ChunkFunc: Dynamic SLO-Aware Configuration of Serverless Functions

Thomas Pusztai and Stefan Nastic, *Member, IEEE*

***Abstract*—Serverless computing promises to be a cost effective form of on demand computing. To fully utilize its cost saving potential, workflows must be configured with the appropriate amount of resources to meet their response time Service Level Objective (SLO), while keeping costs at a minimum. Since determining and updating these configuration models manually is a nontrivial and error prone task, researchers have developed solutions for automatically finding configurations that meet the aforementioned requirements. However, our initial experiments show that even when following best practices and using state-of-the-art configuration tools, resources may still be considerably over- or underprovisioned, depending on the size of functions' input payload. In this paper we present ChunkFunc, an SLO- and input data-aware framework for tuning serverless workflows. Our main contributions include: i) an SLO- and input size-aware function performance model for optimized configurations in serverless workflows, ii) ChunkFunc Profiler, an auto-tuned, Bayesian Optimization-guided profiling mechanism for profiling serverless functions with typical input data sizes to build a performance model, and iii) ChunkFunc Workflow Optimizer, which uses these models to determine an input size dependent configuration for each serverless function in a workflow to meet the SLO, while keeping costs to a minimum. We evaluate ChunkFunc on real-life serverless workflows and compare it to two state-of-the-art solutions, showing that it increases SLO adherence by a factor of 1.04 to 2.78, depending on the workflow, and reduces costs by up to 61% .**

***Index Terms*—Serverless workflows, serverless functions, configuration tuning, SLOs, profiling.**

I. Introduction

ALL major Cloud platforms provide serverless offerings [1], [2], [3], [4] and their usage is continuously growing. In a 2023 survey, Datadog reports that over 70% of its AWS customers and 60% of its Google Cloud customers use at least one serverless solution [5]. Serverless computing provides the advantage that developers can focus on the business logic of their functions and leave scaling and most infrastructure management decisions to the Cloud provider. Typically, developers only configure the amount of memory that should be allocated to a function. The memory maps to a predefined resource profile, which contains a fixed amount of virtual CPU cores (vCPUs) – we adopt the same convention for our work. Despite the seeming simplicity of configuring serverless resources, tuning the amount of memory, vCPUs, and configuration models to ensure that Service Level Objectives (SLOs) are met, while minimizing the costs still remains a challenge [6], [7].

Received 6 June 2024; revised 27 March 2025; accepted 28 March 2025. Date of publication 9 April 2025; date of current version 24 April 2025. This work was supported by Austrian Research Promotion Agency (FFG) through the project RapidREC under Project 903884 and in part by Austrian Internet Stiftung NetIdee under Grant 7442. *(Corresponding author: Thomas Pusztai.)*

The authors are with the Distributed Systems Group, TU Wien, 1040 Vienna, Austria (e-mail: t.pusztai@dsg.tuwien.ac.at; snastic@dsg.tuwien.ac.at).

Digital Object Identifier 10.1109/TPDS.2025.3559021

A. Tuning of Serverless Workflow Configurations

Tuning resource configurations of serverless workflows to meet SLOs is typically done using performance models for the comprising functions. There are two main types of approaches: 1) a-priori profiling of functions to build a performance model in an offline fashion and 2) monitoring of function executions at runtime to build the performance model in an online fashion.

1) A-priori profiling systems normally execute functions under varying resource configurations with typical input data to build a performance profile.
   This is used to configure the function's resources in production to meet the defined Service Level Objective (SLO).
   Most systems that tune entire workflows rely on graph algorithms [8], [9], [10]. Another approach is the use of a max-heap [11]. For a single function or job, linear, binary, and gradient descent search [12], Bayesian Optimization (BO) [13], and CPU time accounting [14] have been used. Two common drawbacks of a-priori profiling systems are that a "typical workload" needs to be defined and the tedious profiling process itself. Finding a typical workload might not be possible for functions that have highly variable inputs, such as those used for log or video processing. Profiling often needs to be done manually and/or takes a long time if all resource profiles need to be tested exhaustively. Some approaches reduce the number of profiling runs, e.g., using BO, but they require manual tuning of parameters to get accurate results.
2) Systems that build a performance model online rely on historical or live monitoring data of function executions. Some approaches passively monitor execution [12], [15], [16]. Others assign different configurations until the performance model is complete [17], [18], often relying on statistical methods, such as Bayesian Optimization, to reduce the number of configurations that need to be explored. However, until the performance model is complete, these approaches may violate the SLO. Thus, to have good SLO adherence from the first day in production, developers need to issue many requests to allow the model to train, which is essentially similar to profiling.

