances for the DNN, and (b) a set of DNN instances (the number of instances is given by FA2) running in Docker containers which fetch requests from the central queue for processing. The central queue holds pending requests and composes batches to feed the DNN instances according to the batch size configuration given by FA2 for the DNN. The central queue sends batched requests to the DNN instances using a round-robin policy [33]. The interactions between the queue and the DNN instances and the communication between different DNNs are handled by gRPC [34].

VI. Evaluation

In this section, we perform comprehensive experiments to demonstrate the effectiveness of FA2 in real-world applications with various realistic workloads. All experiments are performed based on the aforementioned system implementation.

A. Experimental Setup

We now describe the setup for our experiments, including the application and its constituting DNNs, the SLAs, the evaluation metrics, the baselines, and the workloads.

Hardware. We deploy FA2 on a testbed consisting of four servers where two have Core i9-9980x CPUs, and the other two have Core i9-10940x CPUs. Each server is equipped with an NVIDIA RTX2080 GPU. The servers can support up to 60 one-core CPU instances and 40 GPU instances, each taking 10% of the GPU share via CUDA MPS. The servers run the Ubuntu 18.04 operating system and are interconnected with a stable private Ethernet network (1Gbps). We evaluate FA2 on both the CPU and GPU.

The Application and DNNs. To compare FA2 with existing solutions in realistic environments, we use DNNs in the computer vision, natural language processing, and audio recognition domains, which are also heavily used in other DL inference serving systems [1], [2], [5], [35]. We consider an application comprised of the DNNs listed in Table II. Each instance encapsulates the DNN with pre-specified resources: 1 CPU core or 10% GPU share, determined based on the observation from Figure 2. The application contains multiple possible execution paths over the involved DNNs as specified in Table III. Note that FA2 does not assume the path for a request is known a priori—the next hop of a request is revealed only after the processing is done at a DNN. We profile

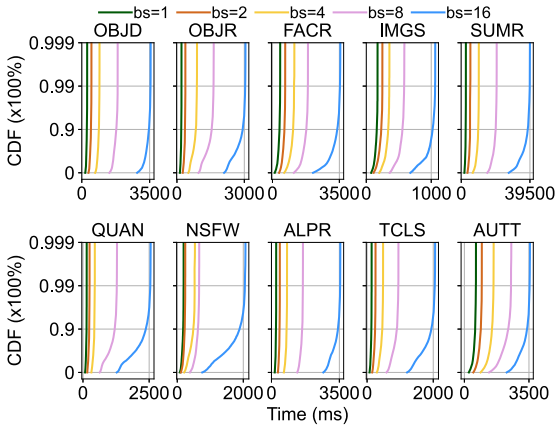

Fig. 6. Inference latency distribution for the considered DNNs under varying batch sizes on CPU (an instance equipped with one CPU core).

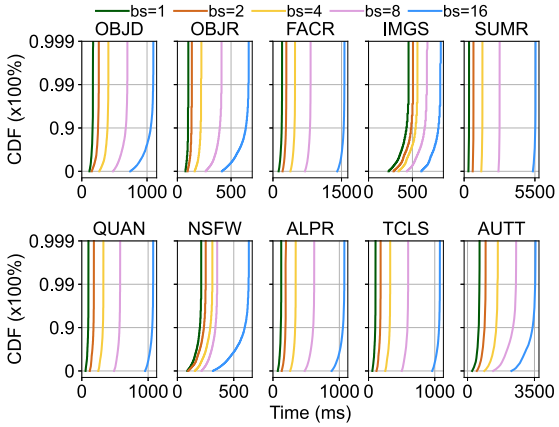

Fig. 7. Inference latency distribution for the considered DNNs under varying batch sizes on GPU (on an instance equipped with 10% of the total GPU processing units specified using CUDA MPS).

all the DNNs to obtain the throughput and latency to build the performance model for the DNN processing with varying batch sizes. Figure 6 and Figure 7 show the distribution of the inference latency over more than 250K data points on CPU and GPU, respectively. We use the data points at the 99-th percentile as the reference numbers, as also done in [1], to build the performance models following the methodology described in Section IV-A.

