SMIless: Serving DAG-based Inference with Dynamic Invocations under Serverless Computing

Chengzhi Lu[1,2,3], Huanle Xu[3,*], Yudan Li[3], Wenyan Chen[3], Kejiang Ye[1], Chengzhong Xu[3,4,*]
[1]*Shenzhen Institute of Advanced Technology, CAS*
[2]*Univeristy of Chinese Academy of Sciences*
[3]*Dept. of Computer and Information Science, University of Macau*
[4]*State Key Lab of Internet of Things for Smart City, University of Macau*
{*cz.lu,kj.ye*}@siat.ac.cn, {*huanlexu,mc25699,yc17498,czxu*}@um.edu.mo

***Abstract*—The deployment of ML serving applications, featuring multiple inference functions on serverless platforms, has gained substantial popularity, leading to numerous developments of new systems. However, these systems often focus on optimizing resource provisioning and cold start management separately, ultimately resulting in higher monetary costs.**

This paper introduces SMIless, a highly efficient serverless system tailored for serving DAG-based ML inference in heterogeneous environments. SMIless effectively co-optimizes resource configuration and cold-start management in the context of dynamic invocations. This is achieved by seamlessly integrating adaptive pre-warming windows, striking an effective balance between performance and cost. We have implemented SMIless on top of OpenFaaS and conducted extensive evaluations using real-world ML serving applications. The experimental results demonstrate that SMIless can achieve up to a 5.73× reduction in the overall costs while meeting the SLA requirements for all user requests, surpassing the performance of state-of-the-art solutions.

***Index Terms*—Serverless, DAG-based Inference, Dynamic Invocation**

I. Introduction

Serverless computing has emerged as a highly promising computing paradigm in the modern cloud-native era, distinguished by its fine-grained compute elasticity, pay-per-use model, remarkable scalability, and the convenience it offers for development and maintenance [1], [2]. The inherent advantages and flexibility of serverless platforms have catalyzed numerous recent initiatives to mitigate machine learning (ML) inference applications to this architecture [3], [4], [5], [6], [7].

In ML serving, it is often crucial to provide comprehensive services by incorporating multiple ML inference models within a single application [8]. Consider, for instance, a conversational Artificial Intelligence pipeline composed of three modules: an automatic speech recognition module that converts input audio waveforms into text, a large language model (LLM) module that comprehends the input and generates a relevant response, and a text-to-speech module responsible for rendering the LLM's output into speech [9]. The various models are typically organized in parallel, sequentially, or a combination of both, and can be represented as Directed Acyclic Graphs (DAGs) [5], [10], [11], [12], [13].

*Corresponding authors

ML inference applications often experience widely varying arrival patterns of service requests [14], [10]. Consequently, traditional server-centric deployments are prone to resource overprovisioning problems. Leveraging serverless computing, applications can precisely tailor their resource utilization by dynamically launching different functions, typically triggered at or near the time of receiving an invocation request, depending on the pre-warming and keep-alive mechanisms implemented to mitigate cold starts [15], [16].

With the shift towards serverless computing, the landscape of underlying hardware resources for serving ML inference in production clusters is undergoing increased heterogeneity [17], [10], [18]. This includes a wide variety of accelerators such as GPUs, TPUs, and FPGAs, all tailored to enhance the performance of ML applications [18], [19], [20]. While high-end accelerators can effectively mitigate the inference overhead, they also come with a significant cost burden. For instance, incorporating a V100 GPU can yield up to a 10× improvement in inference latency for ResNet50 compared to a CPU with four cores. However, it is worth noting that the GPU's unit cost is 16× higher than that of the CPU in AWS cloud [18]. Consequently, this heterogeneity presents a design trade-off to balance between performance and monetary cost.

Crafting efficient resource provisioning policies for deploying ML applications with DAG topologies on a serverless platform that harnesses heterogeneous hardware poses significant challenges. On one hand, the resource configuration of one function influences the cold-start management of all succeeding functions within a DAG application, creating a cascading effect. Consequently, it becomes essential to design resource configuration and cold-start management policies of all functions in tandem to achieve optimal performance. On the other hand, the highly dynamic nature of invocation patterns means that a policy suitable for one invocation may not be suitable for another, necessitating global optimization that takes into account future invocations, thereby adding layers of complexity to the task.

Traditional literature on resource allocation for DAG-based applications in heterogeneous environments neglects to address cold-start issues, resulting in resource wastage and a failure to meet SLA requirements for end-to-end (E2E) latency [21], [22], [23]. Conversely, current research focus-

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

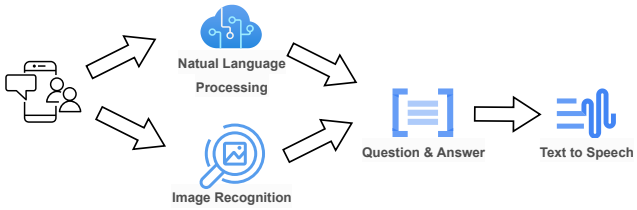


Fig. 1: The processing flow of an IPA application [5].

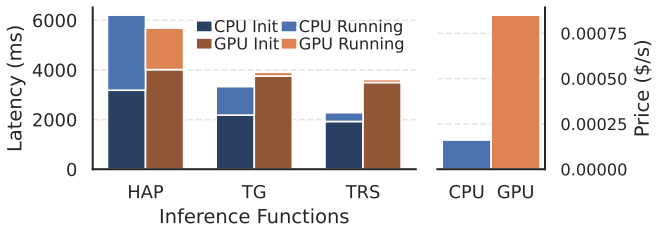


Fig. 2: Inference latency under different hardware. The price comparison is made with reference to a CPU server with 16 cores and a GPU server equipped with one V100 in AWS [40].

ing on ML serving applications using serverless computing predominantly emphasizes isolated optimizations [4], [24], [25], [26], [27], or completely disregards the DAG structure [17]. Specifically, existing works tend to handle cold-starts and resource provisioning as separate entities, falling short in managing dynamic invocation arrivals and resulting in significantly elevated monetary costs. Furthermore, approaches that entirely neglect DAG structures risk substantial reductions in the overall resource utilization.

This paper presents SMIless, a new system for Serving DAG-based ML Inference applications using serverless computing across heterogeneous resources. The distinctive aspect of SMIless lies in its design of adaptive pre-warming windows, which are dynamically updated based on both the function's position within the DAG topology and the invocation patterns. Moreover, SMIless tackles the cascading effect by reformulating a manageable path search problem. During the path search process within a multi-way tree, each node traversed represents a critical decision for heterogeneous resource configuration and the cold-start mitigation policy of all functions. To expedite the search for paths that meet the minimum cost and end with the node satisfying the SLA requirement, SMIless utilizes an efficient ordering mechanism to minimize the search overhead. Addressing the difficulty posed by extensive search space, SMIless employs a decomposition approach where the complex DAG is divided into multiple simple paths and each path is optimized in parallel. Furthermore, SMIless implements dynamic batching and effective scaling to handle burst workloads resulting from multiple invocations arriving within a short timeframe.

We have implemented SMIless using the popular framework OpenFaaS [28]. The SMIless system comprises a profiler that effectively measures the inference and initialization times of individual functions. Additionally, SMIless incorporates new online predictors to accurately forecast request arrival patterns. The implementation also supports GPU multiplexing, enabling a function instance to utilize only a fraction of the available GPU. We evaluate SMIless using real-world ML applications [5] and workloads similar to those employed in Azure Functions [29]. The experimental results substantiate that SMIless can achieve a remarkable up to 5.73× reduction in overall costs without violating the SLA requirement for E2E latency when compared to existing solutions. To summarize, we have made the following contributions in this paper:

- We have recognized the significance of effectively managing dynamic invocation arrivals in co-optimizing heterogeneous resource configuration and cold-start management.
- We have designed an adaptive pre-warming policy tailored to the DAG topology of ML serving applications.
- We have developed a scalable system that efficiently manages ML serving with serverless computing. This system makes optimal use of heterogeneous resources.