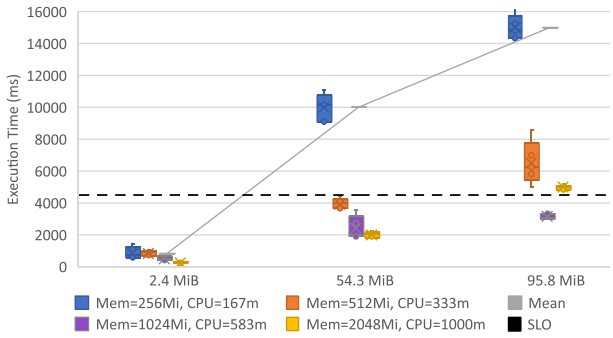


Fig. 1. `Extract-successes` response times under various input data sizes and resource configurations.

Additionally, resources for collecting and processing monitoring data during the entire application lifetime to update the performance model may incur additional costs.

B. Motivation

Current approaches often overprovision resources [19], [20] and do not account for input data size variations, which leads to problems with highly heterogeneous workloads, because different input sizes may result in different performance under various resource configurations [21]. A common use case that deals with varying input data sizes is logs processing, e.g., hourly logs processing of a bank gets more data during the day than at night. Other examples include video processing (varying lengths, resolutions, and bitrates), malware scanning (varying file sizes), and continuous integration workflows in software engineering (varying repository sizes).

To further explore the need for input data size awareness, we run an experiment with a serverless function on Knative[1]. The `extract-successes` function extracts success messages from logs of a real distributed cluster scheduler [22]. We feed three log sizes (2.4, 54.3, and 95.8 MiB) to the function under four resource configurations (256, 512, 1024, and 2048 MiB) and measure the response time within the function itself, i.e., it is not affected by cold starts. Each combination is executed five times, the results are shown in Fig. 1.

We define a *maximum response time (MRT)* SLO of 4,500 ms, indicated by the black dashed line in Fig. 1. We observe that the mean response time across all resource configurations (gray line) increases with the input data size. Furthermore, we see that for meeting the SLO, different resource configurations may be used for different input data sizes. For the smallest input of 2.4 MiB, all four configurations meet the SLO, so the lowest (and cheapest) resource configuration with 256 MiB memory is sufficient. For the medium input of 54.3 MiB, only three configurations meet the SLO, with the lowest possible being 512 MiB memory. For the large input of 95.8 MiB, only the 1024 MiB configuration, meets the SLO, while the highest configuration violates the SLO. In theory, however, the single-threaded `extract-successes` Node.JS function should perform best with at least one CPU core, i.e., the 2048 MiB configuration or higher. Further investigation with a single resource-constrained Docker container showed that this behavior is specific to running the function in our target Kubernetes cluster and is caused by an interplay of the execution environment, the Node.JS IO thread pool, and the function structure. The automatic profiling results in our later experiments confirm that in the target cluster the 1024 MiB configuration is the fastest for the 95.8 MiB input.

Preliminary findings: The initial experiments show two important correlations for many serverless functions: i) when the input data size increases, the response time increases too and ii) for *a given input data size, a different resource configuration may increase or decrease the response time*. Consequently, there is usually not a single resource configuration that is ideal to meet a function's SLO and minimize its cost, but different resource configurations, depending on the input data size of an invocation. While there are exceptions, e.g., image labeling with almost constant runtime, many applications, like the previous examples, exhibit these correlations and, thus, benefit from input data size-aware resource configuration.

Shortcomings of the state-of-the-art: Most existing systems have at least one of two major shortcomings:

1) they do not consider the size of the input data when choosing a resource profile for a function and/or
2) building the performance model for a function is a tedious, long profiling process or requires observing the live system for a long time.

