Kraken : Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms

Abstract

The growing popularity of microservices has led to the proliferation of online cloud service-based applications, which are typically modelled as Directed Acyclic Graphs (DAGs) comprising of tens to hundreds of microservices. The vast majority of these applications are user-facing, and hence, have stringent SLO requirements. Serverless functions, having short resource provisioning times and instant scalability, are suitable candidates for developing such latency-critical applications. However, existing serverless providers are unaware of the workflow characteristics of application DAGs, leading to, in most cases, container over-provisioning, which is further exacerbated in the case of dynamic DAGs, where the function chain for an application is not known a priori. Motivated by these observations, we propose *Kraken*, a workflow-aware resource management framework that minimizes the number of containers provisioned for an application DAG, while ensuring SLO-compliance. We design and implement *Kraken* on *OpenFaaS* and evaluate it on a multi-node *Kubernetes*-managed cluster. Our extensive experimental evaluation using *DeathStarbench* workload suite and real-world traces demonstrates that *Kraken* spawns up to 76% fewer containers, thereby improving container utilization and saving cluster-wide energy by up to 4× and 48%, respectively, when compared to state-of-the art schedulers employed in serverless platforms.

1 Introduction

Cloud applications are embracing microservices as a premier application model, owing to their advantages in terms of simplified development and ease of scalability [28, 37]. Many of these real-world services often comprise of tens or even hundreds of loosely-coupled microservices [39] (e.g., Expedia [15] and Airbnb [2]). Typically, these online service applications are user-facing and hence, are administered under strict Service Level Objectives (SLOs) and response latency requirements. Therefore, choosing the underlying resources (virtual machines or containers) from a plethora of public cloud resource offerings [31, 34, 38, 42, 45] becomes crucial due to their characteristics (such as provisioning latency) that determine the response latency. Serverless computing (FaaS) has recently emerged as a first-class platform to deploy latency-critical user facing applications as it mitigates resource management overheads for developers, while simultaneously offering instantaneous scalability. However, deploying complex microservice-based applications on FaaS has unique challenges owing to its design limitations.

First, due to the stateless nature of FaaS, individual microservices have to be designed as functions and explicitly chained together using tools to compose the entire application, thus, forming a Directed Acyclic Graph (DAG) [31]. Second, the state-management between dependent functions has to be explicitly handled using a predefined state-machine and made available to the cloud provider [6, 23]. Third, the presence of conditional branches in some DAGs, can lead to uncertainties in determining which functions will be invoked by different requests to the same application. For instance, in a train-ticket application [37], actions like *make_reservation* can trigger different paths/workflows (subset of functions) within the application. These design challenges, when combined with the scheduling and container provisioning policies of current serverless platforms, result in crucial inefficiencies with respect to application performance and provider-side resource utilization. Two such inefficiencies are described below:

- The majority of serverless platforms [30, 41, 43, 45] assume that DAGs in applications are static, implying that all composite functions will be invoked by a single request to the application. This assumption leads to the spawning of equal number of containers for all functions in proportion to the application load, resulting in container over-provisioning.
- Dynamic DAGs, where only a subset of functions within each DAG are invoked per request type, necessitate the apportioning of containers to each function. Recent frameworks like Xanadu [26], predict the most likely functions to be used in the DAG. This results in container provisioning along a single function chain. However, not proportionately allocating containers to all functions in the application, can lead to under-provisioning containers for some functions when requests deviate from the predicted path.

To address these challenges, we propose *Kraken*, a DAG workflow-aware resource management framework specifically catered to dynamic DAGs, that minimizes resource consumption, while remaining SLO compliant. The key components of *Kraken* are (i) *Kraken* employs a Proactive Weighted Scaler (PWS) which deploys containers for functions in advance, by utilizing a request arrival estimation model. The number of containers to be deployed is jointly determined by the estimation model and function weights. These weights are assigned by the PWS by taking into account the function invocation probabilities and parameters pertaining to the DAG structure namely, *Commonality* (functions common to multiple workflows) and *Connectivity* (number of descendant functions), (ii) In addition to the PWS, *Kraken* employs a Reactive Scaler (RS) to scale containers appropriately to recover from potential resource mismanagement by the PWS.

We have developed a prototype of *Kraken* using *OpenFaaS*, an open source serverless framework [11], and extensively evaluated it using real-world datacenter traces on a 160 core *Kubernetes* cluster. Our results show that *Kraken* spawns up to 76% fewer containers on average, thereby, improving container utilization and cluster-wide energy savings by up to 4× and 48%, respectively, when compared to state-of-the art serverless schedulers. Furthermore, *Kraken* guarantees SLO requirements for up to 99.97% of the requests.

2 Background and Motivation

We start with providing an overview of serverless DAG's along with the related work (Table 1) and discuss the challenges which motivate the need for *Kraken*.

2.1 Serverless Function Chains (DAG's)

Many applications are modelled as function chains and typically administered under strict SLOs (hundreds of milliseconds) [29]. Serverless function chains are formed by stitching together various individual serverless functions using some form of synchronization to provide the functionality of a full-fledged application. Function chains are supported in commercial serverless platforms such as AWS Step Functions [4, 23], IBM Cloud Functions [8], and Azure Durable functions [6]. By characterizing production application traces from Azure, Shahrad et.al [39] have elucidated that 46% of applications have 2-10 functions. Excluding the most general (and rare) cases, where applications can have loops/cycles within a function chain [26], applications can be modelled as a *Directed Acyclic Graph* where each vertex/stage is a function (DAG) [25] Henceforth, we will use the terms 'function' and 'stage' interchangeably. We define a *workflow* or *path* within an application as a sequence of vertices and the edges that connect them, starting from the first vertex (or vertices) and ending at the last vertex (or vertices). An application invokes functions in the sequence as specified by the path in the DAG. Based on the nature of the workflow, function chains can be classified as Static or Dynamic.

2.1.1 Static DAGs: In static function chains (or DAGs), the workflows are specified in advance by the developer (using a schema), which is then orchestrated by the provider. This results in a predetermined path being traversed in the event of an application invocation. For example, in the *Hotel Reservation* (Figure 1c), if only one path (say, *NGINX-Make_Reservation*) is always chosen, it represents a static function chain. Henceforth, we refer to static function chains as Static DAG Applications (SDAs). Clearly, having prior knowledge of what functions will be invoked for an application, makes container provisioning easier for SDAs.

2.1.2 Dynamic DAGs: Although the application DAG consists of multiple functions that may be invoked, there are cases where the functions can themselves invoke other functions depending on the inputs they receive. We refer to such functions as Dynamic Branch Points (DBPs), and the chains they are a part of as Dynamic Function Chains. In such cases, deploying containers, without prior knowledge about the possible paths in the workflow, leads to sub-optimal container provisioning for individual functions. Figure 1 shows the DAGs for three Dynamic Function Chains. *Social Network* (Figure 1a), for example, is one such chain that has 11 functions in total, with each subset of functions contributing to multiple paths (7 paths in total). For instance, from the start function *NGINX*, any one of *Search*, *Make_Post*, *Read_Timeline*, and *Follow* can be taken. Henceforth, we refer to such Dynamic DAG Applications as DDAs.

Features	Archipelago [41]	Power-chief [46]	Fifer [30]	Xanadu [26]	GrandSLAm [32]	Sequoia [43]	Hybrid Histogram [39]	*Kraken*
SLO Guarantees	✓	✗	✓	✓	✓	✓	✓	✓
Dynamic DAG Applications	✗	✗	✗	✓	✗	✗	✗	✓
Slack-aware batching	✗	✗	✓	✗	✓	✗	✗	✓
Cold Start Spillover Prevention	✗	✗	✗	✗	✗	✗	✗	✓
Function Weight Apportioning	✗	✗	✗	✗	✗	✗	✗	✓
Energy Efficieny	✗	✓	✓	✓	✗	✗	✓	✓
Request Arrival Prediction	✓	✗	✓	✓	✓	✗	✓	✓
Satisfactory Tail Latency	✓	✗	✓	✗	✓	✓	✓	✓

Table 1. Comparing the features of *Kraken* with other state-of-the-art resource management frameworks.

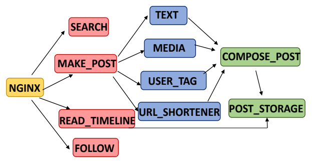
App	DBP	Total Fanout	Possible Paths	Max Depth
Social Network	2	8	7	5
Media Service	3	7	5	6
Hotel Reservation	1	2	2	4

Table 2. Analyzing Variability in Application Workflows.