SLAs. SLAs are commitments between service providers and users and are typically defined based on the processing time of the involved DNNs. To set the application SLAs realistically, we calculate the tail processing time (proved to be highly predictable [3]) required by each of the execution paths under batch size $b = 1$ and multiply that time by a factor of five following a similar methodology described in [36]. This SLA setup serves as a suggestion, and FA2 can work with other SLA setups. The actual SLA depends on the application and can also be part of the service provider's pricing scheme, where different prices will be offered under different SLAs. In the execution paths in Table III, the SLAs varying from

TABLE II
DNNs Involved in the Application

Task	Abbreviation	DNN Model
Object Detection	OBJD	Inception V2
Object Recognition	OBJR	ResNet50
Not Safe For Work	NSFW	MobileNet V2
Car Recognition	ALPR	SSD
Face Recognition	FACR	ResNet50
Image Segmentation	IMGS	MobileNet V2
Question Answering	QUAN	DistilBERT
Text Summarization	SUMR	BART
Text Classification	TCLS	DistilBERT
Audio To Text	AUTT	Wav2Vec2

3020ms$-$25710ms and 960ms$-$4815ms, for the CPU and the GPU cases, respectively.

Evaluation metrics. We consider the following four types of metrics in our evaluation. (a) *Processing and queuing delay*: We use these data to demonstrate the effectiveness of FA2 in predicting the processing and queuing delay of DNNs. (b) *SLA violation ratio*: We measure the end-to-end latency of every inference request and check whether the request is processed within its SLA. (c) *Resource consumption*: We use the total number of DNN instances to denote the resource consumption where instances are homogeneous (1 core in the CPU case and 10% GPU power set with CUDA MPS). (d) *Instance utilization*: We collect the CPU utilization of each DNN instance captured in 100ms intervals.

Baselines. We compare FA2 with the recently proposed inference serving frameworks, namely GrandSLAm [1] and InferLine [2], and a state-of-the-art stream processing autoscaler called DS2 [8]. GrandSLAm maximizes the system's throughput by using dynamic batching and request reordering while guaranteeing SLAs. We consider the same resource configuration as produced by FA2 for GrandSLAm and examine its SLA violation. InferLine uses a greedy approach to find the minimum cost by choosing the most affordable configuration among different hardware and increasing the batch size for DNNs while not violating any SLAs. DS2 is an autoscaler that uses the true processing time and output rate to scale up/down stream processing operators based on the workload. DS2 does not consider request batching, so we use a fixed batch size $b = 1$. Finally, we compare FA2 with the optimal solutions generated by the Gurobi solver [37] to show how close FA2 performs to the optimal.

Workloads. To evaluate FA2 under varying workload conditions, we develop a workload generator that generates requests at different rates (following different traces) and distributes these requests to the application paths uniformly at random. Figure 8 shows the six workload trace types we use for evaluating FA2: two *steady* traces where the request rate follows a Poisson distribution (also used in [1], [35]) with an average arrival rate of around 8RPS (low) or 27RPS (high), *fluctuating* trace where the average request rate jumps between 8RPS and 27RPS, a random trace following a normal distribution with mean 15 and standard deviation 5 to have high-variance

TABLE III
ALL POSSIBLE EXECUTION PATHS IN THE APPLICATION WITH THEIR SLAS WHEN DEPLOYED ON CPUS AND GPUS

Execution Path	SLA-CPU (ms)	SLA-GPU (ms)	Description
OBJD→ALPR→QUAN	3020	1205	Provides answers to queries regarding a car in an image
OBJD→NSFW→FACR	3505	1670	Detects and recognizes a human if the image is safe for work
OBJD→OBJR→IMGS	3095	960	Detects, classifies an object, and provides segmentation of the object
SUMR→QUAN	13580	2200	Summarizes texts and provides answers in the texts
AUTT→QUAN	12245	3005	Converts audios to texts then performs question answering on them
AUTT→SUMR→TCLS	25710	4815	Performs text classification on summarized texts from audios

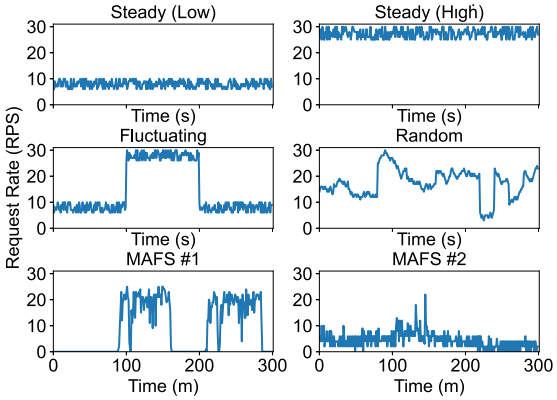

Fig. 8. Workloads used in evaluation: two steady synthetic traces following Poisson distributions, a fluctuating synthetic trace with a spike in the middle, a random workload following a normal distribution, and two realistic traces from the Azure FaaS function invocation traces.