II. Background and Motivation

A. ML Serving under Serverless Platform

In production-level ML serving applications, inference is typically performed on multiple ML models to provide comprehensive services [9], [30], [10], [31]. As an example, Fig. 1 demonstrates a popular intelligent personal assistant (IPA) application designed to answer questions about given images [5]. When a user initiates a request to the IPA application, it triggers several invocations corresponding to distinct inference stages, including Natural Language Processing (NLP) [32], [33], Image Recognition (IR) [34], [35], Question and Answer [36], and Text to Speech [37].

In a serverless platform, the entire application is divided into distinct modules, with each function dedicated to serving a specific model [38], [39]. Consequently, the interactions between these functions can be represented as a DAG [10].

B. Heterogeneous Serverless Computing

In production environments, the utilization of high-end and low-end hardware resources for serverless functions involves distinct design trade-offs that must be carefully considered.

As shown in Fig. 2, there is a significant disparity in the inference latency of three commonly used ML models, namely Human Activity Pose (HAP), Text Generation (TG), and Translation (TRS), when executed on a 16-core CPU compared to a V100 GPU. Simultaneously, there exists a substantial price difference between these two hardware options. In particular, the inference latency of the TRS model with a *warm start* reduces by approximately 10× when running on a GPU, while the unit price of the GPU is only 8× higher than that of the CPU. Consequently, executing the TRS model on a GPU is more cost-effective. However, during the initial phase of serving when the function initializes to handle the first request, known as a *cold start*, the inference latency of the TRS model on a V100 GPU surpasses that of a CPU due to the associated high initializing overhead, rendering the GPU devoid of advantages.

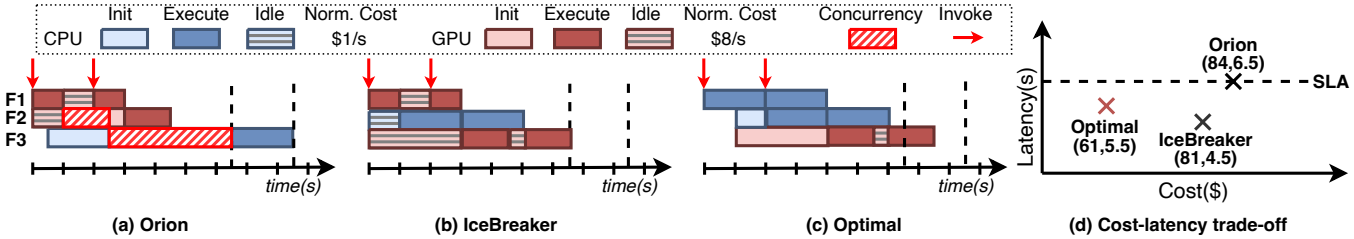


Fig. 3: An example illustrating the limitations of existing approaches where the use of different colors indicates that each function is executed on separate hardware, while a concurrency indicates overlapping execution of two identical functions. (a) Orion. (b) IceBreaker. (c) Optimal solution. (d) The E2E performance and the execution cost for each solution.

C. Resource Management with Dynamic Invocations

Balancing the optimal trade-off between E2E latency and cost for serving ML applications with a DAG under heterogeneous serverless computing presents significant challenges, necessitating global co-optimization across all functions.

1) Challenges: **The primary challenge stems from the cascading effect triggered by the resource configuration of one function on the cold-start management of succeeding functions**. The selection of a heterogeneous configuration directly shapes inference latency, thereby exerting a substantial influence on the overlap between function execution and initialization. Notably, greater inference latency for a function amplifies the opportunity for a broader overlap window, mitigating prolonged keep-alive periods and consequently cutting down on costs. However, the extent of this overlap is also contingent upon the initialization time of the succeeding function, which, in itself, is closely intertwined with its resource configuration, further impacting the subsequent functions in line. Thus, the configuration of each function sets off a cascading effect on succeeding functions within the DAG.

The second challenge arises from the dynamic nature of invocation patterns within inference applications. In scenarios with a high invocation arrival rate, the resource configuration and cold-start management for a specific invocation may not be optimal for subsequent invocations in the long run.

2) Limitation of Existing Works: Current resource management schemes for serverless computing are inadequate in addressing the aforementioned challenges. To illustrate this, we present a straightforward example in Fig. 3 where an ML application, consisting of three functions in a pipeline.

• *Orion* [4]: This method selects the resource configuration under the assumption of right pre-warming, where the execution of each function perfectly overlaps with the initialization of the succeeding function. While this assumption can greatly simplify co-optimization, it holds true only when the inter-arrival time between different invocations is substantial, resulting in suboptimal outcomes when multiple invocations arrive within a short period. As depicted in Fig. 3(a), in line with the cost-minimizing result and SLA requirement of 6.5 seconds for E2E latency, Orion assigns the execution of F_1 and F_2 to GPU, while F_3 is executed on CPU. However, upon the arrival of the second invocation, Orion needs to launch additional instances for both F_2 and F_3 to prevent SLA violation. An optimal approach, as shown in Fig. 3(c), is to launch F_2 on the CPU with pre-warming and F_3 on the GPU with right keep-alive, considering the second invocation. This leads to a 37.7% decrease in the overall cost.

• *IceBreaker* [17]. This approach individually manages the resource configuration and cold-start policy for each function, struggling to effectively leverage the DAG topology for co-optimization. As shown in Fig. 3(b), under IceBreaker, F_2 is warmed up on the CPU owing to its high efficiency-to-cost ratio. Simultaneously, leveraging the GPU's superior speedup, both F_2 and F_3 are warmed up on the GPU. However, due to the lack of consideration for the DAG structure, IceBreaker cannot leverage right pre-warming, resulting in a total cost that is 33% higher than the optimal solution.

III. SMIless Architecture

To tackle the challenges outlined in § II-C1, we present SMIless, a serverless framework for optimizing the execution of ML serving applications with highly dynamic invocations.

A. Key Design Ideas

I_1 : **Leveraging adaptive pre-warming to manage dynamic invocations.** SMIless introduces an innovative adaptive cold-start policy for functions within a DAG application. This policy facilitates the calculation of the pre-warming size of all serverless functions based on a given resource configuration and invocation arrival rate. Such capability empowers SMIless to effectively formulate the E2E delay and overall costs of a complex DAG application.

I_2 : **Addressing the cascading effect through path search formulation.** SMIless initiates its optimization approach by redefining the problem as a path search in a multi-way tree. Each distinct combination of the adaptive cold start strategy and resource allocation policy for all functions within the DAG is treated as a unique node, with alterations in these combinations represented as edges. SMIless systematically navigates this tree, prioritizing combinations that result in minimized costs. This strategic exploration enables SMIless to effectively address the cascading effect, approaching the optimal solution without the need for exhaustive exploration.

B. System Overview

As shown in Fig. 4, SMIless is composed of three main modules: the **Optimizer Engine,** the **Offline Profiler**, and the

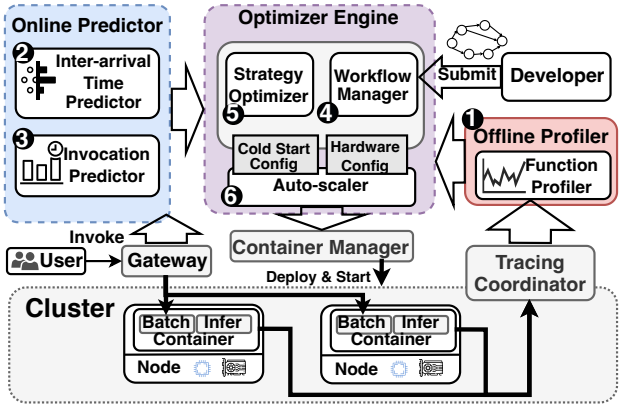


Fig. 4: System architecture of SMIless.

Online Predictor. Upon a developer's submission of a ML application to the serverless platform, SMIless generates the execution strategy through the **Optimizer Engine**, leveraging profiles of each function produced by the **Offline Profiler**, and incorporating predicted invocation patterns from the **Online Predictor**. The execution strategy is then deployed by the Container Manager.

The **Offline Profiler** ❶ operates in the background, systematically gathering execution and initialization times from diverse configurations of deployed functions. Utilizing these traces, the Offline Profiler constructs a tailored model for profiling the inference time and the initialization time.