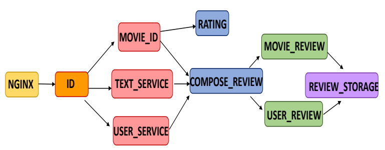
2.2 Motivation

Two specific challenges in the context of DDAs along with potential opportunities to resolve them are described below:

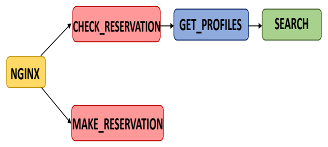
Challenge 1: Path Prediction in DDAs DDAs will only have a subset of their functions invoked for an incoming request to the application because of the presence of conditional paths within their DAGs. Figure 1 depicts the DAGs of three such applications from the *DeathStar* benchmark suite [28], and Table 2 summarizes the various workflows that can be triggered by an incoming request to them. 'Total fan-out' and 'Max Depth' denotes the total number of outgoing branches and maximum distance between the start function and any other function in a DAG, respectively. Note that each function triggers only one other function in the application at a time. The decision to trigger the next function typically depends on the input to the current function, although there are cases like *Media Service* where this decision may depend on previous function inputs as well. Therefore, there is considerable variation in the functions that can be invoked in DDAs, thus, negating the inherent assumption in many frameworks [30, 39, 41, 45] that all functions will be invoked with the same frequency as the application itself. This discrepancy can lead to substantial container overprovisioning.



(a) Social Network.

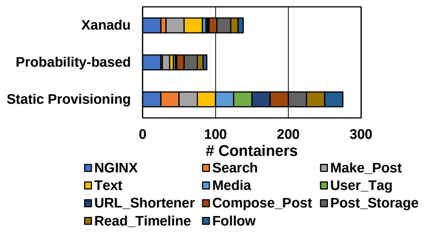


(b) Media Service.

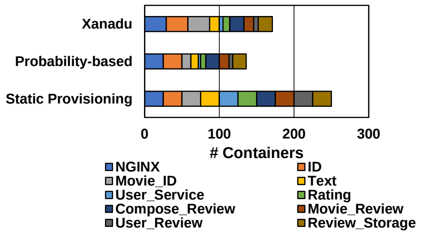


(c) Hotel Reservation.

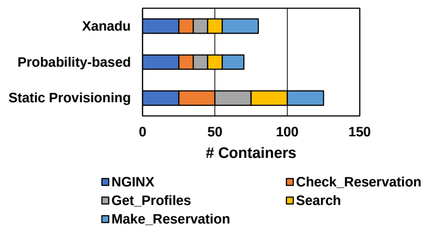
Figure 1. DAGs of Dynamic Function Chains.



(a) Social Network.



(b) Media Service.



(c) Hotel Reservation.

Figure 2. Function-wise Breakdown of Container Provisioning across Applications.

Opportunity 1: *In order to reduce overprovisioning of containers, it is vital to design a workflow-aware resource management (RM) framework that can dynamically scale containers for each function, as opposed to uniformly scaling for all functions. To design such a policy, the RM framework needs to know each function's invocation frequency, which is a good estimator of its relative popularity.*

We introduce weights to estimate the appropriate number of containers to be spawned for each function. *A function's weight is calculated using the relative invocation frequency of a function along other DAG-specific parameters* (explained in the next section). The relative invocation frequency of a function is measured with respect to its constituent application. The same function belonging to multiple applications can, therefore, have distinct weights in each application.

To analyze the benefits of using invocation frequency, we designed a scaling policy that employs weighted container scaling. For the purposes of this experiment, we base our function weights only based the invocation frequencies. Figure 2 depicts the number of containers provisioned per function for three container provisioning policies subject to a Poisson arrival trace (μ = 25 requests per second (rps)) for three applications. The static provisioning policy is representative of current platforms [45] which spawn containers for functions in a workflow-agnostic fashion. *Xanadu* [26] represents the policy that scales containers only along the Most Likely Path (MLP), which is the request's expected path. If the request takes a different path, *Xanadu* provisions containers along the path actually taken, in a *reactive* fashion, and scales down the containers it provisioned along the MLP. Consequently, *Xanadu*, when subject to moderate/heavy load, over-provisions containers by 32% compared to the Probability-based policy (from Figure 2) as a result of being locked into provisioning containers for the MLP until it is able to recalculate it. Our probability-based policy, on the other hand, provisions containers for functions along *every possible path in proportion to their assigned weights*. Note that variability in application usage patterns can lead to changes in function probabilities within each DDA, which the policy will have to account for.

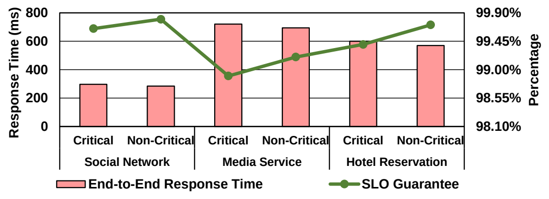


Figure 3. Performance Deterioration resulting from Container Deficiency at Critical Functions. The Primary Y-axis denotes the Average End-to-End Response Time, the Secondary Y-axis represents the percentage of SLOs satisfied and the X-axis indicates the Application under consideration.

Challenge 2: Adaptive Container Provisioning. While probability-based container provisioning can significantly reduce the number of containers, the presence of container cold-starts leads to SLO violations (requests not meeting their expected response latency). This is because, cold starts can take up a significant proportion of a function's response time (up to 10s of seconds [13, 14]). A significant amount of research [18, 22, 24, 35, 36, 40, 47] has been focused towards reducing cold-start overheads (in particular, proactive container provisioning [3, 30, 41, 43]) However, in the case of DDAs, DBPs make it unclear as to how many containers should be provisioned in advance for the functions along each path in the DAG.

We identify two interlinked factors, in the context of DDAs, that need to be accounted for when making container scaling decisions. The first, is what we call, *critical functions*. These are functions within a DAG that have a high number of descendant functions that are linked to it and we use the term *Connectivity* to denote the ratio of number of descendant functions to the total number of functions. Inadequately provisioning containers for such functions causes requests to queue up as the containers are spawned in the background.

Moreover, this additional request load trickles down to all the descendants, adversely affecting their response times as well. We refer to this effect as *Cold Start Spillover*. Figure 3 compares the performance degradation resulting from underprovisioning both Critical and Non-Critical functions. The (Critical, Non-Critical) function pairs chosen for this experiment were (*Make_Post*, *Text*), (*ID*, *Rating*) and (*NGINX*, *Search*) for *Social Network*, *Media Service* and *Hotel Reservation*, respectively. It can be observed that underprovisioning containers for just one Critical function has a greater impact on application performance than doing so for a single Non-Critical function, with the end-to-end response time and SLO guarantees becoming 24ms and 0.25% worse on average. This can lead to worsening effects if the same were to happen with multiple critical functions.

In addition to critical functions, it is also crucial to assign higher weights to common functions as well. Common functions refer to those which are a part of two or more paths within an application DAG. Figure 4 shows the 'hit rate' of functions within an application that is subject to a constant load where any path in the application is equally likely to be picked. It can be seen that functions which are common to a larger number of paths are invoked at a higher rate by such a request arrival pattern. Therefore, common functions have a higher chance of experiencing increased load due to being present in multiple paths. Therefore, higher weights have to be assigned to such functions to ensure resilience in the presence of varying application usage patterns.

Opportunity 2: *Although proactive provisioning combined with probability-based scaling is useful, it is essential to identify critical and common functions in each DDA and assign them higher weights in comparison to standard functions.*

Hence, rather than simply measuring the weights only in terms of function invocation frequency, we also need to account for DAG specific factors like *Commonality* and *Connectivity*. The above discourse motivates us to rethink the design of serverless RM frameworks to cater to DDAs as well. One key driver for the design lies in a *Probability Estimation Model* for individual functions, which is explained below.

3 Function Probability Estimation Model

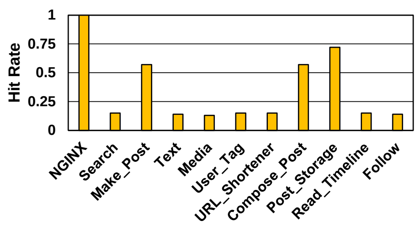
As elucidated in *Opportunity-1*, to specifically address the container over-provisioning problem for DDAs, we need to estimate the weights to be assigned to their composite functions, a key component of which is the function invocation probability. In this section, we model the function probability estimation problem using a Variable Order Markov Model (VOMM) [21]. VOMMs are effective in capturing the invocation patterns of functions within each application while simultaneously isolating the effects of other applications that share them. This aids us in the calculation of function invocation probabilities. Wherever appropriate, we draw inspiration from the related work that model user web surfing behavior [19, 20]. VOMMs are an extension of Markov Models [27], where the transition probability from the current state to the next state depends not only on the current state, but possibly on its predecessors (which we refer to as the 'context' of the state). It is essential to capture this in some of our workloads such as the *Media Service* application. The order of the VOMM denotes the number of predecessors that influence the transition decision.