request rates, and two different *real-world* traces where we use request arrival rates directly from the Microsoft Azure FaaS (MAFS) traces provided in [38]. The Azure traces consist of more than 46K functions, counting invocation counts per minute per function. The invocation counts vary from 0 to over 150K per minute, per function. We take two traces of function innovation counts for this experiment, which our experimental environment can handle with the maximum utilization and have different request arrival patterns.

B. End-to-End Performance

In this part, we evaluate FA2 and compare its performance with the baselines under different workloads.

1) Steady Workloads: Under steady workloads, the number of DNN instances remains constant, and the system is stable. We notice that the SLA violation rate is less than 1% for all the approaches under such a situation. However, FA2 needs less resources to serve the requests compared with the baselines (discussed in Section VI-C), which can be explained by the CPU utilization statistics. As illustrated in Figure 9, with GrandSLAm and InferLine, the DNN instances need to wait for request arrivals to create the maximum possible batch at each DNN, resulting in considerable CPU idle time, leading to resource waste. DS2 does not leverage the batching technique, which also leads to CPU under-utilization. With generally lower CPU utilization, the baseline approaches require more computing resources than what is needed (shown as *Optimal* in the figure). FA2 overcomes these issues by making holistic scaling decisions where the number of instances and the batch size for each DNN are determined jointly.

2) Fluctuating and Random Workloads: When the request rate fluctuates, the system needs to adapt by providing a new set of configurations for all the DNNs. To this end, FA2 collects metrics in 10-second intervals and decides new configurations for the next interval.

The adaptation interval of FA2 cannot be further reduced due to the following system overheads: (a) FA2 optimizer needs around 500ms to make a new adaptation decision, (b) the system needs up to 6s to bring up new instances (Kubernetes pod cold-start time) in case of scaling out, and (c) the runtime system needs a few seconds to drain the queues and stabilize the system. One possible solution to this limitation is to use a workload predictor to make autoscaling decisions proactively. To have a fair comparison, we set the same adaptation interval for the baselines. Moreover, we use the same scaling decisions produced by FA2 in GrandSLAm as their approach does not decide the number of instances for each DNN.

Figure 10 shows the number of instances required for the fluctuating (left) and random (right) workloads in each approach and the corresponding SLA violation rate. When a change in the arrival rate is detected (after around three seconds, as shown in Figure 10), the framework provides the new configuration to the Kubernetes cluster, which brings up/down containers for the concerned DNNs. Before the instances can start serving requests, a few seconds delay is observed as the instances need to be initiated. After that, the framework starts to serve the upcoming requests, and the system becomes stable again gradually. This procedure takes longer for the baseline approaches as the number of instances to add to the system is higher than that is in FA2 and the management overhead (of Kubernetes) slows down the scaling-out process of the instances due to performance interference caused by that the instances are co-located on the same physical machine, resulting in a higher SLA violation rate (up to 4% more). Overall, the results from the fluctuating and random workloads are consistent, except that a higher SLA violation rate is observed due to the higher workload variation.

3) Real-World Workload: We follow the same procedure described above and use five hours of two of the real-world FaaS traces provided by Microsoft Azure. Figure 11 depicts the required number of instances and the SLA violation rate of all the approaches. As the number of queries suddenly increases or decreases, the framework captures the changes within a few seconds and adapts the system accordingly.