The **Online Predictor** consists of two components: the Inter-arrival Time Predictor ❷ and the Invocation Predictor ❸. The **Online Predictor** collects the information regarding the application's invocation pattern and then uses the Inter-arrival Time Predictor to construct a model specific to each application to predict the inter-arrival time, which aids in managing cold-start scenarios and hardware configuration. The Invocation Predictor predicts a limit for the potential number of invocations within each time window, enabling the Auto-scaler to effectively handle burst workloads.

Once receiving ML requests from the developer, SMIless uses the **Optimizer Engine** to produce the optimal initialization and execution strategies through three key modules: the Workflow Manager ❹, Strategy Optimizer ❺, and Auto-scaler ❻. The Workflow Manager breaks down the DAG into multiple simplified subgraphs. After fetching the optimized strategy of each subgraph from the Strategy Optimizer, the Workflow Manager intelligently combines these strategies to minimize the overall cost. Subsequently, the Auto-scaler operates dynamically to control the scaling of all inference functions based on the predicted invocation number and the function's hardware configuration.

IV. Offline Profiling and Online Prediction

A. Function Profiling

The process of function execution consists of two stages: the *initialization* stage and the *inference* stage. To capture the initialization and inference times of each function instance, SMIless utilizes the event tracking method, allowing the system to accurately record the time taken for each stage. These timing records are stored using Prometheus [41], a widely used open-source tracing coordinator. In addition to timing information, Prometheus also stores details about the hardware configuration and setting of batch size.

1) Profiling initialization time: The initialization process of the inference function encompasses data accessing, model loading, and function dependency initialization steps. In the case of starting a new instance of an inference function, SMIless first accesses the container image of the inference model from the remote repository. Subsequently, it initializes a container on the appropriate host. Although the size of the container image remains constant, the initialization time can still fluctuate due to shared resources contention such as network bandwidth, PCIe bandwidth, and memory bandwidth. To address this issue, SMIless' Offline Profiler calculates both the average μ and the standard deviation σ among all the initialization time for each function and then utilizes $(\mu + n \times \sigma)$ as a robust measurement of the initialization time.

The initialization process on the GPU device involves initializing the CUDA context and runtime for inference frameworks (such as PyTorch and TensorFlow), allocating GPU capacity, and transferring the model from host memory to GPU memory. These additional operations result in a longer initialization time compared to that of the CPU. Consequently, the Offline Profiler conducts separate estimations for the initialization time of each inference function on both the CPU and GPU. SMIless repeats the initialization process 10 times for each function to collect a sufficient number of records.

2) Profiling inference time: The inference time of a function is highly impacted by the input batch size and the hardware configuration, which encompasses factors such as the number of CPU cores, memory size, and GPU quantity of the container [14], [42]. Additionally, during the inference process, only a small amount of memory of the container is typically required to cache the incoming data from the invocation request, in addition to the memory used for hosting the container image. Beyond a certain threshold known as the knee point [14], increasing the memory capacity does not yield significant performance improvements. Hence, SMIless ensures that the inference function is equipped with a hardware configuration that includes a memory capacity slightly above the knee point, effectively preventing memory from becoming a bottleneck. Moreover, SMIless leverages MPS (Multi-Process Service) [43] to partition a GPU into multiple portions and distribute them among several instances. In order to mitigate resource contention, such as PCIe bandwidth and GPU memory, among the instances sharing the same GPU, the minimum unit of the GPU allocation is set at 10%.

SMIless primarily focuses on adjusting the CPU core count or GPU quantity to switch between different configurations. Hence, the Offline Profiler develops learning models to profile the inference time, taking into account factors such as batch size B, the number of CPU cores (# of CPU cores), or GPU proportions (% of GPU). Due to the excellent parallelism

offered by deep learning frameworks like PyTorch and TensorFlow during model inference, the Offline Profiler effectively captures this acceleration effect based on the Amdahl's law [44]. Specifically, when it comes to CPU configurations, the learning model is expressed as follows:

$$\text{Inference time} = \lambda_c \times B \times \left(\frac{\alpha_c}{\#\text{ of CPU cores}} + \beta_c\right) + \gamma_c. \quad (1)$$

Here, α_c represents the computational volume required for the model on the CPU, while β_c accounts for the additional time needed during CPU execution, such as context switching, instruction and data load/store operations, and cache misses. Furthermore, a coefficient λ_c is introduced to capture performance degradation stemming from the architecture's use of multiple CPU cores, which may result in increased cache misses and branch prediction errors when processing batches of invocation requests. Finally, the network transmission time γ_c is incorporated into the E2E inference time. To minimize profiling overhead, SMIless restricts the maximum CPU cores to a ratio determined by the unit cost of GPU to CPU.

For the GPU backend, the learning model is expressed by:

$$\text{Inference time} = \lambda_g \times B \times \left(\frac{\alpha_g}{\%\text{ of GPU}} + \beta_g\right) + \gamma_g. \quad (2)$$

In contrast to the CPU backend, model inference on the GPU backend involves more processing steps, including data transmission between host memory and GPU memory, as well as frequent synchronization of control flow and data flow [45]. As a result, the parameters of the learning model need to be determined separately compared to their CPU counterparts. The parameters $\{\lambda_i, \alpha_i, \beta_i, \gamma_i\}_{i=c,g}$ are obtained through curve-fitting for each function.

B. Online Prediction

Upon receiving invocation requests from users, the Gateway in SMIless forwards this information to the Online Predictor for the purpose of counting the invocation number received for each application within a specified time window, which is set to one second under SMIless.

1) Predicting the invocation number: The Invocation Predictor forecasts the anticipated invocation number for each application within the next time window. To prevent underestimation and avoid SLA violations, the Invocation Predictor adopts a classification method over a regression approach, utilizing the LSTM model [24]. Specifically, the Predictor divides the prediction space into multiple buckets and determines the upper bound of the bucket as the prediction. The bucket size is established to be equal to the minimum batch size of the application's functions. To train the LSTM model, the input data consists of a time sequence that encompasses the invocation numbers within past time windows. The length of the input sequence is tailored to each individual application.

2) Predicting the inter-arrival time: The Invocation Predictor is also capable of predicting the inter-arrival time, which essentially represents the time interval between two consecutive non-zero predictions of invocation numbers. However, the classification method employed to establish an upper bound can easily lead to estimation inaccuracies in inter-arrival times. Consequently, SMIless introduces a dedicated Inter-arrival Time Predictor. This Predictor utilizes a dual-input approach, utilizing both the time series of inter-arrival time and invocation number to prevent potential overestimations that may lead to SLA violations. It incorporates two distinct LSTM modules to process these inputs separately, merging their respective hidden state outputs. This merged output then passes through an activation layer and a linear layer to predict the inter-arrival time for the subsequent invocation accurately.

V. SMIless Resource Optimization

A. Co-optimization Framework

The primary objective of the SMIless Optimizer Engine is to minimize the comprehensive cost associated with function execution, ensuring adherence to SLA requirements for the E2E latency of all user requests. This execution cost encompasses the function initialization cost, inference cost, as well as keep-alive cost. Due to the inherent heterogeneity of the underlying hardware infrastructure, each function has various options for configuring its running instance. Specifically, for function k, we represent its hardware configuration as $\star_k$ and its cold start management policy as $\triangle_k$. In this context, $\star_k \in \mathcal{C}$ and $\triangle_k \in \mathcal{S}$, where $\mathcal{C}$ denotes the set of all possible configurations and $\mathcal{S}$ represents the set of policies for cold-start management. The unit execution cost $U(\cdot)$ is solely dependent on $\star_k$, while the associated execution time $E_k(\cdot)$ is influenced by both $\star_k$ and $\triangle_k$. Consequently, the execution cost for the entire function k, denoted as $C_k(\cdot)$, can be written as:

$$C_k(\star_k, \triangle_k) = E_k(\star_k, \triangle_k) \cdot U(\star_k). \quad (3)$$

Considering that SMIless needs to manage N functions within a DAG graph representing an ML serving application, the overall optimization problem can be formulated as:

$$\min_{\{\vec{\chi}, \vec{\varphi}\}} \sum_{k=1}^{N} C_k(\star_k, \triangle_k), \quad \text{subject to, } \mathcal{L}(\vec{\chi}, \vec{\varphi}) \leq \text{SLA}. \quad (4)$$

This problem involves two optimization vectors: $\vec{\chi}$ and $\vec{\varphi}$, where $\vec{\chi} = \{\star_1, \cdots, \star_N\}$, $\vec{\varphi} = \{\triangle_1, \cdots, \triangle_N\}$. The function $L(\vec{\chi}, \vec{\varphi})$ captures the the E2E latency of the application under the configuration $\vec{\chi} \times \vec{\varphi}$. This problem constitutes a complex combinatorial optimization problem, which can be readily reduced to the Constrained Shortest Path Problem, a well-known NP-hard problem [46].