An application DAG can map neatly onto a Markov model wherein the functions within the application DAG are modelled as states of the VOMM. The process of one function invoking another function corresponds to a transition from the caller function state to the callee function state. The weight for each function corresponds to the state transition probability from the start state to the current one (note that this may require possibly transitioning through a number of intermediate states).

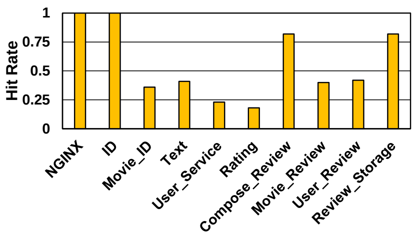
Thus, for a DAG with n functions, the transition probability matrix, T, is an $n \times n$ matrix, where n is the total number of states and each entry, $t_{j\ i}$, is the transition probability from the state corresponding to the function along the column j, (f_j), to that of the function along the row i, (f_i). An example of a Transition Matrix for the *Social Network*, with 11 functions, is depicted in Figure 5. An additional state, *end*, is added to represent the state the model transitions to after a path in the DAG is completely executed. In Figure 5, assuming both column and row indices of T start at 0, an entry $t_{0\ 4}$ represents the transition probability from *NGINX*'s state to *Follow*'s state and is equal to 0.2. In general, this transition probability, $t_{j\ i}$, is calculated as the number of requests from f_j to f_i divided by the number of incoming requests to f_i in the context of the application being considered.

The Probability Vector is an n×1 column vector that captures the probabilities of the model being in different states after a number of time steps have elapsed, given that the model was initialized at a known state. A 'time step' refers to a unit of measuring state change in the Markov Model. For practical purposes, we fix it to be the execution time of the slowest function at the current function depth. The 'depth' of a function, in this context, is defined as the distance, in terms of the number of edges in the DAG, from the start state to the current state. The Probability Vector after d number of time steps can be represented as P_d. Then, the Probability Vector for the next time step, $d + 1$ is given by the *transition equation*, $P_{t+1} = T \cdot P_t$. This equation infers that the Probability Vector at the next time step is obtained by performing a transition operation across all possible current states.

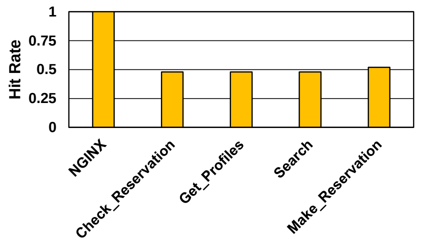
Repeatedly carrying out this transition process, starting from the initial Probability Vector, enables the estimation of probabilities of each function along all possible workflows. Iterating this process for d time steps would yield the probabilities of functions at a depth of d from the start function, given by $P_d = T^d \cdot P_0$. Thus, we can compute the probability



(a) Social Network.

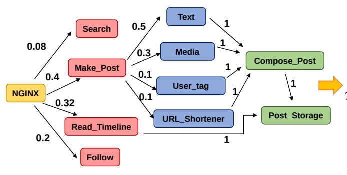


(b) Media Service.



(c) Hotel Reservation.

Figure 4. Function Hit Rate for an Evenly Distributed Load across all Paths in each Application.



$$T = \begin{array}{c|c} \begin{array}{cccccccccccc} \text{NGINX} & \text{Search} & \text{Make\_Post} & \text{Read\_Timeline} & \text{Follow} & \text{Text} & \text{Media} & \text{User\_Tag} & \text{URL} & \text{Compose\_Post} & \text{Post\_Storage} & \text{end} \end{array} & \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.08 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} & \begin{array}{l} \text{NGINX} \\ \text{Search} \\ \text{Make\_Post} \\ \text{Read\_Timeline} \\ \text{Follow} \\ \text{Text} \\ \text{Media} \\ \text{User\_Tag} \\ \text{URL} \\ \text{Compose\_Post} \\ \text{Post\_Storage} \\ \text{end} \end{array} \end{array}$$

Figure 5. Transforming the Social Network DAG into a Transition Matrix.

Notation	Meaning
T	Transition Matrix
P_d	Probability Vector for functions at depth, d
n	# functions in application or # states in model
f_i, f_j	functions along row, i or column, j in T
$t_{j\ i}$	Transition probability from f_j *to* f_i
W_p	Probability calculation time window
t	Request arrival time
d	# time steps for which transitions are done
PL_t	Scalar that represents the anticipated # requests at time, t
NC_t^d	# containers needed for functions at depth d, at time t

Table 3. Notations used in Equations.

of any function in the DAG by varying the depth, d, using this equation. In order to apply this to proactive container allocation decisions, we can adopt the following procedure.

The incoming load to the application at time stamp, t, is denoted as PL_t and can be predicted using a load estimation model. Assuming each request to a function within the application spawns one container for that function, the number of containers to be provisioned in advance for functions at depth d is given by:

$$NC_t^d = \lceil\, PL_{\,t} \cdot (T^d \cdot P_0) \rceil$$

Here, NC_t^d is a column vector of n elements, each corresponding to the number of elements required to be provisioned for functions at a depth, d, from the start function. Provisioning these containers at a fixed time window in advance from t prevents cold starts from affecting the end-user experience. For example, if PL_t is estimated to be 25 requests, then from Figure 5, we obtain the number of containers needed for functions at depth, $d = 1$, by multiplying 25 with P_1 (which is $T^1 \cdot P_0$). Consequently, the total number of containers required for each function in the application can be computed by performing a summation of NC_t^d across all possible depths, d, from the start function.

We can now transform our previously-assumed Markov Model into a VOMM by splitting up context-dependent states into multiple context-independent states (the number of which is dependent on the DAG structure and the order of the VOMM). For example, in Figure 5, if the transition from *Compose_Post* to *Post_Storage* depended on the immediate predecessors of *Compose_Post*, the *Compose_Post* state would be context-dependent and would therefore, be split into context-independent states, namely, *Compose_Post*$|$*Text* (*Compose Post* given *Text* was already invoked), *Compose_Post*$|$*Media* etc. for the previous equations to hold. This changes the total number of states from n to N, the number of extended states, resulting in a larger Transition Matrix and Probability Vector. To calculate the required number of containers for a single function that has multiple context-independent states associated with it, we take the sum of the calculated values for all of those states.

4 Overall Design of Kraken

In this section, we describe the high-level design of *Kraken*[1] (Figure 6). *Kraken* leverages the function weight estimation model described in the above section along with several other design choices given as follows. Users submit requests in the form of invocation triggers to applications ❶ hosted on a Serverless platform. In Kraken, containers are provisioned in advance by the Proactive Weighted Scaler (PWS) ❷ to serve these incoming requests by avoiding cold starts. To achieve this, the PWS ❷ first fetches relevant system metrics (using a monitoring tool ❸ and the resource orchestrator logs). These metrics, in addition to a developer-provided DAG Descriptor ❹, are then used by the Weight Estimation module 2a of PWS ❷ to assign weights to functions on the basis of their invocation probabilities. *Commonality* and *Connectivity* (parameters in 2a) are additional parameters used in weight estimation to account for critical and common functions. Additionally, a Load Predictor module 2b makes use of the system metrics to predict the incoming load and uses this in conjunction with the calculated function weights to proactively spawn containers for each function. However, only a fraction of these containers are actually spawned. This is determined by the function's batch size. The batch

[1]Kraken is a legendary sea monster with tentacles akin to multiple paths/chains in a Serverless DAG.

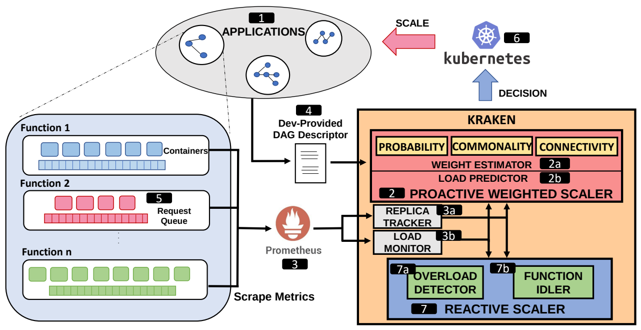


Figure 6. High-level View of Kraken Architecture

size denotes the number of requests per function each container can simultaneously serve without exceeding the SLO. The underlying resource orchestrator ❻ is responsible for deploying these containers. In order to effectively handle mispredictions in load, *Kraken* also employs a Reactive Scaler (RS) ❼ that consists of two major components. First, is an Overload Detector 7a that keeps track of request overloading at functions by monitoring queuing delays at containers. Subsequently, it triggers container scaling ❻ by calculating the additional containers needed to mitigate the delay. Second, a Function Idler component 7b evicts containers from memory ❻ when an excess is detected. Thus, *Kraken* makes use of PWS and RS to scale containers to meet the target SLOs while simultaneously minimizing the number of containers by making use of function invocation probabilities, function batching, and container eviction, where appropriate.