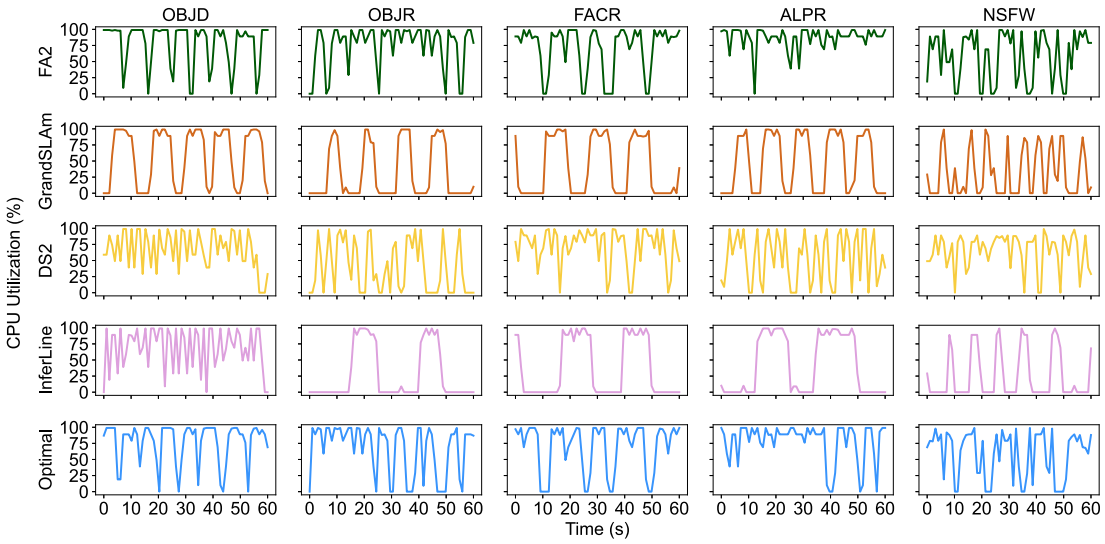

Fig. 9. CPU utilization of five DNN instances for all approaches under the steady workload at a given time-window.

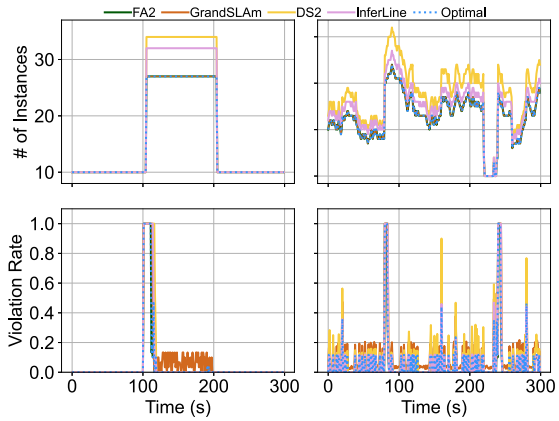

Fig. 10. Resource usage and SLA violation over time under fluctuating (left) and random (right) workloads.

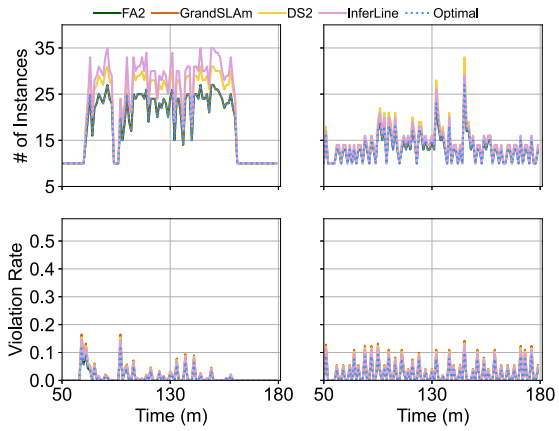

Fig. 11. Resource usage and SLA violation over time under two (left and right) Azure FaaS workload traces.

Similar to the case with the fluctuating and random workloads, FA2 needs up to 14.6% less computational resources when compared with DS2 and InferLine while reducing the SLA violation by 4.8% compared with DS2 and GrandSLAm.

C. Resource Efficiency

We assess the resource efficiency of FA2 compared with the baselines under a steady workload with varying request rates, i.e., from 6 to 60. Figure 12 depicts the required number of instances for each approach under each request arrival rate. InferLine tries to reduce the number of instances by maximizing local batch sizes one by one until there is no room to increase the batch size of any DNN. This approach needs on average 19% (and up to 48%) more computational resources when compared with FA2 as the InferLine approach does not consider coordinating the scaling decisions of the DNNs on the same execution path. DS2 does not leverage batching, leading to overall low resource utilization. Consequently, the DS2 approach requires on average 26% (and up to 62%) more computational resources when compared with FA2. Also, FA2 performs close to the optimal (produced by Gurobi solver) where the match to optimal decisions is over 96.8%. This proves that FA2 is effective in improving the resource efficiency of DL inference serving systems.