B. Adaptive Cold-Start Management

To address the optimization problem, SMIless proposes an innovative design known as "adaptive cold-start management". This design aims to enable dynamic updates of the pre-warming size of each individual function based on DAG topology and invocation patterns. Building on this policy, SMIless is able to compute the latency function $\mathcal{L}(\vec{\chi}, \vec{\varphi})$.

1) Adaptive pre-warming: To demonstrate the idea behind adaptive pre-warming, let us consider a straightforward DAG consisting of two functions, F_1 and F_2, executed sequentially in a pipeline, as shown in Fig. 5. The inter-arrival time between two successive invocations is denoted as IT, the time required

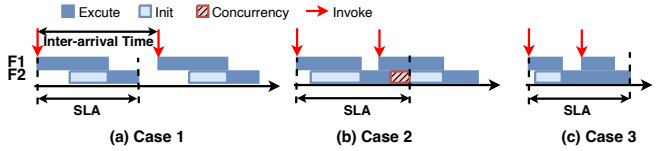


Fig. 5: The pre-warming policies vary in different settings based on the inter-arrival time between successive invocations. The functions F1 and F2 are executed in a pipeline.

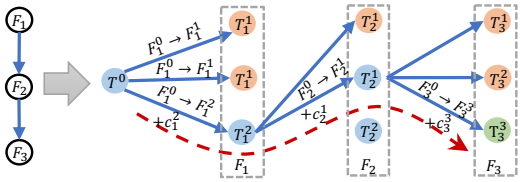


Fig. 6: The path-search process where the orange (green) node indicates the E2E latency violates (meets) SLA.

for loading F_k during initialization is denoted as T_k, and the inference time for F_k is I_k, Both of T_k and I_k depend on the hardware configuration $\star_k$.

Case I: Low invocation arrival rate. In scenarios where the arrival rate is low, there is significant flexibility to select a larger pre-warming window size, thereby minimizing the overall cost. Specifically, when $\mathsf{T}_2+\mathsf{I}_2 < \mathsf{IT}$, the pre-warming size for F_2 under SMIless is determined as $(\mathsf{IT}-\mathsf{T}_2-\mathsf{I}_2)$, and the corresponding warming-up process begins I_2 units of time earlier before the completion of the successor F_1, as depicted in Fig. 5(a). In other words, after the inference process of an invocation is completed on F_2, the function will be unloaded and remain idle for a duration of $(\mathsf{IT}-\mathsf{T}_2-\mathsf{I}_2)$ before being loaded again to process the next invocation. As a result, the initialization overhead of F_2 is fully overlapped with the inference process of F_1. In this scenario, the following relationship holds:

$$\mathcal{L}(\vec{\chi},\vec{\varphi}) = \mathsf{I}_2+\mathsf{I}_1, \text{ and } C_2 = (\mathsf{T}_2+\mathsf{I}_2)\cdot U(\star_2). \quad (5)$$

By selecting an appropriate hardware configuration for all functions such that $\mathcal{L}(\vec{\chi},\vec{\varphi})$ remains within the SLA, this dynamic pre-warming approach fully mitigates the negative impact of long initialization time and ensures optimal cost efficiency, as demonstrated in the subsequent theorem.

Theorem 5.1: When $\mathsf{I}_2+\mathsf{I}_1 < \text{SLA}$ and $\mathsf{T}_2+\mathsf{I}_2 < \mathsf{IT}$, the warming-up policy of SMIless guarantees the minimum overall execution cost.

The theorem discussed above also implies that when faced with strict SLA requirements, SMIless should prioritize a more advanced hardware configuration for F_2. This selection will result in a smaller inference time I_2 and subsequently a larger pre-warming window size.

Case II: High invocation arrival rate. In cases where the arrival rate is high, there is little to no opportunity for pre-warming a function. Specifically, when $\mathsf{T}_2+\mathsf{I}_2 \geq \mathsf{IT}$, two possible strategies exist for handling subsequent invocations of F_2: 1) terminating the instance after the last invocation and creating a new one before the next invocation arrives, or 2) keeping live the function after the last invocation. Under the first strategy, two instances exist concurrently, leading to an execution cost of $(\mathsf{T}_2+\mathsf{I}_2)\cdot U(\star_2)$ for each invocation of F_2, as illustrated in Fig. 5(b). In contrast, the second strategy keeps the instance alive even if the last invocation has completed and the subsequent invocation has not yet arrived, incurring a cost of $\mathsf{IT}\cdot U(\star_2)$ for each invocation of F_2. Hence, SMIless chooses the second strategy, effectively setting the pre-warming window size to zero, yielding the same expression of $\mathcal{L}(\vec{\chi},\vec{\varphi})$ as that in Eq. (5).

The above comparison also implies that, when the SLA requirement is lenient, there is a higher likelihood of selecting low-end hardware for each function and keeping the corresponding instance alive for a long period, thereby minimizing the occurrence of initialization.

2) Adaptive batching: During peak workloads on a serverless platform, which refers to a situation where multiple invocations arrive within a short period of time, processing invocations sequentially within each function instance can easily result in SLA violations. As illustrated in Fig. 5(c), even with a zero pre-warming window size configuration, subsequent invocations are still required to wait for the completion of early-arrived invocations due to the smaller inter-arrival time IT compared to the function inference time I_2. To address this issue, the Auto-scaler under SMIless incorporates adaptive batching, where multiple invocations are batched together for parallel processing, reducing the waiting time. However, it should be noted that under this policy, the inference time of F_2 is increased compared to the sequential processing manner. To ensure compliance with the SLA requirement, SMIless also dynamically scales up to higher-end configurations as needed.

In cases where even higher-end hardware fails to meet the SLA, the Auto-scaler implements dynamic scaling out by launching multiple instances of the same function to handle the increased workload. In such scenarios, the overall performance metric $\mathcal{L}(\vec{\chi},\vec{\varphi})$ can be characterized as the sum of inference time for each individual function. Additionally, the execution cost of each function F_k is calculated as the product of the inter-arrival time IT and the unit cost $U(\star_k)$.

C. Co-optimization Algorithm Design

Drawing on the formulations of the E2E latency $\mathcal{L}(\vec{\chi},\vec{\varphi})$ and the overall execution cost $\sum_{k=1}^{N} C_k(\star_k,\triangle_k)$, the Strategy Optimizer optimizes the hardware configuration and associated cold-start management policy for all functions through *Top-K path searching* combined BFS (Breadth-First Search) with DFS (Depth-First Search). To initiate the path-search process, the Strategy Optimizer employs a multi-way tree to construct a comprehensive search space. In the multi-way tree, each node represents the hardware configurations and corresponding cold-start management policies for all functions within the application, thereby effectively addressing the cascading effect. By following a path from the root node to a leaf node, all the functions in the application are traversed, resulting in a unique solution to the original optimization problem. Additionally,

each node along the path contains a cost value that indicates the expense associated with the function based on the current policy.