4.1 Proactive Weighted Scaler

This subsection describes in detail the integral components of PWS.

4.1.1 Estimating function weights:

Since workflows in SDAs are pre-determined, pre-deploying resources for them is straightforward in comparison to DDAs, whose workflow activation patterns are not known a priori. For DDAs, deploying containers for each function in proportion to the application load will inevitably lead to resource wastage. To address this, we design a Weight Estimator 2a to assign weights to all functions so as to allocate resources in proportion to them. Explained below is the working of the procedure *Estimate_Containers* in Algorithm 1 which is used to estimate function weights.

Probability: As alluded to in Section 2, one of the factors used in function weight estimation is its invocation probability. The procedure in Section 3 describes how the transition probabilities of the states associated with functions are computed through repeated matrix multiplications of the Transition Matrix, T with the Probability Vector, P. *Compute_Prob*, in Algorithm 1, first estimates the invocation probabilities of a function's immediate predecessors and uses it along with system log information and load measurements of the function to calculate its invocation probability.

Connectivity: In addition to function invocation probabilities, it is necessary to also account for the effects of cold starts on DDAs while estimating function weights. Cold start spillovers (that often occur due to container underprovisioning), as described in Section 2, can impact the response latency of applications harshly. Provisioning critical functions with more containers helps throttle this at the source. To this end, *Kraken* makes use of a parameter called *Connectivity*, while assigning function weights. The *Connectivity* of a function is defined as the ratio of number of its descendant functions to the total number of functions. The *Conn* procedure in Algorithm 1 makes use of this formula. For example, in Figure 1c, the *Connectivity* of *Check_Reservation* is $\frac{2}{5}$ since it has two descendants and there is a total of five functions. Bringing *Connectivity* into the weight estimation process helps *Kraken* assign a higher weight to critical functions, in turn, ensuring that more containers are assigned to them, resulting in improved response times for the functions themselves, as well as their descendants.

Commonality: As described in Section 2, in addition to cold start spillovers, incorrect probability estimations may arise due to variability in workflow activation patterns. This may be due to change in user behavior manifesting itself as variable function input patterns. Such errors can lead to sub-optimal container allocation to DAG stages in proportion to the wrongly-calculated function weights. To cope with this, we introduce a parameter called *Commonality*, which is defined as the fraction of number of unique paths that the function can be a part of with respect to the total number of unique paths. This is how the procedure *Comm* calculates *Commonality* in Algorithm 1. For example, in Figure 1a, the *Commonality* of the function *Compose_Post* in the *Social Network* application is given by the fraction $\frac{4}{7}$ as it is present in four out of the seven possible paths in the DAG. Using *Commonality* in the weight estimation process allows *Kraken* to tolerate function probability miscalculations by assigning higher weights to those functions that are statistically more likely to experience rise in usage because of their presence in a larger number of workflows. Note that we deal with the possibility of container overprovisioning due to the increased function weights by allowing both *Connectivity* and *Commonality* to be capped at a certain value.

4.1.2 Proactive Container Provisioning:

Once function weights are assigned by considering the above factors, they are employed in estimating the number of containers needed per DAG stage (*Estimate_Containers* in Algorithm 1). These containers have to be provisioned in advance to service future load to shield the end user from the effects of cold starts and thereby meet the SLO. This load will have to be predicted in order to make timely container provisioning decisions. *Kraken* makes use of a Load Predictor 2b (Algorithm 1 ⓐ) which uses the EWMA model to predict the incoming load at the end of a fixed time window, PW. This time window is

Algorithm 1 Proactive Scaling with weight estimation

```
for Every Monitor_Interval= PW do
    Proactive_Weighted_Scaler(∀functions)
procedure PROACTIVE_WEIGHTED_SCALER(FUNC)
    $cl \leftarrow Current\_Load(func)$
    $pl_{t+PW} \leftarrow \textbf{Load\_Predictor}(cl, pl_t)$ ⓐ
    $batches \leftarrow \left\lceil \frac{pl_{t+PW}}{func.batch\_size} \right\rceil$ ⓑ
    $total\_con \leftarrow \textbf{Estimate\_Containers}(batches, func)$
    $reqd\_con \leftarrow max(min\_con, total\_con)$
    $\textbf{Scale\_Containers}(func, reqd\_con)$
procedure ESTIMATE_CONTAINERS(LOAD, FUNC)                ▷ Output: reqd_con
    $func.prob \leftarrow \textbf{Compute\_Prob}(func)$
    $reqd\_con \leftarrow \lceil load * func.prob \rceil$
    $extra \leftarrow \lceil (\textbf{Comm}(func) + \textbf{Conn}(func)) * reqd\_con \rceil$
    $reqd\_con \leftarrow reqd\_con + extra$
```

chosen according to the time taken to scale all functions in the respective application. Note that t in the algorithm refers to the current time. We choose this model so as to have a light-weight load prediction mechanism that has minimal impact on the end-to-end latency. This Load Predictor 2b can be used in conjunction with the aforementioned Weight Estimator 2a to calculate the fraction of application load each function will receive. *Kraken* uses this load distribution to pre-provision the requisite number of containers for all functions in the application.

4.2 Request Batching

Many serverless frameworks [5, 10, 17, 26, 41, 43, 45] spawn a single container to serve each incoming request to a function. While this approach is beneficial to minimize SLO violations, comparable performance can be achieved by using fewer containers by leveraging the notion of slack [30, 32]. Slack refers to the difference in expected response time and actual execution time of functions within a function chain. Functions in a chain can have widely varying execution times. Allotting stage-wise SLOs to each function in a chain in proportion to their execution times reveals that there are cases where there is significant difference (slack) between the function's expected SLO and its run-time. Figure 7 depicts this slack for all functions in the applications considered.

This slack is leveraged by *Kraken* by batching multiple requests to the functions by queueing requests at their containers. Requests are batched onto containers in a fashion similar to the First Fit Bin Packing algorithm [33]. The number of containers to be spawned is determined by the batch size associated with each function. The batch size for a function, f, is defined as $BatchSize\ (f) = \left\lfloor \frac{StageSLO\ (f)}{ExecTime\ (f)} \right\rfloor$. The batch size represents the number of requests that can be served by a function without violating the allotted stage-wise SLO. This formula is used to compute the function batchsize in Algorithm 1 ⓑ. The introduction of batching results in a reduction in the number of containers spawned for each function in the application by a factor of its batch size (Algorithm 1 ⓑ).

4.3 Reactive Scaler (RS)

Though the introduction of Request Batching ❺ allows *Kraken* to reduce the containers provisioned, load mispredictions and probability miscalculations can still occur, leading to resource mismanagement, which could potentially affect the SLO compliance. To deal with this, *Kraken* also employs the RS ❼ to scale containers up or down in response to request overloading at containers (due to under-provisioning) and container over-provisioning, respectively. In case of inadequate container provisioning, the Overload Detector 7a in the RS ❼ detects the number of allocated containers for each DAG stage and calculates the estimated wait times of their queued requests (Algorithm 2 ⓑ). If it detects requests whose wait times exceed the cost of spawning a new container (the cold start of the function), overloading is said to have occurred at the stage. In such a scenario, *Kraken* batches these requests (*#_delayed_requests* in Algorithm 2) onto a newly-spawned container(s) (Algorithm 2 ⓒ). This is because requests that have to wait longer than the cold start would be served faster at a newly created container than by waiting at an overloaded container. Similarly, for stages where container overprovisioning has occurred, the RS ❼ gradually scales down its allocated containers to the appropriate number, if its Function Idler module 7b detects excess containers for serving the current load (Algorithm 2 ⓐ). Thus, the RS ❼, in combination with the PWS ❷ and request batching ❺, helps *Kraken* remain SLO compliant while using minimum resources.