D. Impact of Optimized Batch Sizes

We evaluate FA2 under static workloads and fixed computational resources and compare it with the baselines and the optimal solution produced by Gurobi. We feed the workload to the Gurobi solver to obtain the optimal number of instances for each workload and apply the results to all the approaches, including FA2. We also consider that a request is dropped if its latency has exceeded $3\times$ the SLA to avoid constant queue overflow. We run the experiment for each approach for 1200 seconds and collect the results when the system is stable.

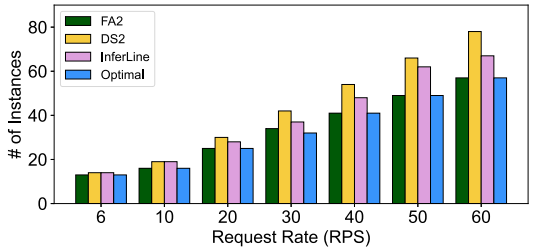

Fig. 12. Comparison of resource consumption under varying request rates on CPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum.

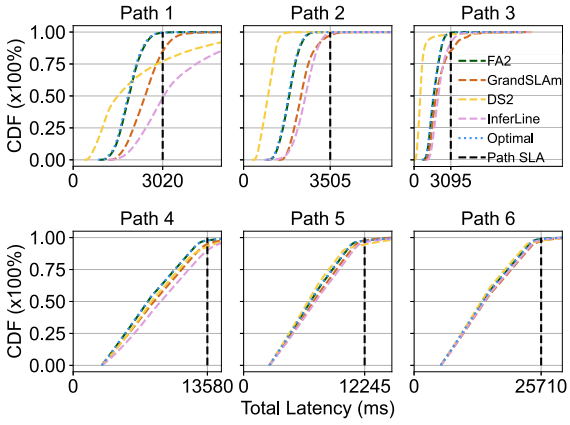

Fig. 13. End-to-end latency distribution for the three execution paths in the application under different scaling approaches on CPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations less than 1%.

Figure 13 shows the CDF of the end-to-end latency of all the approaches under fixed resources and static workload with more than 300K queries over time. DS2 does not consider request batching, leading to the least number of violations (less than 17% among the other baseline approaches. However, due to the lack of resources, over 32% of the requests are dropped. GrandSLAm adopts the "split-and-conquer" approach with a static strategy for SLA partitioning, resulting in poor adaptivity to the changes in workload distribution over the execution paths. InferLine tries to maximize the system's throughput via local optimization, where it iterates over the DNNs to increase the batch size of each DNN until there is no possibility for further batch size increases. While resulting in numerous SLA violations, the request dropout rate is the minimum among all the baselines. FA2 overcomes all these issues by considering the current workload and dynamic SLA allocations to all DNNs in a holistic manner. Moreover, FA2 servers all the requests (0% dropout) with an SLA violation rate as low as 1.4%, matching the optimal solution.

E. Performance of FA2 on GPUs

This section evaluates the performance of FA2 under GPU resources and different workloads and compares its performance with the baseline approaches. We use CUDA Multi-Process Service (MPS) [39] to share GPU processing power

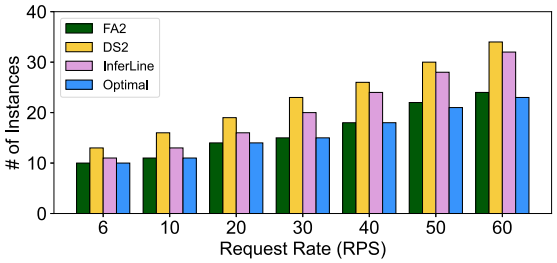

Fig. 14. Comparison of resource consumption under varying request rates on GPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum.