1) Top-K path search process: Fig. 6 illustrates a straightforward example demonstrating the path-search process with top-1 selection. The path search algorithm commences at the root node T^0, which represents a combination of the configuration $\vec{\chi^0}$ and the policy $\vec{\varphi^0}$ that result in the lowest overall execution cost. Under the configuration $\vec{\chi^0}$, all functions execute on hardware that incurs the lowest cost, determined by

$$\star_k^0 = \arg\min_{\star_k \in \mathcal{C}} C_k(\star_k, \triangle_k), \triangle_k = \text{adaptive pre-warming.} \quad (6)$$

If the corresponding $\mathcal{L}(\vec{\chi^0}, \vec{\varphi^0})$ falls within the SLA, the search process is concluded, and $\{\vec{\chi^0}, \vec{\varphi^0}\}$ is deemed the optimal solution. In case the SLA is not met, the Strategy Optimizer shifts its focus to T_1^1, where F_1 is assigned hardware resulting in the second smallest execution cost, and the remaining functions run on hardware leading to the shortest inference time. If the resulting latency meets the SLA, the Strategy Optimizer proceeds to traverse T_2^1, assigning F_2 the most cost-effective hardware while other functions continue on the most advanced hardware. The Strategy Optimizer checks if T_2^1 satisfies the SLA, and if so, SMIless proceeds to T_3^1, where F_3 is assigned the most cost-effective hardware. However, if T_2^1 fails to meet the SLA, the Strategy Optimizer moves to T_2^2. Here, F_2 is assigned the second most cost-effective hardware, as it is infeasible for the present configuration of F_2 to meet the SLA. SMIless repeats this process until it reaches a leaf node. At this stage, the Strategy Optimizer has successfully identified a path satisfying the SLA requirement.

The top-1 path search process can be seamlessly extended to a top-K path search, further optimizing the overall execution cost. Specifically, upon meeting the SLA with the most cost-effective strategy combination in layer l_i, the Strategy Optimizer shifts focus to exploring the subsequent $K-1$ strategy combinations with minimal costs that still fulfill the SLA, rather than immediately delving into the next layer l_{i+1} in the multi-way tree. After identifying the top-K strategy combinations with the lowest costs within layer l_i, the Strategy Optimizer progresses to layer l_{i+1}, pinpointing the top-K lowest-cost nodes among all child nodes stemming from the previously selected nodes in layer l_i. This iterative process iterates until reaching the leaf node. However, it is crucial to note that in scenarios where the DAG presents a lengthy path, the Strategy Optimizer may spend excessive time exploring the top-K lowest-cost strategy combinations, potentially leading to suboptimal solutions, especially in highly dynamic invocation scenarios. Consequently, this work solely focuses on the top-1 path search process.

2) Handling complex applications: When dealing with applications comprising intricate parallel branches, the Workflow Manager decomposes the application's DAG into multiple subgraphs that only contain sequential dependency. The Strategy Optimizer then executes the basic algorithm in parallel for each subgraph to obtain an initial solution. To achieve a solution with reduced cost, the Workflow Manager combines the results obtained from these subgraphs.

The combining process begins by traversing the DAG graph to identify the smallest substructure containing parallel branches. Suppose the starting and end functions of a parallel branch structure are F_s and F_e respectively, the Workflow Manager traverses each path containing the functions F_s and F_e in order to discover the configuration with the shortest inference time, which is subsequently assigned as the final configuration for both F_s and F_e. Following this, the Workflow Manager updates the configurations of other functions along these parallel branches to ensure that the overall E2E latency of each path remains unchanged before and after combining the subgraphs. The workflow Manager repeats this process for the next minimum parallel substructure until all parallel substructures have been processed.

3) Complexity Analysis: By employing DAG decomposition, the Strategy Optimizer can simultaneously handle each decomposed simple graph, thereby making the algorithm's complexity dependent on the longest path's length. Assuming there are N functions along this longest path, with each function having M hardware configuration candidates, the algorithm first identifies the cost-minimizing configuration combination for all functions. This process involves traversing the longest path across the N functions and selecting the most cost-effective configuration for each function. As a result, the time complexity for obtaining the initial configuration is $O(N \cdot M \cdot \log M)$. If the end-to-end latency of the application fails to meet the SLA requirement, the algorithm proceeds with replacing hardware configurations for functions to reduce the application's latency. In the worst-case scenario, the algorithm may need to traverse all M configurations of each of the N functions to verify whether the end-to-end latency with the current configuration meets the SLA requirement. Based on Eq. (5), calculating the end-to-end latency for updating the configuration of a single function involves summing the end-to-end latency of the previous configuration combination with the incremental inference time resulting from the current function's configuration change, with an asymptotic time complexity of $O(1)$. Consequently, the algorithm's time complexity in the worst case is $O(N \cdot M)$. Therefore, the overall time complexity of SMIless' search algorithm remains at $O(N \cdot M \cdot \log M)$.

D. Container Autoscaling

At the start of each time window, the Optimizer Engine determines both the desired hardware configuration and warming-up time for each function instance. In cases where the corresponding inference time on the selected hardware exceeds the predicted inter-arrival time, subsequent invocations of the application may violate the SLA requirement, as illustrated in Fig. 5(c). In such situations, the Optimizer Engine adopts resource autoscaling to modify the hardware configuration and number of function instances accordingly.

When predicting the number of invocations within the next interval for a specific inference model, denoted as G, and

considering the required inference time as $\mathsf{I_s}$ obtained from the optimization algorithm in § V-C, the Auto-scaler selects to batch B invocations for parallel processing within each function instance. Consequently, the number of instances is determined as $\frac{G}{B}$. In order to determine the optimal container configuration and batch size, the Auto-scaler formulates the following optimization problem:

$$\min_{\{\star_k, B\}} \frac{G}{B} \cdot \mathsf{IT} \cdot U(\star_k),$$
$$\text{subject to, } \lambda_c \times B \times \left(\frac{\alpha_c}{\#\text{ of CPU cores}} + \beta_c\right) + \gamma_c \leq \mathsf{I_s}. \tag{7}$$

This optimization problem applies when the inference function is served under CPU resources. However, considering that the GPU backend may potentially result in a lower cost, the Auto-scaler also formulates the problem by using GPU instances. In this case, the constraint in Expression (7) is replaced with the corresponding GPU version:

$$\lambda_g \times B \times \left(\frac{\alpha_g}{\%\text{ of GPU}} + \beta_g\right) + \gamma_g \leq \mathsf{I_s}. \tag{8}$$

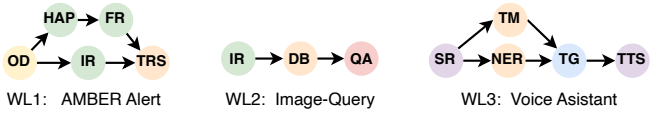
The Auto-scaler employs multiple threads to solve the optimization problems for all functions in parallel. When dealing with each optimization problem, it adopts the Bisection method to determine the optimal solution for B and the corresponding configuration $\star_k$.

VI. System Implementation

SMIless, our serverless ML inference system, is implemented on top of OpenFaaS [28], an event-driven computing framework that utilizes Kubernetes [47]. Prometheus [41] is used by SMIless during the runtime to capture and store data relevant to inference functions. The Optimization Engine module, Offline Profiling module, and Online Predictor module are developed via the Python library in approximately 3K lines of code. In addition, the Container Manager component is implemented with 2K lines of Go code.

The Workflow Manager module leverages NetworkX [48] to efficiently handle the DAG structure of each application. The Optimization Engine module utilizes the Kubernetes Python client library to monitor the submission of applications. To enable parallel processing, the Strategy Optimizer module incorporates the Python multi-processing library to generate optimal strategies. The Auto-scaler module runs as a separate backend process in Python, dynamically determining the scaling strategy. Another backend process, the Offline Profiler, continuously fetches running information of each inference model from Prometheus for profiling purposes. Moreover, the Online Predictor module operates in a separate process, directly communicating with the Gateway to obtain invocation patterns for training.

The Container Manager utilizes timers to control the start and termination of each instance based on the function's pre-warming policy. We have implemented an Agent module within each instance, which efficiently handles batched invocation requests received from the invocation client in parallel. The batchsize for each instance is stored in Kubernetes as a ConfigMap, which is updated by the Auto-scaler module. Once the inference tasks are completed, the Agent returns the results back to the corresponding invocation client. The communication between the Agent and invocation client occurs through HTTP requests, while the Agent effectively communicates with the inference function's process via RPC.



WL1: AMBER Alert WL2: Image-Query WL3: Voice Asistant

Fig. 7: ML serving applications with DAG workflows.