Algorithm 2 Reactive Scaling

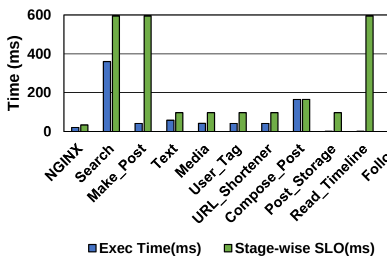
```
for Every Monitor_Interval= DR do
    Reactive_Resource_Manager(∀functions)
procedure REACTIVE_RESOURCE_MANAGER(FUNC)
    $cl \leftarrow Current\_Load(func)$
    $func.existing\_con \leftarrow Current\_Replicas(func)$
    if $\left\lceil \frac{cl}{func.batch\_size} \right\rceil \le func.existing\_con$ then ⓐ
        $reqd\_con \leftarrow \left\lceil \frac{cl}{func.batch\_size} \right\rceil$
    else
        $\#\_delayed\_requests \leftarrow \textbf{Delay\_Estimator}(func)$ ⓑ
        $extra\_con \leftarrow \left\lceil \frac{\#\_delayed\_requests}{func.batch\_size} \right\rceil$ ⓒ
        $reqd\_con \leftarrow func.existing\_con + extra\_con$
    $\textbf{Scale\_Containers}(func, reqd\_con)$
```

5 Implementation and Evaluation

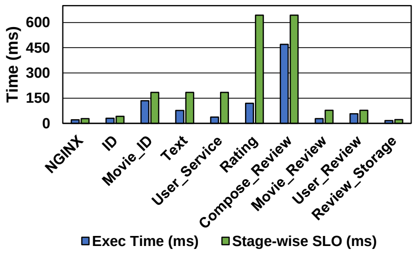
We have implemented a prototype of *Kraken* using open-source tools for evaluation with synthetic and real-world traces. The details are described below.

5.1 Prototype Implementation

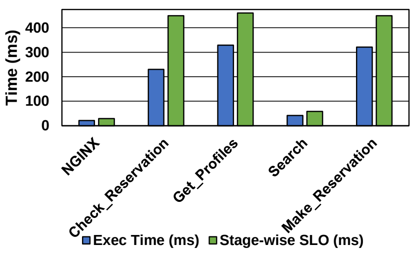
Kraken is implemented using, primarily, Python and Go, on top of *OpenFaaS* [11], an open-source serverless platform. *OpenFaaS* is deployed on top of *Kubernetes* [9], which acts as the chief container orchestrator. *OpenFaaS*, by default, comes packaged with an Alert Manager module which is responsible for alerting the underlying orchestrator of request surges by using metrics scraped by *Prometheus*, which is an open-source systems monitoring toolkit [12]. This, in turn, triggers autoscaling to provision extra containers to service the load surge. To cater to the design requirements of *Kraken*, we disable the in-built Alert Manager and deploy



(a) Social Network.



(b) Media Service.



(c) Hotel Reservation.

Figure 7. Slack for various Functions in each Application.

Policy	Component	Implemented using/as
PWS	Probability	System log info, Sparse Data Structures
	Commonality & Connectivity	DAG Descriptor
	Load Predictor	Pluggable model (EWMA)
Batching		Function containers persisted in memory
RS	Load Monitor Replica Tracker	Metrics from Prometheus & System logs

Table 4. Implementation details of *Kraken*'s policies.

the Proactive Weighted Scaler (PWS) and Reactive Scaler (RS) to carry out our container provisioning policies.

Both PWS and RS collect relevant metrics, such as the current container count, load history and request rate for a function for a given time window, from *Prometheus* and the *Kubernetes* system log, using the Replica Tracker and Load Monitor modules. The load to each function within each application is calculated separately using this information. This prevents other applications from interfering with the probability calculation of shared functions, thus allowing *Kraken* to run multiple applications concurrently. Additionally, the PWS uses a DAG descriptor (provided by the developer) to get an overview of the inter-function relationships. Table 4 gives an overview of *Kraken*'s policies and their implementation details.

5.2 Large Scale Simulation

To evaluate the effectiveness of *Kraken* in large-scale systems, we built a high-fidelity multi-threaded simulator, in Python, using container cold start latencies and function execution times profiled from our real-system counterpart. We have validated its correctness by comparing and correlating various metrics of interest generated using synthetic traces in both the simulator and the real-system. The simulator allows us to evaluate our model for a larger setup, where we mimic an 11k core cluster which can handle up to 7000 requests (70× more). Additionally, it helps compare the resource footprint of *Kraken* against a clairvoyant policy (Oracle) that has 100% load prediction accuracy.

5.3 Evaluation Methodology

We evaluate *Kraken* prototype on a 160-core *Kubernetes* cluster with a dedicated manager node. Each node is a Dell PowerEdge R740 server with 256GB of RAM and an Intel CascadeLake Xeon CPU host. For energy measurements, we use an open-source version of Intel Power Gadget [16] that measures the energy consumed by all sockets in a node.

Load Generator: We provide different traces as inputs to a load generator, which is based on Hey, an HTTP Load generator tool [7]. First, we use a synthetic Poisson-based request arrival rate with an average rate $\mu = 100$. Second, we use real-world request arrival traces from Wiki [44] and Twitter [1] by running each experiment for about an hour. The Twitter trace has a large variation in peaks (average = 3332 rps, peak= 6978 rps) when compared to the Wiki trace (average = 284 rps, peak = 331 rps).

Applications: Each request is modelled after a query to one of the three applications (DDAs) we consider from the *DeathStar* benchmark suite [28]. We implement each application as a workflow of chained functions in *OpenFaaS*. To model characteristics of the original functions, we invoke sleep timers within our functions to emulate their execution times (including the time for state recovery, if any). Transitions between functions are done using function calls on the basis of pre-assigned inter-function transition probabilities that vary (by approximately ±0.2) about a fixed value randomly. Note that these probabilities are not visible to *Kraken*, but are only used to model function invocation patterns.

Metrics and Resource Management Policies: We use the following metrics for evaluation: (i) average number of containers spawned, (ii) percentage of requests satisfying the SLO (SLO guarantees), (iii) average application response times, (iv) end-to-end request latency percentiles, (v) container utilization, and (vi) cluster-wide energy savings. We set the SLO at 1000ms. We compare these metrics for *Kraken* against the container provisioning policies of Archipelago [41], *Fifer* [30] and *Xanadu* [26], which we will, henceforth, refer to as *Arch*, *Fifer* and *Xanadu*, respectively. Additionally, we perform a brick-by-brick comparison of *Kraken* with the weight estimator based on (a) statically assigned function probabilities (*SProb*) and (b) function probabilities that dynamically adapt to changing invocation patterns (*DProb*). These policies use all the components of *Kraken* except *Commonality* and *Connectivity*. Finally, using the simulator we perform different sensitivity studies on varying the load, SLOs and also comparing against an ideal (Oracle) scheme.

6 Analysis of Results

This section presents experimental results for single applications run in isolation for all schemes on the real system and simulation platform. We have also verified that *Kraken* (as well as the other schemes) yield similar results (within 2%) when multiple applications are run concurrently.

6.1 Real System Results

6.1.1 Containers Spawned: Figure 8 depicts the function-wise breakdown of the number of containers provisioned across all policies for individual applications. This represents NC_t^d (Section 3) for all possible depths, d. It can be observed that, existing policies, namely, *Arch*, *Fifer* and *Xanadu* spawn, respectively, 2.41x, 76% and 30% more containers than *Kraken*, on average, across all applications. Overallocation of containers in case of *Arch* is due to two reasons: (i) it assumes that all functions in the application will be invoked at runtime; and (ii) it spawns one container per invocation request. On the other hand, *Fifer* improves upon this by reducing the total number of containers spawned using request batching. However, it does not take workflow activation patterns into consideration while spawning containers, leading to container overprovisioning. The recently proposed scheme, *Xanadu*, is based on a workflow-aware container deployment mechanism, but does not employ request batching, leading to extra containers being deployed in comparison to *Kraken*. Furthermore, it can be seen that *Xanadu* provisions a relatively high number of containers for a particular group of functions as compared to the rest. This is because it allocates containers to serve the predicted load along only the Most Likely Path (MLP) of a request. The rest of the containers are a result of *reactive scaling* that follows from MLP mispredictions, which accounts for 34% of the total number of containers spawned.

The reduction in the number of containers spawned by *Kraken* in comparison to other policies is roughly proportional to the total number of application workflows and the slack available for each function within a workflow (see Table 2 and Figure 7). For instance, Figure 8 indicates that the *Social Network*, *Media Service* and *Hotel Reservation* applications show the highest (73%, 53% and 36%), moderate (40%, 28% and 7%) and least (at most 33%) reductions in the number of containers spawned with respect to existing policies, *Arch*, *Fifer* and *Xanadu*, respectively. Both *Social Network* and *Media Service* have a high number of workflows, but the former has more functions with higher slack, leading to increased batching, thereby resulting in the most reduction in containers spawned. *Hotel Reservation* has the least number of workflows as well as the lowest overall slack for all functions, resulting in the least reduction in the number of containers. On the other hand, *DProb* and *SProb* spawn fewer containers than *Kraken* as a consequence of not using *Commonality* and *Connectivity* to augment function weights, while making container allocation decisions. As a result, *Kraken* provisions up to 21% more containers than both *DProb* and *SProb* for the three applications. Note that, these additional containers are necessary to reduce SLO violations.