Most systems disregard the input data size when assigning a resource profile to a function, e.g., [8], [9], [10], [11], [17], [23]. This can result in SLO violations if a production input is substantially larger than the one(s) used for building the performance model and in excessive costs if the input is smaller than expected by the model. Building the performance model through profiling or by observing the live system requires time. Some approaches try to shorten that time, e.g., using Bayesian Optimization [17], [24] or regression [15] to reduce the amount of observations needed to build the performance model. However, to the best of our knowledge, none of these approaches account for different input sizes. With the contributions of this paper, we address both of these shortcomings.

C. Contributions

In this paper, we present ChunkFunc, a framework that dynamically adapts resource configurations of serverless functions, based on their input data size (payload) and reduces costs, while ensuring that the SLOs of the entire workflow are met. ChunkFunc is part of Polaris SLO Cloud[2], a SIG of the Linux Foundation Centaurus project[3], a platform for building unified and highly scalable distributed Cloud and Edge systems. Specifically, the main contributions include:

1) **An SLO- and input data size-aware function performance model** for determining optimized configurations in serverless workflows, depending on the input data size (Section II).

[1]https://knative.dev

[2]https://polaris-slo-cloud.github.io

[3]https://www.centaurusinfra.io

2) ***ChunkFunc Profiler*, which automatically builds performance models for serverless functions and workflows** based on typical input data sizes. Profiling is automatic, users only deploy a function and specify typical input data. A novel, auto-tuned BO approach reduces the profiling costs by up to 90% compared to exhaustive profiling and ensures high accuracy of the results. Contrary to state-of-the-art BO approaches we reuse the Gaussian Process (GP) of the BO to infer missing parts of our performance model (Section III).
3) ***ChunkFunc Workflow Optimizer*, which leverages various heuristics to dynamically adapt the resource configuration of functions in a workflow to meet a performance-based SLO** (e.g., response time), while minimizing cost. Unlike existing systems, the ChunkFunc Workflow Optimizer considers the size of a function's input when selecting a resource profile, which, depending on the workflow, increases SLO adherence by a factor of 1.04 to 2.78 and reduces costs by up to 61% The Workflow Optimizer is extensible with arbitrary performance-based SLOs (Section IV).

II. ChunkFunc System Model & Optimization Problem

A. ChunkFunc System Model

A serverless workflow consists of functions chained together in sequence, in parallel, or in a combination of both, can be represented as a directed acyclic graph (DAG) $W = G(F, E)$. The set of nodes consists of the functions of the workflow, i.e., $F = \{f_0, f_1, \ldots, f_n\}$, and the set of edges $E = \{(f_i, f_j), \ldots\}$ consists of the invocation relationships among those functions. A directed edge (f_i, f_j) indicates that f_j is invoked with the output of f_i. The input to a function f_i is denoted as x_i and its size as $|x_i|$. The size of the output $f_i(x_i)$ depends on the particular function and, typically, it cannot be determined from the input data size without executing the function. The same input data size may yield different output sizes, e.g., the output size of a function that extracts error messages from a 1 GB log file depends on how many error messages the file contains.

Each function instance is assigned a set of resources, such as CPU and memory, which are defined in a resource profile p. The set RP contains all resource profiles that are available on the underlying serverless platform. Typically, commercial Cloud providers, allow users to only choose the amount of memory that should be assigned – each memory size is associated with a predefined number of CPU cores or fraction of CPU cores [25], [26]. We denote an instance of function f deployed with resource profile p as f^p. As noted in Section I-B, serverless functions often exhibit a different performance for different resource profiles. Thus, we denote the SLO metric of f^p, when executed with input x as $M_{SLO}(f^p, x)$. This metric can be the response time or another metric that corresponds to the desired SLO. Each resource profile has a cost associated per unit of execution time. $C(f^p, x)$ expresses the cost incurred by executing f with input x, when it is deployed with resource profile p. We observe that, given a function f to be invoked with input x, the SLO metric value and cost of this invocation depend on the chosen resource profile p.