Field	Function Name	Model	Input
♦	Image Recognition (IR)	ResNet50 [52]	ImageNet [53]
♦	Face Recognition (FR)	FaceNet	ImageNet
♦	Human Activity Pose (HAP)	ResNet50 [34]	ImageNet
♡	DistilBert (DB)	BERT [32]	SQuAD [54]
♡	Name Entity Recognition (NER)	Flair [55]	SQuAD
♡	Topic Modeling (TM)	TweetEval [33]	SQuAD
♡	Translation (TRS)	T5 [56]	SQuAD
□	Text Generation (TG)	GPT2 [57]	SQuAD
⋆	Speech Recognition (SR)	Wav2Vec [58]	SQuAD
⋆	Text To Speech (TTS)	FastSpeech [37]	SQuAD
♠	Object Detection (OD)	YOLOv5 [59]	COCO [60]
♣	Question Answering (QA)	Roberta [36]	SQuAD

♦ Image Classification. ♡ Language Modeling. □ Text Generation. ⋆ Audio Processing. ♠ Object Detection. ♣ Question Answering.

TABLE I: Inference models

VII. Evaluation

A. Experimental Setup

Applications. We utilize three popular ML serving applications, along with their corresponding topology depicted in Fig. 7. Detailed descriptions of the inference functions can be found in Table I. Unless explicitly stated otherwise, the SLA target for each application is set to two seconds.

- AMBER Alert [49], [10] serves as an emergency alert system for child abduction cases. It is activated when an image is uploaded and subsequently conducts object detection to identify and label vehicles and humans present in the image. The application then generates an alert message that is translated into various languages.
- Image Query [50], [5] generates natural language descriptions of images as output for the user. It is activated when the user sends an image to the server.
- Voice Assistant [51], [5] promptly addresses user voice queries by converting the spoken input into text. It then analyzes the text to generate appropriate responses, transforming them into audio outputs for the user.

Load generator. We generate custom-shaped workloads based on scaled-down invocation pattern traces obtained from the Azure Function Dataset [61]. These function invocation traces are used to simulate the invocation requests of applications with a DAG workflow. We scale down the invocation interval from one minute to two seconds and deploy a dedicated load generator for each application, operating simultaneously. The evaluation duration for each application spans 2 hours.

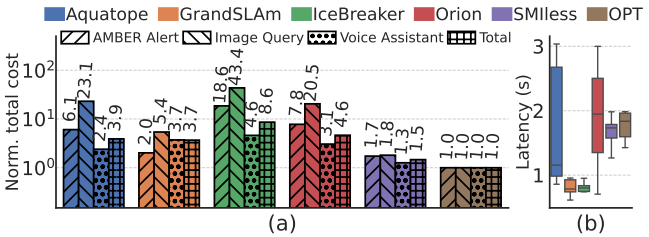


Fig. 8: The comparison across different solutions where OPT denotes optimal policy. (a) Overall execution cost. (b) Distribution of the E2E latency.

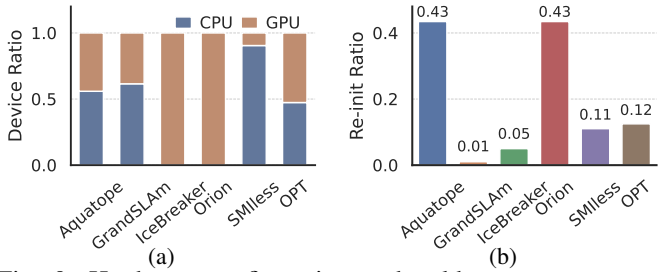


Fig. 9: Hardware configuration and cold-start management across different systems. (a) The ratio of CPU to GPU usage. (b) Fractions of container reinitialization.

System Settings. We evaluate SMIless on an 8-machine cluster. Each machine is equipped with two 52-core Intel x86 Xeon Gold 5320 CPUs, 128GB of memory, and a Nvidia RTX 3090 GPU. The machines are connected via a 10GbE NIC and they have CUDA 11.5 and cuDNN 8 installed. SMIless executes serverless functions within container instances on the cluster, utilizing both the CPU and GPU devices. The CPU containers come in five different specifications, equipped with 1, 2, 4, 8, or 16 cores, which correspond to the AWS c6g series instances [62]. The usage cost for these instances is $x \times \$0.034/\text{hour}$, where x represents the number of CPU cores. Furthermore, GPU resources for the containers are allocated in units of 10% using MPS. The price for a container with 10% GPU allocation is 10% of the price of AWS GPU instances p3.2xlarge [63], which amounts to \$3.06/hour.

Baselines. We benchmark SMIless against several state-of-the-art serverless systems focused on optimizing resource provisioning: GrandSLAm [5], IceBreaker [17], Orion [4], and Aquatope [24]. Specifically, GrandSLAm is a holistic runtime framework designed for multi-stage ML applications with the goal of maximizing throughput while maintaining SLA requirements. Aquatope is an uncertainty-aware QoS scheduler for serverless workflows, utilizing Bayesian Optimization to identify the optimal resource configuration.

B. E2E Performance

As illustrated in Fig. 8, it is crucial to note that SMIless closely approximates the optimal solution (determined through exhaustive search), incurring only an additional total cost of 50% across all three applications. Importantly, as the complexity of the DAG increases, the cost disparity between SMIless and the optimal solution diminishes. We also observe that SMIless achieves a cost reduction of up to 5.73× compared

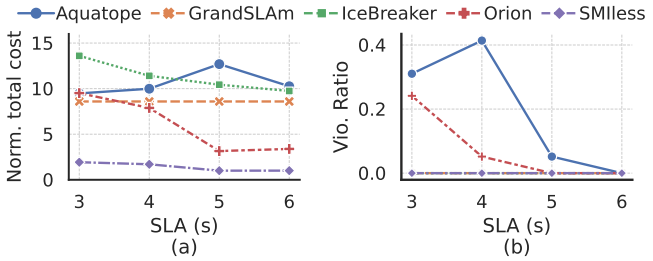


Fig. 10: The E2E performance under different SLA settings. (a) Total execution cost. (b) SLA violation ratio.

to IceBreaker, while maintaining SLA compliance without any violations. This is because IceBreaker solely considers invocation numbers and function speed-ups across different hardware, disregarding the DAG structure. Consequently, it resulted in a high proportion of long-keeping-alive instances with GPU resources, causing most functions to remain active on GPUs, as illustrated in Fig. 9(a).

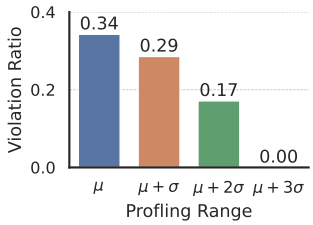
Orion experiences a relatively high violation ratio of approximately 40% due to the lack of effectively co-optimizing hardware selection and cold-start mitigation. As a result, Orion cannot fully leverage invocation patterns to optimize the configuration of different functions, resulting in a cost that is twice as high as SMIless. Conversely, Aquatope achieves a low cost but at the expense of a high SLA violation ratio, which can reach up to 40%. Although Aquatope considers invocation patterns and designs resource configurations based on the DAG structure, the frequent initialization of containers leads to many cold-starts. To validate this argument, we evaluate the fraction of function reinitialization during the entire experiment, as depicted in Fig. 9(b), where Aquatope exhibits the most frequent initialization among all solutions. While GrandSLAm demonstrates low E2E latency due to fewer initialization times, the absence of cold-start management results in a cost that is as high as 2.46× that of SMIless.

Furthermore, we investigate the impact of SLA settings on the overall execution cost. As shown in Fig. 10(a), Orion demonstrates the greatest benefit from stringent SLA settings. Specifically, when the SLA exceeds 5 seconds, Orion incurs only double the costs compared to SMIless. Fig. 10(b) provides the corresponding SLA violation ratio. Importantly, it should be noted that regardless of the SLA settings, SMIless consistently achieves the lowest cost with no SLA violations. Moreover, SMIless demonstrates stable performance in terms of total costs, attributed to its path search strategy, which updates the hardware configuration and cold-start policy of only a few functions when the SLA requirement changes.

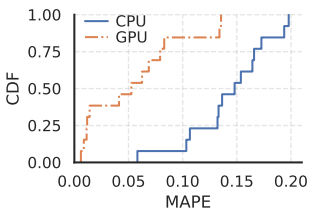
C. Source of Improvement

In this section, we deep-dive into the reasons behind SMIless' wins in both the cost and SLA compliance.