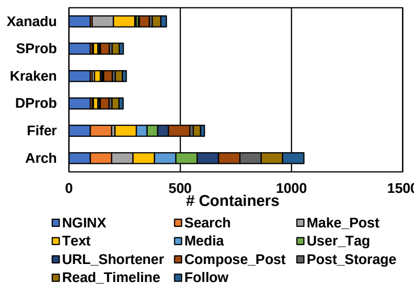
6.1.2 End-to-End Response Times and SLO Compliance: Figure 9 shows the breakdown of the average end-to-end response times and Figure 10 juxtaposes the total number of containers provisioned against the SLO Guarantees for all policies and applications, averaged across all traces. From these graphs, it is evident that *Kraken* exhibits comparable performance to existing policies while having a minimal resource footprint. For the *Social Network* application, *Kraken* remains within 60 ms of the end-to-end response time of *Arch* (Figure 9a), which performs the best out of all policies with respect to these metrics, while ensuring 99.94% SLO guarantees (Figure 10a) . However, *Arch* uses 4x the number of containers used by *Kraken* (Figure 10a).

Kraken also performs similar to *Fifer*, while using 58% reduced containers for *Social Network*. From Figures 9 and 10, it can be seen that *Xanadu* has similar (or worse) end-to-end response times than *Kraken* (up to 50 ms more), but spawns more containers as well (up to 70% more) and satisfies fewer SLOs on average (0.2% lesser). This can be attributed to *Xanadu*'s container pre-deployment policy which causes reactive scale outs as a result of MLP mispredictions. This effect is highlighted in applications such as *Social Network* and *Media Service* which have relatively high MLP misprediction rates (80% and 50%, respectively[2])) due to the presence of multiple possible paths (Table 2). *Media Service* suffers from higher end-to-end response times, further exacerbating this effect. *Xanadu* has only a 34% misprediction rate for *Hotel Reservation*, due to the lower number of workflows, and is seen to match *Kraken* in terms of SLOs satisfied (99.87%).

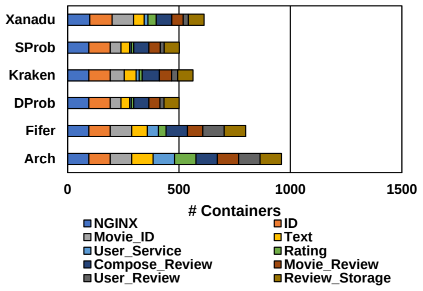
The breakdown of the average response times in Figure 9 shows that both *Arch* and *Xanadu* do not suffer from queueing delays. This is because both policies spawn a container per request, resulting in almost zero queueing. The relatively high cold start-induced delay experienced by *Xanadu* can be attributed to the reactive scaling it uses to cope with MLP mispredictions. *Kraken* exhibits delay characteristics similar to *Fifer* owing to both policies having batching and a similar container pre-deployment policy. However, *Kraken* allocates fewer containers (57% lesser, on average across all applications) along each workflow compared to *Fifer*. *DProb* and *SProb* exhibit higher overall end-to-end response times compared to *Kraken*, with *SProb* experiencing a disproportionately high queueing delay compared to its cold start delay. This is because it uses statically assigned function weights, which prevents it from being able to proactively spawn containers according to the varying user input. This results in the majority of requests getting queued at the containers.

6.1.3 Analysis of Key Improvements: This subsection focuses on the key improvements offered by *Kraken* in terms of Container Utilization, Response Latency Distribution and Energy Efficiency. Although we use specific combinations of applications and traces to highlight the improvements, the

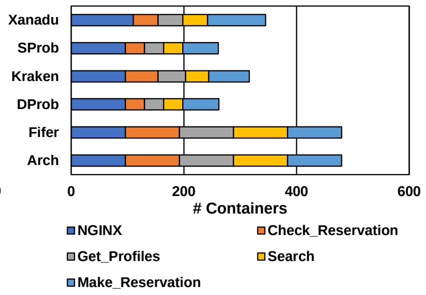
[2]MLP misprediction rates are not shown in any Figure



(a) Social Network.

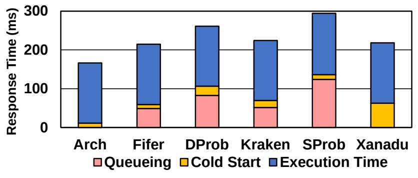


(b) Media Service.

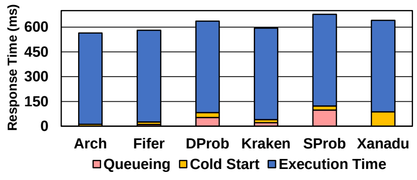


(c) Hotel Reservation.

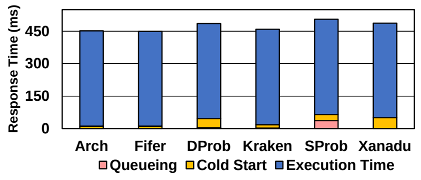
Figure 8. Real System: Stage-wise Breakdown of Containers spawned by each policy.



(a) Social Network.

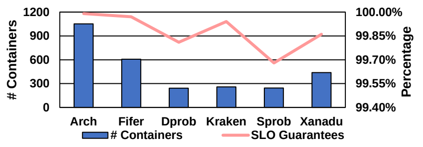


(b) Media Service.

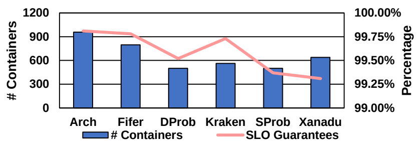


(c) Hotel Reservation.

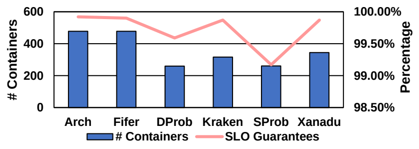
Figure 9. Real System: Breakdown of Average End-to-End Response Times in terms of queueing delay, cold start delay and execution time.



(a) Social Network.

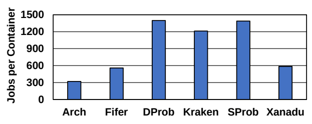


(b) Media Service.

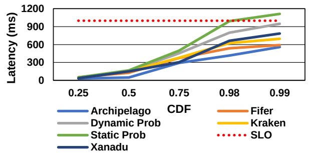


(c) Hotel Reservation.

Figure 10. Real System: Comparison of Total Number of Containers spawned VS SLOs satisfied by each policy. The Primary Y-Axis denotes the number of containers spawned, The secondary Y-axis indicates the percentage of SLOs met and the X-axis represents each policy.



(a) Container Utilization.



(b) Response Time Distribution.

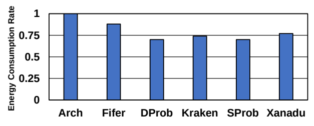
Figure 11. Real System: Comparison of Container Utilization (a.k.a. average #jobs executed per Container) and Response Time Distribution.

results are similar for other workload mixes as well.

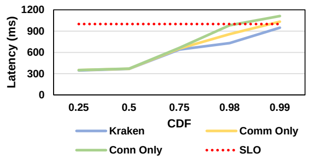
Container Utilization: Figure 11a plots the average number of requests executed per container (Jobs per container) across all functions in *Social Network* for the Poisson trace. An ideal scheme would focus on packing more number of requests per container to improve utilization without causing SLO violations. *Kraken* shows 4x, 2.16x and 2.06x more container utilization compared to *Arch*, *Fifer*, and *Xanadu* respectively. This is because *Kraken* limits the number of containers spawned through function weight assignment and request batching. *DProb* and *SProb* both exhibit higher utilization compared to *Kraken* (15%) as a result of spawning fewer containers overall, owing to not accounting for *critical* and *common* functions while provisioning containers. Consequently, they exhibit up to 0.24% more SLO Violations compared to *Kraken*, for this workload mix.