TABLE I
Symbols Used in the System Model

Symbol	*Definition*
f	Serverless Function
p	Resource Profile
RP	All resource profiles p that are available on the underlying serverless platform
f^p	f deployed with resource profile p
$\|x\|$	Size of input data x
$M_{SLO}\left(f^p, x\right)$	SLO metric value of f^p, when executed with input x
$C\left(f^p, x\right)$	Cost of executing f^p with input x
PP_f	Performance profiles for f $PP_f = \bigcup_{\forall x \in X_f} \{(M_{SLO}\left(f^p, x\right), C\left(f^p, x\right))\}$
$W = G\left(F, E\right)$	Workflow DAG with $F = \{f_0, f_1, \ldots, f_n\}$ $E = \{(f_i, f_j), \ldots\}$
s_W	SLO for the entire workflow W
RP_W	Selected resource profiles $\forall f \in W$
X_W	Set of input sizes $\forall f \in W$
$M\left(W, RP_W, X_W\right)$	SLO metric value for executing W with inputs X_W under resource profiles RP_W
$C\left(W, RP_W, X_W\right)$	Cost of executing W with inputs X_W under resource profiles RP_W

The pair $(M_{SLO}(f^p, x), C(f^p, x))$ constitutes a performance profile for f under the resource profile p. The performance profiles for all typical inputs for f are collected in the set PP_f.

In addition to functions, a complex workflow may contain branch statements or loops. For the sake of simplicity, we consider these constructs also as functions within our optimization problem, albeit with special properties. Branch functions always have an SLO metric and cost of zero and loop functions wrap another function. The SLO metric value and cost of the loop function is equal to that of the wrapped function, multiplied by the number of loop iterations, which is known only at runtime. We denote the SLO of the entire workflow W as s_W.

For clarity, all symbols used in the system model and the optimization problem are summarized in Table I.

B. Optimization Problem

The ChunkFunc optimization problem aims to find a set of resource profiles to deploy the functions of a workflow, given a particular input, while ensuring that the SLO is met and the cost of the workflow execution is minimized.

We use $RP_W = \{(f_0, p_0), (f_1, p_1), \ldots, (f_n, p_n)\}$ to denote the set of resource profiles that have been chosen to spawn the function instances in a particular execution of W, such that p_i is used to spawn f_i.

Let $X_W = \{|x_0|, |x_1|, \ldots, |x_n|\}$ be the set of input sizes to the functions of an execution of W, such that x_i is the input to f_i. The only element of X_W that is known at the beginning of the workflow is the input to the first function x_0; the remaining elements are added as the workflow progresses.

To enforce the SLO for a workflow W, we need the SLO metric value of a particular workflow execution, given the set of chosen resource profiles RP_W and function inputs X_W. It is calculated by aggregating all function SLO metric values:

$$M_{SLO}\left(W, RP_W, X_W\right) = \underset{f_i \in W}{\Delta} M_{SLO}\left(f_i^{p_i}, x_i\right) \quad (1)$$

Based on the type of SLO metric the semantics of the aggregation operator Δ change. There are two types of SLO metrics:

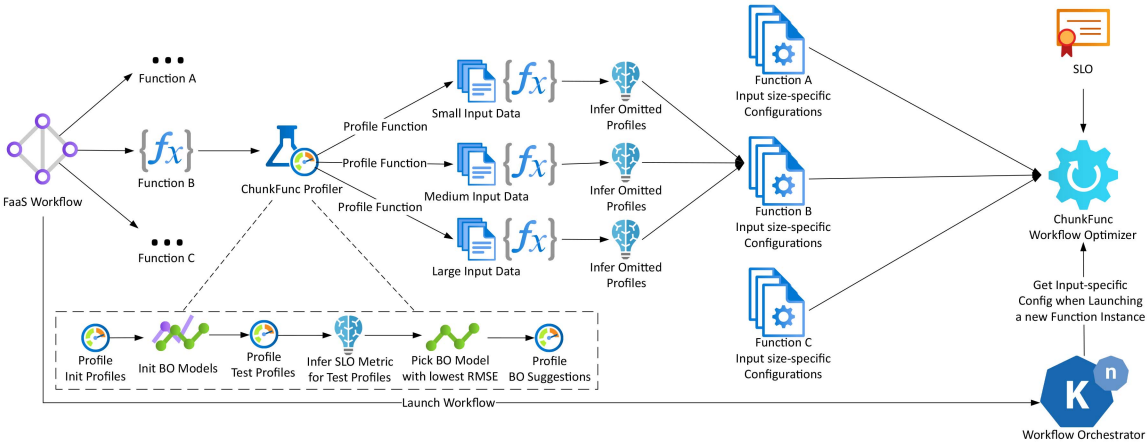


Fig. 2. Overview of the ChunkFunc System and Lifecycle of a Serverless Workflow.

i) *additive metrics*, such as response time, which are summed along a path in the workflow ($\Delta = \sum$) and ii) *min-metrics*, such as throughput, where the minimum of all edges in a path is taken ($\Delta = \min$).