1) Offline profiling: First, the profiling methodology has a heavy impact on the SLA violation ratio. During the experiments, the Profiler utilizes running traces that are 30 minutes long for each function to perform profiling. As depicted in Fig. 11(a), when the average initialization time is used as the

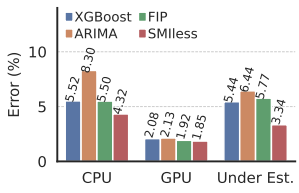


(a) The influence of profiling initialization time

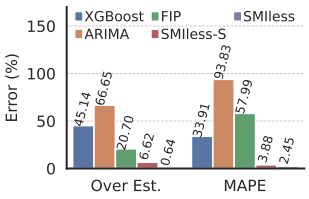


(b) The accuracy of profiling inference time

Fig. 11: Offline profiling results under SMIless.



(a) Invocation number



(b) Inter-arrival time

Fig. 12: The prediction accuracy of the invocation number and the inter-arrival time varies with different models.

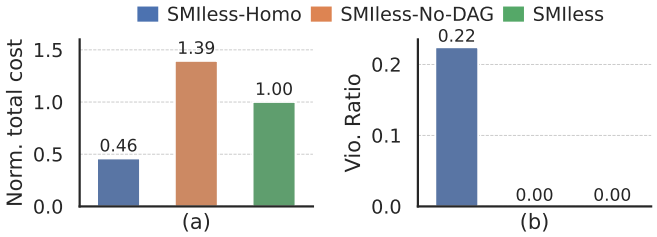


Fig. 13: The advantage of co-optimization. SMIless-Homo only launches containers with CPU backend while SMIless-No-DAG starts all functions simultaneously.

measurement, the SLA violation ratio can reach as high as 34%. However, with SMIless, by incorporating $3\times$ uncertainty, SLA violations can be completely avoided.

Additionally, the profiling of function inference time plays a crucial role in accurately guiding the co-optimization process. As depicted in Fig. 11(b), the profiling methods demonstrate high accuracy. All functions exhibit a Symmetric Mean Absolute Percentage Error (SMAPE) of less than 20%, with the overall average being below 8%. It is worth noting that the prediction of GPU inference time is more precise compared to CPU inference time. This discrepancy is attributed to the fact that execution on the CPU is more susceptible to interference. The exceptional accuracy of inference time profiling is achieved with only $5\times5=25$ samples on the CPU backend (encompassing 5 types of batch sizes from $2^1\sim2^5$ and $2^0\sim2^4$ CPU cores) and 50 samples on the GPU backend (covering 10 different percentages of GPU allocation).

2) Online prediction: Second, the accurate prediction enables SMIless to perform reliable proactive resource provisioning in case of highly dynamic invocation arrivals. We compare results obtained from different predictors trained on a 1-hour trace and tested on an 21-hour dataset, which exhibits a variance-to-mean ratio of invocation numbers greater than two. These predictors are: 1) XGBoost [64]; 2) ARIMA [61], a widely adopted time-series sequence model; 3) FIP [17], a method based on Fourier Transformation used by IceBreaker; 4) SMIless, employing one (or two) LSTM module with 30 (128) hidden states for invocation number (inter-arrival time).

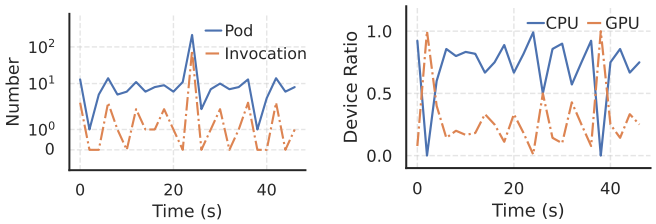
As shown in Fig. 12(a), the predictor for invocation number achieves an underestimation error of 3%, outperforming baselines, primarily attributed to its superior classification approach. To compensate for this error, SMIless incorporates an additional 3% to the original quantity. Without this compensation, the SLA violation ratio would escalate to 5%. Regarding overestimation, SMIless estimates a marginal increase of 4.32% in the container number with the CPU backend and 1.85% with the GPU backend when compared to the resource usage based on the true invocation number.

The predictor for inter-arrival times exhibits an impressively low MAPE of just 2.45%, as depicted in Fig. 12(b). Moreover, it exhibits a probability of over-estimations of less than 0.64%, attributed to its innovative design featuring two STM modules. In comparison to employing a single LSTM module with inter-arrival times as the sole input (referred to as SMIless-S), this configuration yields a remarkable reduction in over-estimation errors by a factor of 10. This high accuracy enables SMIless to achieve perfect overlap between function initialization and inference within the same application. Notably, substituting the LSTM predictor with ARIMA would result in an overall cost increase of 18%.

3) Co-optimization: The key contribution to the improvement achieved by SMIless can be attributed to the effective co-optimization of resource configuration and cold-start management. We evaluate two variants of SMIless: SMIless-No-DAG and SMIless-Homo, where the former disregards the DAG structure and warms up all function instances simultaneously based on inter-arrival time, while the latter only utilizes CPU backend for resource configuration. As demonstrated in Fig. 13(a), the overall cost under SMIless-No-DAG is 39% higher than that of SMIless, emphasizing the significance of global cold-start management among all functions. Additionally, when SMIless-Homo only considers homogeneous resources, the applications suffer from a high SLA violation ratio of up to 22%, as illustrated in Fig. 13(b).

D. Adaptation to Bursty Arrivals

We conduct an evaluation of SMIless in handling burst workloads. Specifically, we sample a 60-second time window during which the workloads exhibited wide fluctuations. As depicted in Fig. 14(a), SMIless shows a fast response to workload changes, with the number of pods varying significantly in accordance with the number of invocations. Regarding the proportion of containers with different backend types, Fig. 14(b) illustrates that when the number of invocations increases, the ratio of CPU-to-GPU usage also goes up dramatically. This can be attributed to the fact that GPUs are more efficient in processing batched invocation requests in parallel, thus requiring only a small number of GPU instances to be scaled out in a burst workload setting.



(a) The number of containers (b) CPU to GPU usage

Fig. 14: Resource provisioning under burst workload.

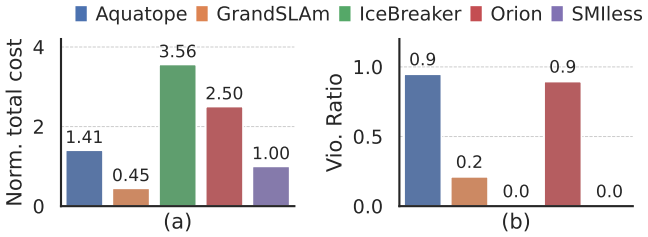
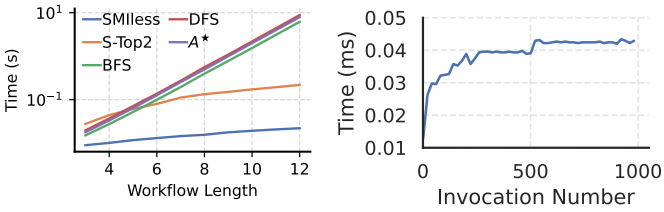


Fig. 15: Auto-scaling performance.

As shown in Fig. 15, SMIless demonstrates the optimal trade-off between reducing overall costs and meeting SLA requirements when making online scaling decisions. Specifically, during this period of bursty workloads, Auqatope, Orion, and Icebreaker result in costs that are more than $1.41\times$ higher. Although GrandSLAm achieves the lowest cost due to its restricted resource scaling capability, it consequently leads to SLA violations, with rates as high as 20%.



(a) Co-optimization overhead (b) The overhead of auto-scaler

Fig. 16: System overhead within SMIless.

E. System Overhead

As illustrated in Fig. 16(a), the overhead of finding a strategy using the Strategy Optimizer increases almost linearly as the length of the longest path within a DAG increases. Specifically, SMIless can find a near-optimal strategy within 20 ms for the longest path of a DAG with 12 functions, which achieves a $10\times \sim 100\times$ reduction compared with other path searching methods. Additionally, the Auto-scaler's optimization process takes less than 0.1 ms to compute optimal scaling results, as shown in Fig. 16(b). This overhead is relatively small, highlighting the efficiency of SMIless in co-optimizing resource configuration and cold-start management.