Latency Distribution: The end-to-end latency distribution for all policies for the *Social Network* application with the Twitter trace is plotted in Figure 11b. In particular, *Arch*, *Fifer* and *Kraken* show comparable latencies, with P99 values within 700ms while remaining well within the SLO of 1000ms. However, *Arch* and *Fifer* use 3.51x and 2.1x more containers than *Kraken* to achieve this. The tail latency (measured at P99) for *DProb* almost exceeds the SLO, whereas it does so for *SProb*. *Kraken* manages to avoid high tail latency by assigning augmented weights to key functions, thus, helping it tolerate incorrect load/probability estimations. *SProb* does worse than *DProb* at the tail because of its lack of adaptive probability estimation. *Kraken* makes use of 21% more containers to achieve the improved latencies. *Xanadu* experiences a sudden rise in latency towards the tail, with its P99 latency being 100ms more than that of *Kraken*, while using 96% more containers. This is due to *Xanadu*'s MLP misprediction and the resultant container over-provisioning.

Energy Efficiency: We measure the energy-consumption as total Energy consumed divided over total time. *Kraken* achieves one of the lowest energy consumption rates among all the policies considered, with it bettering existing policies, namely, *Arch*, *Fifer* and *Xanadu* by 26%, 14% and 3% respectively (for the workload mix of *Media Service* application with Wiki trace) as depicted in Figure 12a. These savings



(a) Energy Consumption Rate.



(b) Response Time Distribution.

Figure 12. Real System: Normalized Energy Consumption of all Schemes and Response Time Distribution of *Kraken*, *Comm Only* and *Conn Only*

Application	*Kraken*	*Comm Only*	*Conn Only*
Social Network	(99.94%, 284)	(99.91%, 276)	(99.89%, 256)
Media Service	(99.73%, 572)	(99.66%, 561)	(99.64%, 552)
Hotel Reservation	(99.87%, 316)	(99.77%, 290)	(99.74%, 282)

Table 5. Real System: Comparing (SLO Guarantees,#Containers Spawned) against *Comm Only* and *Conn Only*.

can go up to 48% compared to *Arch* for applications like *Social Network*. The resultant energy savings of *Kraken* are a direct consequence of the savings in computation and memory usage from the fewer containers spawned. Only *DProb* and *SProb* consume lesser energy than *Kraken* (4% lesser), due to their more aggressive container reduction approach.

6.1.4 Ablation Study:

This subsection conducts a brick-by-brick evaluation of *Kraken* using *Conn Only* and *Comm Only*, schemes that exclude *Commonality* and *Connectivity* components from *Kraken*, respectively. From Table 5, it can be seen that *Comm Only* spawns 8% more containers than *Conn Only* for *Social Network*. This difference is lesser for the other applications. Upon closer examination, we see that this is due to functions having different degrees of *Commonality* and *Connectivity*. Moreover, the majority of functions whose *Commonality* and *Connectivity* differ, have a high batch size, thereby reducing the variation in the number of containers spawned. Following this, we observe that the variation in the number of containers in *Social Network* is mainly due to the significant difference in the *Commonality* and *Connectivity* of the *Compose Post* function whose batch size is only one. There is lesser difference in containers spawned by *Comm Only*, *Conn Only* and *Kraken* for *Media Service* because we have implemented *Kraken* with a cap on the additional containers spawned due to *Commonality* and *Connectivity* when the sum of their values exceeds a threshold. This threshold is exceeded in *Media Service* for the majority of functions. Due to the difference in container provisioning, the difference in response times between the three schemes is evident at the tail of the response time distribution (Figure 12b). *Comm Only* and *Conn Only* are seen to exceed the target SLO at the 99th percentile. The tail latency of *Kraken*, in comparison, grows slower and remains within the target SLO.

6.2 Simulator Results

Since the real-system is limited to a 160-core cluster, we use our in-house simulator, which can simulate an 11k-core cluster, to study the scalability of *Kraken*. We mimic a large scale Poisson arrival trace (μ = 1000rps), Wiki (μ = 284 rps) and Twitter (μ = 3332 rps) traces. Figure 13 plots the containers spawned versus the SLO guarantees for each application for all traces. The simulator results closely correlate to those of the real system. *Kraken* is seen to reduce container overprovisioning when applications have numerous possible workflows and enough slack per function to exploit. Notably, *Kraken* spawns nearly 80% less containers for *Social Network* in comparison to *Arch*. Container overprovisioning is inflated 15% more than the corresponding real system result, due to the large-scale traces. Table 6 shows the median and tail latencies of each policy averaged across all applications for the three traces. The trend we observe is that traces with higher variability, such as the Twitter trace, affect the tail latencies of policies more harshly than the other, more predictable, traces. Nevertheless, *Kraken* is resilient to unpredictable loads as well, with tail latencies always remaining within the SLO (1000 ms). However, the tail latencies of *DProb* and *SProb* sometimes exceeds the SLO, since they don't use *Commonality* and *Connectivity*. It is observed that *Xanadu* also violates the SLO for the Twitter trace, owing to the reactive scale-outs resulting from MLP mispredictions.

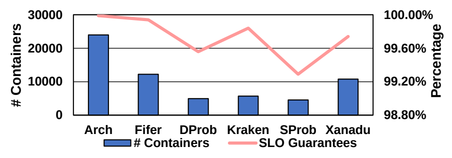
Policy	**Poisson**		**Wiki**		**Twitter**	
	Med	**Tail**	**Med**	**Tail**	**Med**	**Tail**
Arch	336	568	336	568	336	599
Fifer	362	612	360	611	373	833
DProb	371	746	368	753	381	1549
Kraken	366	634	358	633	371	974
SProb	395	1101	382	1073	395	1610
Xanadu	343	723	340	774	340	1244

Table 6. Simulator: Median and tail latencies (in ms) averaged across all applications for the three traces

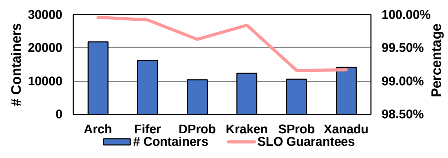
6.2.1 Sensitivity Study:

This subsection compares *Kraken* against *Oracle*, which is an ideal policy that is assumed to be able to predict future load and all path probabilities with 100% accuracy and also has request batching. Consequently, *Oracle* does not suffer from cold starts and minimizes containers spawned. Figure 14 shows the breakdown of total number of containers spawned for each application, averaged across all realistic large-scale traces using the simulator. It is observed that *Kraken* spawns more containers (7%) than *Oracle*, on average. This is due to *Kraken*'s load/path probability miscalculations and the usage of *Commonality* and *Connectivity* to cope with this. It is seen that *Kraken* spawns 10% more containers for *Media Service* and 6% more for *Hotel Reservation* and *Social Network*. This may be due to *Media Service* having higher path unpredictability than *Hotel Reservation* (Table 2) as well as lower slack per function than *Social Network* (Figure 7). From Figure 15b, it is observed that *Oracle*, being clairvoyant, spawns containers in accordance with the peaks and valleys of the request arrival trace. *Kraken*, while spawning more containers, also is seen to lag behind the trend of the trace due to load prediction errors.

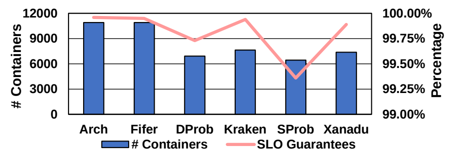
Performance under Sparse Load: Analysis of logs collected from the Azure cloud platform [39] shows request



(a) Social Network.

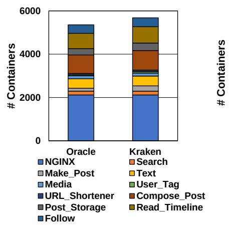


(b) Media Service.

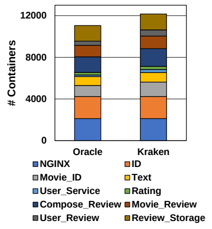


(c) Hotel Reservation.

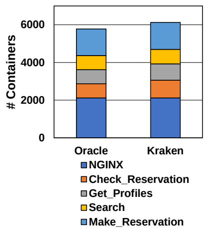
Figure 13. Simulator: Comparison of Total Number of Containers spawned VS SLOs satisfied by each policy. The Primary Y-Axis denotes the number of containers spawned, The secondary Y-axis indicates the percentage of SLOs met and the X-axis represents each policy.



(a) Social Network.

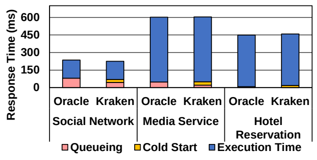


(b) Media Service.

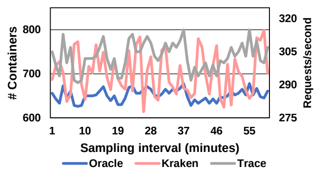


(c) Hotel Reservation.

Figure 14. Simulator: Comparison of Function-wise Breakdown of Containers spawned by *Kraken* and *Oracle*.