We compute the total cost of the workflow execution, by summing the costs of its function executions:

$$C\left(W, RP_W, X_W\right) = \sum_{f_i \in W} C\left(f_i^{p_i}, x_i\right) \tag{2}$$

The optimization problem consists in finding a set $RP_W \subset F \times RP$ for an input set X_W that fulfills constraints (3) and (4). The former is a hard constraint and establishes the relation between the SLO metric value of the workflow and the SLO s_W. Depending on the type of SLO, $\lessdot$ is typically either $\leq$ (e.g., for response time) or $\geq$ (e.g., for throughput). The latter is a soft constraint that seeks to minimize the total cost of the workflow execution.

$$M\left(W, RP_W, X_W\right) \lessdot s_W \tag{3}$$

$$\min C\left(W, RP_W, X_W\right) \tag{4}$$

This optimization problem is NP-hard and the fact that only the first input size is known at the beginning of the workflow execution further complicates finding a solution. Any function's input, other than f_0's, is only known once all of its immediate predecessors have executed.

For example, consider a simple, sequential workflow with two functions, f_0 and f_1, and an MRT SLO of $s_W = 80$ ms. For the input data size $|x_0|$ the function f_0 takes 50 ms when deployed with the cheap resource profile p_0 and 25 ms when deployed with the expensive resource profile p_0'. The output of f_0 will either be small (x_s) or large (x_l). For a small input x_s the function f_1 takes at most 20 ms, while for a large input x_l, it takes at least 40 ms. Thus, when selecting a resource profile for f_0, the circumstance that we do not know the size of its output does not allow us to select the cheap resource profile with an execution time of 50 ms, because if the output happens to be large, f_1 will run for at least 40 ms, leading to a total response time of 90 ms, which violates the SLO. Since elements of X_W, are missing when the workflow is invoked, we cannot find an exact solution to the optimization problem at this point. However, we can approximate a solution using a heuristic, which we describe in Section IV.

III. ChunkFunc Framework Overview & Profiler

The ChunkFunc framework consists of two major components: the Profiler and the Workflow Optimizer. In this section, we first present an overview of the system and then describe the Profiler.

A. Framework Overview

Fig. 2 presents an overview of ChunkFunc and the lifecycle of a serverless workflow within the system. Upon their deployment, serverless functions are automatically picked up by the ChunkFunc Profiler. It deploys function instances using various resource configurations to execute profiling runs with their typical input data sizes, without any user interaction. To reduce the number of profiling runs, while maintaining a high accuracy of the results, the choice of resource configurations is guided by Bayesian Optimization. Our BO Dynamic Hyperparameter Selection picks the hyperparameter that yields the most accurate results for a particular function type and input size combination. Finally, the input-specific performance profiles are leveraged by the ChunkFunc Workflow Optimizer, which provides a suitable resource profile, to meet the workflow's SLO and minimize cost, to the serverless orchestrator prior to invoking a function.

B. ChunkFunc Bayesian Optimization-based Profiler

The ChunkFunc Profiler automatically creates input data size-specific performance profiles for every deployed serverless function. The user only needs to specify several typical input data payloads (normally two or three) of different sizes for the function as ChunkFunc-specific metadata. For each defined typical input data size, a distinct performance profile is computed fully automatically by ChunkFunc.

While exhaustively profiling the function under every resource profile is supported, it can incur high costs. Thus, we leverage Bayesian Optimization (BO) to reduce the number of profiling runs. BO is a technique that is normally used to find

the maximum of an unknown objective function, based on a limited set of samples [27]. It builds a surrogate model, typically using a GP, to approximate the objective function using known samples and uses an acquisition function to guide the exploration of further samples.

BO is deeply integrated into the ChunkFunc profiler in two ways. First, we use the acquisition function to determine the most promising input data sizes to profile, similar to [17], [24]. Second, we leverage the surrogate model to infer the SLO metric for those input data sizes that were not profiled. This use of BO to infer the missing parts of the performance profile has, to the best of our knowledge, not been attempted by the state-of-the-art. Our BO-based profiler allows for an up to 90% reduction of profiling costs.