VIII. Related Work

Serverless workflow management. Recently, there has been a significant amount of research dedicated to resource management for serverless platforms that employ workflows. The key objective is to strike a balance between minimizing execution costs and avoiding SLA violations [4], [15], [1], [65], [66], [67], [68], [24], [69], [27], [10], [70]. However, many of these studies that focus on managing function configurations in serverless workflows tend to overlook the critical aspect of cold start management. Consequently, they may be more susceptible to SLA violations or have lower cost-efficiency [65], [69], [24], [10]. On the other hand, works that specifically address cold start management in serverless workflows [68], [4], [66], [1], [67], [15] often neglect the coupling between cold-start mitigation and resource configuration.

Serverless in heterogeneous environments. Heterogeneous resources present good opportunities to enhance the performance of serverless applications, and several works have been proposed to address this issue [14], [17], [10], [2], [18], [5]. However, they mainly focus on the cold start management of individual functions, without being aware of the DAG [14], [17], [2], [18]. While LLama [10] leverages heterogeneous resources, it requires function instances to be always kept alive, leading to significantly high costs.

IX. Conclusion

This paper pioneers a comprehensive approach to globally co-optimize resource configuration and cold-start management for applications with complex DAGs under a serverless computing platform, utilizing heterogeneous resources. Addressing challenges stemming from the cascading effect and highly dynamic invocation patterns, we propose a new cold-start management policy and an efficient path search algorithm capable of achieving near-optimal costs with minimal computation overhead. Extensive experiments demonstrate that our design can significantly reduce costs, without violating SLAs for E2E latency in handling all user requests.

X. Acknowledgement

This work is supported in part by the National Key R&D Program of China (No. 2021YFB3300200), the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2, 0081/2022/A2, and 0123/2022/AFJ), the National Natural Science Foundation of China (No. 62072451, 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), and Shenzhen Basic Research Program (No. JCYJ20220818101610023, KJZD20230923113800001), as well as the Multi-Year Research Grant of University of Macau (MYRG-GRG2023-00019-FST-UMDF). We also thank the anonymous reviewers for their valuable feedback.

References

[1] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus:{NIMBLE} task scheduling for serverless analytics," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 653–669.

[2] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[3] Serverless. Serverless machine learning. [Online]. Available: https://www.serverless-ml.org/

[4] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 303–320.

[5] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303958

[6] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[7] Y. Wang, K. Chen, H. Tan, and K. Guo, "Tabi: An efficient multi-level inference system for large language models," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 233–248.

[8] J. Simon, "Building machine learning inference pipelines at scale," in *AWS Summit Tel*, 2019.

[9] M. Radzihovsky, F. Memarian, E. Can, and B. Yoldemir. Serving ml model pipelines on nvidia triton inference server with ensemble models. [Online]. Available: https://developer.nvidia.com/

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

[11] Y. Tong, W. Liao, and Q. Ji, "Inferring facial action units with causal relations," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1623–1630.

[12] B. Li, T. Wu, and S.-C. Zhu, "Integrating context and occlusion for car detection by hierarchical and-or model," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI 13*. Springer, 2014, pp. 652–667.

[13] Z. Wang, H. Chen, G. Wang, H. Tian, H. Wu, and H. Wang, "Policy learning for domain selection in an extensible multi-domain spoken dialogue system," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 57–67.

[14] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.

[15] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 356–370.

[16] R. B. Roy, T. Patel, and D. Tiwari, "Daydream: executing dynamic scientific workflows on serverless platforms with hot starts," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2022, pp. 291–308.

[17] R. R. Basu, P. Tirthak, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 753–767.

[18] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving." in *Proceedings of ATC*, 2021.

[19] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Building heterogeneous cloud system for machine learning inference," *arXiv preprint arXiv:2210.05889*, 2022.

[20] W. Jiang, Z. He, S. Zhang, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang *et al.*, "Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 3097–3105.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[22] H. Zhao and R. Sakellariou, "Scheduling multiple dags onto heterogeneous systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 1–14.

[23] F. Dong and S. G. Akl, "Pfas: a resource-performance-fluctuation-aware workflow scheduling algorithm for grid computing," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–9.

[24] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2023, pp. 1–14.

[25] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[26] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[27] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1868–1877.

[28] Openfaas. [Online]. Available: https://www.openfaas.com/

[29] Azure durable functions. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/

[30] W. Zhang, Q. Chen, K. Fu, N. Zheng, Z. Huang, J. Leng, and M. Guo, "Astraea: towards qos-aware and resource-efficient multi-stage gpu services," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 570–582.

[31] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[33] F. Barbieri, J. Camacho-Collados, L. Espinosa Anke, and L. Neves, "TweetEval: Unified benchmark and comparative evaluation for tweet classification," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1644–1650. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.148

[34] B. Xiao, H. Wu, and Y. Wei, "Simple baselines for human pose estimation and tracking," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 466–481.

[35] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *In Proceedings of CVPR, 2015*. IEEE, jun 2015. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2015.7298682

[36] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[37] C. Wang, W.-N. Hsu, Y. Adi, A. Polyak, A. Lee, P.-J. Chen, J. Gu, and J. Pino, "fairseq s^2: A scalable and integrable speech synthesis toolkit," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 143–152. [Online]. Available: https://aclanthology.org/2021.emnlp-demo.17

[38] Google workflows. [Online]. Available: https://cloud.google.com/workflows

[39] Tfx. [Online]. Available: https://www.tensorflow.org/tfx

[40] Aws ec2. [Online]. Available: https://aws.amazon.com/ec2/

[41] Prometheus. [Online]. Available: https://prometheus.io/

[42] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[43] Nvidia. (2023) Nvidia mps. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[44] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[45] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and optimizing dynamic control flow in deep learning," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 681–699.

[46] Y. Xiao, K. Thulasiraman, G. Xue, A. Jüttner, and S. Arumugam, "The constrained shortest path problem: Algorithmic approaches and an algebraic study with generalization," *AKCE International Journal of Graphs and Combinatorics*, vol. 2, no. 2, 2005.

[47] Kubernetes. [Online]. Available: https://kubernetes.io/

[48] Networkx. [Online]. Available: https://networkx.org/

[49] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian, "Scanner: Efficient video analysis at scale," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–13, 2018.

[50] T. Lüddecke and A. Ecker, "Image segmentation using text and image prompts," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 7086–7096.

[51] C. Yan, X. Ji, K. Wang, Q. Jiang, Z. Jin, and W. Xu, "A survey on voice assistant security: Attacks and countermeasures," *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–36, 2022.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *In Proceedings of CVPR*, 2016, pp. 770–778.

[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *In Proceedings of CVPR*, 2009.

[54] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[55] A. Akbik, D. Blythe, and R. Vollgraf, "Contextual string embeddings for sequence labeling," in *COLING 2018, 27th International Conference on Computational Linguistics*, 2018, pp. 1638–1649.

[56] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[57] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[58] A. Baevski, Y. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: A framework for self-supervised learning of speech representations," *Advances in neural information processing systems*, vol. 33, pp. 12 449–12 460, 2020.

[59] G. Jocher, A. Stoken, J. Borovec, A. Chaurasia, L. Changyu, V. Abhiram *et al.*, "Ultralytics/yolov5: V5. 0—yolov5-p6 1280 models, aws, supervise," *Ly and YouTube Integrations*, vol. 10, 2021.

[60] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014.

[61] 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 *Proceeding of ATC*, 2020.

[62] Amazon ec2 c6g instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/c6g/

[63] Amazon ec2 p3 instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p3/

[64] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[65] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.

[66] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1505–1519.

[67] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 782–796.

[68] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 153–167.

[69] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, and M. Shahrad, "UnFaaSener: Latency and cost aware offloading of functions from serverless platforms," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 879–896.

[70] H. Wu, J. Deng, H. Fan, S. Ibrahim, S. Wu, and H. Jin, "Qos-aware and cost-efficient dynamic resource allocation for serverless ml workflows," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 886–896.