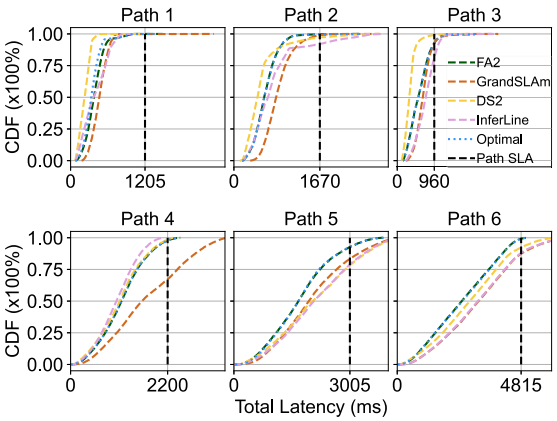

Fig. 15. End-to-end latency distribution for the six execution paths in the application under different scaling approaches on GPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations around 1%.

across multiple instances, as also used in [40] recently. Each instance receives a subset of the available processing units on the GPU enforced by CUDA MPS. We allocate 10% of the GPU processing units (similar to [40]) and memory to each instance. We show how FA2 saves GPU resources and achieves a better SLA guarantee when compared with the baseline approaches.

Figure 14 shows the average required number of instances for all the DNNs under the given workload with GPU. FA2 saves on average 25% (up to 50%) and 31% (up to 54%) GPU resources compared to InferLine, and DS2 respectively. The results differ from the CPU evaluations as the GPU instance has much better performance (higher throughput, lower latency) than the CPU instance, thus being more sensitive to the SLA division policy. FA2 achieves near-optimal SLA division, but the heuristics-based baselines are poor in this respect, thus amplifying the inefficiencies of these baselines.

Figure 15 illustrates the CDF of the end-to-end latency in all approaches under static GPU resource allocations and steady workloads with more than 12K queries per approach. In static GPU resource allocations, the number of instances for each DNN remains the same during the experiment, but the batch size is different for different approaches. The results confirm that FA2 can be effectively applied on different computing platforms without obvious performance deviations.

VII. Related Work

This section discusses resource autoscaling in four related research areas: (a) inference serving, (b) microservices, (c) stream processing, and (d) serverless computing. We note that there are autoscaling studies in batch processing systems (e.g., MapReduce) [41], [42]. However, these systems differ significantly from inference serving systems in that the tasks are transient, and the task dependency is simple and deterministic (e.g., two stages as in MapReduce) in such systems.

Autoscaling for inference serving. InferLine [2] enables autoscaling by starting from a feasible system configuration and then optimizing the configuration by adapting the hardware and batch size for each of the DNNs. InferLine does not account for the conditional execution, and the heuristic approach in choosing the configuration for each DNN leads to considerable resource underutilization. Clockwork [3] focuses on inference serving with SLA guarantees without considering DNN dependency and conditional execution. Nexus [17] is addressing a similar autoscaling problem. However, Nexus focuses mainly on (a) simple tree-like dataflow graph while FA2 does not assume the graph structure and can work with arbitrary graphs, (b) applications specified with a single SLA while FA2 targets consolidated applications with different SLAs, (c) GPU-level resource allocation, while FA2 allows fine-grained GPU resource allocation via CUDA MPS, and (d) allocating half of the application SLAs to queuing delays leading to reduced GPUs utilization, while FA2 carefully allocates the exact amount of time for each DNN's queue.

Microservice autoscaling. Existing microservice autoscaling mechanisms are mostly rule-based heuristics [15], [43]–[47] or meta-heuristics [48]–[50]. For microservices, request processing typically follows a deterministic dataflow-graph, instead of one with conditional execution paths as in inference serving systems. Machine learning methods are also employed for workload prediction in microservice systems for better autoscaling [51], [52]. Recently, DAGOR [14] provided overloading detection and collaborative load shedding in microservice systems. ATOM [13] is a model-driven microservice autoscaler based on layered queuing networks [53], but it does not consider request batching and end-to-end SLA guarantees. GrandSLAm is a microservice management framework focusing on improving throughput while guaranteeing application SLAs [1]. However, GrandSLAm does not deal with the autoscaling issue.

Autoscaling for stream processing. A large body of work has been dedicated to scaling operators in stream processing systems [8]–[10], [21], [22], [54]–[62]. Most of them rely on coarse-grained metrics such as the CPU/memory utilization, system throughput, queue size, and/or back-pressure and apply threshold-based policies for scaling in/out operators. Some alternative solutions such as DRS [59] or Nephele SPE [56] adopt queuing theory models to characterize the system. DS2 focuses on estimating the true processing and output rates of individual dataflow operators and figures out the scaling decisions for all the operators in one shot [8]. Scaling DL inference differs from scaling stream processing operators in that a DL application may contain multiple (conditional) execution paths, each with a specific SLA requirement [63]. While there exist stream processing systems that provide SLA guarantees [10], [56], none of them consider conditional execution while guaranteeing a set of SLAs over multiple execution paths. Therefore, none of the existing autoscaling solutions for stream processing systems can be directly applied to elastic DL inference serving.