Common choices for the acquisition function of BO include Probability of Improvement (POI) [27] and Expected Improvement (EI) [28], [29]. POI returns the probability that sampling a certain point will yield an improvement, but it may easily result in focusing only on a specific region of the objective function (exploitation) or jump around too much (exploration). EI aims to quantify the improvement and is less prone to the aforementioned issues [30]. In ChunkFunc we rely on both: we use EI to determine which point, i.e., resource profile p, to profile next and POI to define the stopping criterion. Since EI yields an absolute value and POI a percentage, the latter is more suitable as a stopping criterion.

Our aim is to achieve a relative root mean square error (RMSE) of 10% or less when comparing the BO-guided profiling results for an input data size to exhaustive profiling results to ensure that the inferred profiling results adequately reflect the actual performance. We use two stopping criteria for BO: i) the POI for sampling the next resource profile is below 2%, provided that we have sampled at least 10% of the available resource profiles, or ii) we have sampled 40% of the available resource profiles. Based on our experience it is necessary to sample at least 10% of all resource profiles, because for some functions the POI is already below 2% after the initial samples, but the relative RMSE would be above 10% . The second stopping criterion is necessary, because for some functions the POI does not drop below 2%, even though the RMSE is already sufficiently low.

Each input data size $|x|$, resulting from the user-defined discrete set of typical inputs, is profiled independently with a distinct BO model. Once a stopping criterion is fulfilled, the performance profile of the function with the input data size $|x|$ is built. For each resource profile the profiler takes either the mean SLO metric that was measured, if the resource profile has been evaluated, or uses the BO's surrogate model (GP) to infer the SLO metric. The number of inputs needed for profiling varies depending on the function. Determining this number is beyond the scope of this paper, but we will outline a solution in Section VI-C.

C. Bayesian Optimization Hyperparameter Selection

The key to make the ChunkFunc Profiler converge quickly to an accurate solution is to pick the acquisition function's hyperparameters correctly – for EI this is the ξ hyperparameter.

Algorithm 1. Bayesian Optimization Dynamic Hyperparameter Selection.

Data: f: Function to be profiled;
x: Input to be profiled;
Ξ: Set of ξ candidate values;
RP_{all}: Set of available platform resource profiles;
Result: BO_f^x: Initialized BO model for f with input x;

$RP_{init} \leftarrow$ `GetInitProfiles`(RP_{all});
$M_{init} \leftarrow$ `RunProfiling`(f, x, RP_{init});
$BO \leftarrow \{ \forall \xi \in \Xi :$ `NewBOModel` $(\xi, M_{init}) \}$;
$RP_{test} \leftarrow$ `GetTestProfiles`(RP_{all});
$M_{test} \leftarrow$ `RunProfiling`(f, x, RP_{test});
$RMSE \leftarrow \{\}$;
forall $BO_\xi \in BO$ **do**
  $M_{inf} \leftarrow \{\}$;
  forall $p \in RP_{test}$ **do**
    $M_{inf} \leftarrow M_{inf} \cup \{$ `Infer`$(BOxi, p)$ $\}$;
  end
  $RMSE \leftarrow RMSE \cup \{(BO_\xi,$`CalcRMSE`$(M_{inf}, M_{test}))\}$
end
$(BO_f^x, rmse) \leftarrow$ `FindLowestRMSE`$(RMSE)$;
`AddSamples`(BO_f^x, M_{test});
return BO_f^x;

function `RunProfiling`(f, x, RP)
  $M_{RP} \leftarrow \{\}$; // SLO metric measurements
  forall $p \in RP$ **do**
    $(m_x, c_x) \leftarrow$ `ProfileWithInput`(f, p, x);
    $M_{RP} \leftarrow M_{RP} \cup \{(p, m_x)\}$;
  end
  return M_{RP};
end function

function `ProfileWithInput`(f, p, x)
  $f^p \leftarrow$ `Spawn`(f, p) ; // Spawn f with profile p
  $f^p(x)$; // Invoke f^p to avoid cold start
  $M_x \leftarrow \{\}$; // SLO metric measurements for x
  for $i \leftarrow 0$ *to numIterations* **do**
    $f^p(x)$;
    $M_x \leftarrow M_x \cup \{$`GetSloMetric()`$\}$;
  end
  `Destroy`(f^p);
  $(m_x, c_x) \leftarrow$ `CalcMeanSloMetricAndCost`(M_x);
  return (m_x, c_x);
end function

ξ determines whether the Profiler's acquisition function will favor exploring unknown ranges of the input domain to find this maximum (higher ξ values) or focus on finding the maximum in an already known range (lower ξ values). Since we could not observe any correlation pattern between function type, input data size, and the ξ value that yields the lowest RMSE, we devised a dynamic hyperparameter selection approach, which we describe in Algorithm 1.