(a) E2E Response Time Breakdown.



(b) Containers spawned over time.

Figure 15. Simulator: Comparison of End-to-End (E2E) Response Times and Containers Spawned Over Time (60 minutes) of *Kraken* and *Oracle*.

volumes that are much lighter (average of 2 requests/hour) than those of the traces we have considered. Moreover, more than 40% of requests show significant variability in inter-arrival times. To deal with such traces, we modified *Kraken*'s load prediction model to predict future request arrival times, owing to the sparse nature of the trace. We also spawn containers much more in advance than the predicted arrival time and also keep them alive for at least a minute before evicting them from memory, to account for arrival unpredictability. It is seen that *Kraken* meets the SLOs for all requests from the lightly-loaded trace over 18 hours while averaging 0.85 memory-resident containers at any given second[3]. Other existing policies such as *Arch* and *Fifer* exhibit similar performance and resource usage when their prediction models and keep-alive times are similarly adjusted. *Xanadu*, on the other hand, while having 0.74 memory-resident containers per second, suffers from 55% SLO Violations on average across all applications as a result of MLP mispredictions whose effects are exacerbated in this scenario, due to low request volume.

Varying SLO: Table 7 shows the SLO guarantees and number of containers spawned for existing policies as well as *Comm Only* and *Conn Only*, when the SLO is reduced from 1000ms to a value 30% higher than the response time of the slowest workflow in each application. The resultant SLOs

Trace	*Arch*	*Fifer*	*Kraken*	*Xanadu*	*Comm Only*	*Conn Only*
Wiki	(99.91%, 2737)	(99.90%, 2092)	(99.86%, 1396)	(99.66%, 1737)	(99.78%,)	(99.75%,)
Twitter	(99.72%, 45,107)	(99.63%, 34,210)	(99.50%, 22,377)	(99.10%, 25,132)	(99.22%,)	(99.15%,)

Table 7. Simulator: Comparing (% SLO met,# Containers Spawned) against Existing Policies after Varying the Target SLOs.

are 500ms, 910ms and 809ms for *Social Network*, *Media Service* and *Hotel Reservation* respectively. Reducing the SLO, in turn, can potentially reduce the batch sizes of functions as well. Moreover, the reduced SLO target results in increased SLO violations across all policies. However *Kraken*, is able to maintain at least 99.5% SLO guarantee and spawns 50%, 34% and 15% less containers compared to *Arch*, *Fifer* and *Xanadu*, respectively. It can be seen that the difference in SLO compliance between *Kraken*, *Comm Only*, and *Conn Only* increases due to the reduced target SLO. This difference, in terms of percent of SLO violations, changes from being at most 0.1% to being between 0.1 to 0.35%. This is a result of *Kraken* being more resilient at the tail of the response time distribution as it uses both *Commonality* and *Connectivity* while spawning containers. In comparison, *Comm Only* and *Conn Only* fail to spawn enough containers for each important function as they do not consider both these parameters, resulting in increased tail latency and further exacerbates the SLO violations.

7 Concluding Remarks

Adopting serverless functions for executing microservice-based applications introduces critical inefficiencies in terms of scheduling and resource management for the cloud provider, especially when deploying Dynamic DAG Applications. Towards addressing these challenges, we design and evaluate *Kraken*, a DAG workflow-aware resource management framework, for efficiently running such applications by utilizing minimum resources, while remaining SLO-compliant. *Kraken* employs proactive weighted scaling of functions, where the weights are calculated using function invocation probabilities and other parameters pertaining to the application's DAG structure. Our experimental evaluation on a 160-core cluster using *Deathstarbench* workload suite and real-world traces demonstrate that *Kraken* spawns up to 76% fewer containers, thereby improving container utilization and cluster-wide energy savings by up to 4× and 48%, respectively, compared to state-of-the art schedulers employed in serverless platforms.

[3]These results are not shown in any graph.

References

[1] Twitter stream traces. https://archive.org/details/twitterstream. Accessed: 2020-05-07.

[2] Airbnb AWS Case Study., 2019. https://aws.amazon.com/solutions/case-studies/airbnb/.

[3] Provisioned Concurrency., 2019. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[4] Amazon States Language., 2020. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html.

[5] AWS Lambda. Serverless Functions, 2020. https://aws.amazon.com/lambda/.

[6] Azure Durable Functions., 2020. https://docs.microsoft.com/en-us/azure/azure-functions/durable.

[7] hey HTTP Load Testing Tool., 2020. https://github.com/rakyll/hey.

[8] IBM-Composer., 2020. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_composer.

[9] Kubernetes., 2020. https://kubernetes.io/.

[10] Microsoft Azure Serverless Functions, 2020. https://azure.microsoft.com/en-us/services/functions/.

[11] Openfaas. https://www.openfaas.com/, 2020.

[12] Prometheus., 2020. https://prometheus.io/.

[13] AWS Lambda Cold Starts., 2021. https://mikhail.io/serverless/coldstarts/aws/.

[14] Azure Functions Cold Starts., 2021. https://mikhail.io/serverless/coldstarts/azure/.

[15] Expedia Case Study - Amazon AWS., 2021. https://mikhail.io/serverless/coldstarts/azure/.

[16] Intel Power Gadget, Feb 24, 2020. https://github.com/sosy-lab/cpu-energy-meter.

[17] Google Cloud Functions, February 2018. https://cloud.google.com/functions/docs/.

[18] Akkus, I. E., et al. Sand: Towards high-performance serverless computing. In *ATC* (2018).

[19] Awad, M., Khan, L., and Thuraisingham, B. Predicting www surfing using multiple evidence combination. *The VLDB Journal 17*, 3 (2008), 401–417.

[20] Awad, M. A., and Khalil, I. Prediction of user's web-browsing behavior: Application of markov model. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 42*, 4 (2012), 1131–1142.

[21] Begleiter, R., El-Yaniv, R., and Yona, G. On prediction using variable order markov models. *Journal of Artificial Intelligence Research 22* (2004), 385–421.

[22] Brooker, M., Florescu, A., Popa, D.-M., Neugebauer, R., Agache, A., Iordache, A., Liguori, A., and Piwonka, P. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI* (2020).

[23] Buddha, J. P., and Beesetty, R. Step functions. In *The Definitive Guide to AWS Application Integration.* Springer, 2019.

[24] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[25] Carver, B., Zhang, J., Wang, A., and Cheng, Y. In search of a fast and efficient serverless dag engine. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)* (2019), IEEE, pp. 1–10.

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

[27] Gagniuc, P. A. *Markov chains: From Theory to Implementation and Experimentation.* John Wiley & Sons, 2017.

[28] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 3–18.

[29] Gujarati, A., Elnikety, S., He, Y., McKinley, K. S., and Brandenburg, B. B. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *USENIX Middleware Conference* (2017).

[30] Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., and Das, C. R. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 280–295.

[31] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[32] Kannan, R. S., Subramanian, L., Raju, A., Ahn, J., Mars, J., and Tang, L. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *EuroSys* (2019).

[33] Korte, B., and Vygen, J. Bin-packing. In *Combinatorial Optimization.* Springer, 2018, pp. 489–507.

[34] Kuhlenkamp, J., Werner, S., and Tai, S. The ifs and buts of less is more: a serverless computing reality check. In *2020 IEEE International Conference on Cloud Engineering (IC2E)* (2020), IEEE, pp. 154–161.

[35] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[36] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX ATC* (2018).

[37] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 805–825.

[38] Shahrad, M., Balkind, J., and Wentzlaff, D. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), pp. 1063–1075.

[39] Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)* (2020), pp. 205–218.

[40] Silva, P., Fireman, D., and Pereira, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[41] Singhvi, A., Houck, K., Balasubramanian, A., Shaikh, M. D., Venkataraman, S., and Akella, A. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849* (2019).

[42] Taibi, D., El Ioini, N., Pahl, C., and Niederkofler, J. R. S. Patterns for serverless functions (function-as-a-service): A multivocal literature review. In *CLOSER* (2020), pp. 181–192.

[43] Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., and Lanka, S. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 311–327.

[44] Urdaneta, G., Pierre, G., and Van Steen, M. Wikipedia workload analysis for decentralized hosting. *Computer Networks* (2009).

[45] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *ATC* (2018).

[46] Yang, H., Chen, Q., Riaz, M., Luan, Z., Tang, L., and Mars, J. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *Computer Architecture News* (2017).

[47] Zhang, Y., Crowcroft, J., Li, D., Zhang, C., Li, H., Wang, Y., Yu, K., Xiong, Y., and Chen, G. Kylinx: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference* (2018), pp. 173–186.