Autoscaling for serverless computing. Serverless computing is a widely adopted paradigm to provide autoscaling in cloud environments [18]. There exist numerous serverless platforms in academia (e.g., [11], [18], [64]–[66]), industry (e.g., [67]–[69]) as well as open-source solutions (e.g., [70]–[72]). Largely, autoscaling is achieved using three approaches: (a) request-based, (b) concurrency value-based, and (c) metric-based [12]. Most industry providers use a request-based scaling approach in which the cloud resources are scaled up when there are more requests for executing functions, while the resources are scaled-down otherwise. However, this approach does not fulfill SLAs. The second approach executes the function concurrently on a given number of instances, and when this value is reached, the resources are scaled down [66], [73]. Finally, most of the open-source platforms such as OpenFaaS [70] and Kubeless [72] use a metric-based scaling approach. This approach aims to maintain metrics such as latency, throughput, and CPU usage within a predefined threshold. However, this approach has the worst delay in adapting to fluctuating workloads. Overall, these solutions lack data dependency and conditional execution support, as typically seen in inference serving systems.

VIII. Conclusion

In this paper, we presented FA2, a fast, accurate resource autoscaler tailored for efficiently provisioning DL inference serving while guaranteeing service-level agreements on the end-to-end latency. FA2 leverages a graph-based model to capture the resource scaling problem and makes resource scaling decisions for all the DNNs in a holistic manner. Evaluation results based on an actual system prototype and real-world workload traces show that FA2 improves the overall resource utilization significantly compared with the state-of-the-art resource scaling solutions. Overall, FA2 is able to match the optimal theoretical decisions almost always. Although we only tested with CPUs and GPUs, our design of FA2 is generally applicable to other mixed setups. In the future, we plan to extend FA2 to more heterogeneous environments (including CPUs, GPUs, TPUs, and other accelerators) by introducing decision variables for hardware type for each DNN.

Acknowledgment

We would like to thank the anonymous reviewers and our anonymous shepherd for their valuable comments and suggestions. This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI.

REFERENCES

[1] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks," in *ACM European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–16.

[2] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in *ACM Symposium on Cloud Computing (SoCC)*, 2020, pp. 477–491.

[3] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 443–462.

[4] F. Ahmad, H. Qiu, R. Eells, F. Bai, and R. Govindan, "CarMap: Fast 3D feature map updates for automobiles," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 1063–1081.

[5] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *ACM Symposium on Cloud Computing (SoCC)*, 2021, pp. 1–17.

[6] Y. Choi, Y. Kim, and M. Rhu, "Lazy batching: An sla-aware batching system for cloud machine learning inference," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 493–506.

[7] V. Nigade, L. Wang, and H. Bal, "Clownfish: Edge and cloud symbiosis for video stream analytics," in *IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 55–69.

[8] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 783–798.

[9] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Very Large Data Bases (PVLDB)*, vol. 10, no. 12, pp. 1825–1836, 2017.

[10] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 1–13.

[11] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Very Large Data Bases (PVLDB)*, vol. 13, no. 12, pp. 2438–2452, Jul. 2020.

[12] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, pp. 1–15, 2020.

[13] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1994–2004.

[14] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 149–161.

[15] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "Smartvm: a sla-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[16] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[17] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 322–337.

[18] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *ACM Symposium on Cloud Computing (SoCC)*, 2017, pp. 445–451.

[19] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 613–627.

[20] L. Liu, H. Li, and M. Gruteser, "Edge Assisted Real-time Object Detection for Mobile Augmented Reality," in *ACM Annual International Conference On Mobile Computing And Networking (MobiCom)*, 2019, pp. 1–16.

[21] B. Gedik, S. Schneider, M. Hirzel, and K. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[22] V. Gulisano, R. Jiménez-Peris, M. Patiño-Mart´(i) nez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[23] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 929–946.

[24] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *International Conference on Autonomic Computing (ICAC)*, 2014, pp. 57–64.

[25] J. L. Hellerstein, Y. Diao, S. S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Wiley, 2004.

[26] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[27] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2019, pp. 25–32.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[29] K. Razavi and L. Wang, "Fa2 source code," accessed on 3.3.2022. [Online]. Available: https://github.com/vunetsys/FA2

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[31] "Docker Inc," https://www.docker.com, accessed on 29.10.2021.

[32] T. L. Foundation, "Kubernetes," https://kubernetes.io, 2019, accessed on 29.10.2021.

[33] T. Brisco, "Rfc1794: Dns support for load balancing," 1995.

[34] "gRPC," https://grpc.io, accessed on 29.10.2021.

[35] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *USENIX Annual Technical Conference (ATC)*, 2021, pp. 397–411.

[36] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 109–120.

[37] "Gurobi," https://www.gurobi.com, accessed on 29.10.2021.

[38] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX Annual Technical Conference (ATC)*, 2020, pp. 205–218.

[39] N. Corporation, "Cuda mps," https://docs.nvidia.com/deploy/mps/index.html, 2021, accessed on 29.10.2021.

[40] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "GSLICE: Controlled spatial sharing of GPUs for a scalable inference platform," in *ACM Symposium on Cloud Computing (SoCC)*, 2020, pp. 492–506.

[41] X. Zeng, S. Garg, Z. Wen, P. Strazdins, L. Wang, and R. Ranjan, "Sla-aware scheduling of map-reduce applications on public clouds," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2016, pp. 655–662.

[42] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for mapreduce in a cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1265–1279, 2014.

[43] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018, pp. 157–167.

[44] L. Florio and E. D. Nitto, "Gru: An approach to introduce decentralized autonomic behavior in microservices architectures," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2016.

[45] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017.

[46] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: autoscaling and monitoring as a service,"

in *ACM Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2017.

[47] A. Sriraman and T. F. Wenisch, "μTune: Auto-tuned threading for OLDI microservices," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[48] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 618–632, 2017.

[49] S. Frey, F. Fittkau, and W. Hasselbring, "Search-based genetic optimization for deployment and reconfiguration of software in the cloud," in *IEEE International Conference on Software Engineering (ICSE)*, 2013.

[50] C. Verbowski, E. Thayer, P. Costa, H. Leather, and B. Franke, "Right-sizing server capacity headroom for global online services," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 645–659.

[51] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things Journal*, vol. preprint, pp. 1–1, 2020.

[52] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in *IEEE International Conference on Big Data (BigData)*, 2017, pp. 2433–2441.

[53] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Trans. Software Eng.*, vol. 21, no. 8, pp. 689–700, 1995.

[54] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[55] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[56] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2015, pp. 399–410.

[57] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[58] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[66] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: a case of knative," in *International Workshop on Serverless Computing*, 2019, pp. 13–18.

[59] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: auto-scaling for real-time stream analytics," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3338–3352, 2017.

[60] G. Mencagli, P. Dazzi, and N. Tonci, "Spinstreams: a static optimization tool for data stream processing applications," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 66–79.

[61] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao, "Chi: A scalable and programmable control plane for distributed stream processing systems," *Very Large Data Bases (PVLDB)*, vol. 11, no. 10, pp. 1303–1316, 2018.

[62] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Very Large Data Bases (PVLDB)*, vol. 12, no. 9, pp. 1002–1015, 2019.

[63] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 19–33.

[64] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[65] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 199–208.

[67] A. W. Services, "Aws lambda," https://aws.amazon.com/lambda, 2019, accessed on 29.10.2021.

[68] Google, "Google cloud functions," https://cloud.google.com/functions, 2019, accessed on 29.10.2021.

[69] Microsoft, "Microsoft azure functions," https://azure.microsoft.com/en-us/services/functions, 2019, accessed on 29.10.2021.

[70] A. Ellis, "OpenFaaS," https://www.openfaas.com, 2019, accessed on 29.10.2021.

[71] IBM, "Ibm openwhisk," https://developer.ibm.com/open/projects/openwhisk, 2019, accessed on 29.10.2021.

[72] Kubeless, "Kubeless," https://kubeless.io, 2019, accessed on 29.10.2021.

[73] Google, "Cloudrun," https://cloud.google.com/run, 2021, accessed on 29.10.2021.