We start by selecting a fixed number of resource profiles RP_{init}, evenly distributed from the set of all resource profiles, and profiling the serverless function with each of them. For each resource profile a function instance is deployed, executed once to avoid cold starts, and then executed five times to obtain mean measurements for the SLO metric and the cost. These measurements are used to initialize one BO model for each of the candidate hyperparameter ξ values.

Next, we pick another set of evenly distributed profiles RP_{test}, which will be used to test the accuracy of the BO models. We profile the function with these to get SLO metric measurements M_{test} to compare the predictions against. For each BO

model we infer the SLO metric values for the profiles in RP_{test} and, then, compute the RMSE to the actual measurements M_{test}.

Finally, we pick the BO model that yields the lowest RMSE, add M_{test} to it, and continue profiling with it until one of the stopping criteria is fulfilled. This allows us to select the most suitable ξ hyperparameter without additional profiling runs in the average case. In the worst case, if a stopping criterion would be met after 10% of the resource profiles, obtaining M_{test} results results in a negligible number of additional profiling runs.

IV. ChunkFunc Workflow Optimizer

ChunkFunc Workflow Optimizer leverages the performance profiles to assign resource profiles to each individual function instance in a workflow, based on the input data sizes, while fulfilling the SLO and minimizing cost. The SLO serves as an upper or lower bound for the aggregated SLO metric of the entire workflow, while the total cost should be minimized. Since the set of function inputs X_W is filled step by step as the workflow executes, we need a heuristic to approximate the solution of the ChunkFunc optimization problem as the workflow progresses.

Before executing a function, the workflow orchestrator queries the Workflow Optimizer for the resource profile. Akin to the optimization problem, the Workflow Optimizer models the workflow as a DAG. To determine the resource profile for a function, the Workflow Optimizer needs its input data size. The heuristic receives as input the workflow graph, the SLO, the input data size for the current function, and the SLO metric value for the current execution path. Since the heuristic is invoked for each node, while the workflow is executing, it can react if previous functions affected the SLO metric differently than expected, e.g., they took more time than expected.

The *Proportional Critical Path heuristic* can use any performance metric as SLO metric. This heuristic derives a sub-SLO for the current function and chooses the cheapest resource profile that allows meeting the sub-SLO, based on the function's performance profiles. Adapting the heuristic for cost-based SLOs is possible and planned as future work.

Since metrics of functions may vary between workflow invocations, for any function, other than the first one, the remaining SLO metric until a violation of the workflow SLO may differ. For example, for an MRT SLO suppose $s_W = 100$ ms, if f_0 takes 10 ms, the remaining time for f_1 and its successors is 90 ms. If f_0 took 15 ms, the remaining time would be 85 ms. Thus, the each function's sub-SLO must be calculated dynamically before selecting a resource profile.

To compute the sub-SLO of a function f_i, we need to know how much it contributes to the overall SLO metric of the remaining workflow. The latter is the length of the critical path from (including) f_i until the end of the workflow. We define the critical path as the longest path between two nodes [31], with an edge's weight being the SLO metric of its target node. Since many metrics vary depending on input data sizes, finding the critical path is not trivial. We compute the mean SLO metric value of every function across all resource profiles and input data sizes and use these values as weights for the critical path. If the SLO metric is an additive metric, we now add the mean SLO metric value of f_i to the critical path to allow us to calculate

Algorithm 2. Proportional Critical Path Heuristic.

Data: f: Function, for which to select the resource profile;
x: Input for f;
PP_f: Set of performance profiles for f;
$W = G(F, E)$: Workflow DAG;
s_W: SLO for W;
M_{avg}: Mean SLO metrics for all functions in W;
m_{curr}: Current SLO metric value, e.g., elapsed time for MRT SLO;
Result: p: Selected resource profile for f;

```
1  s_f ← ComputeSubSLO(f, s_w, m_curr, M_avg);
2  if PP_f contain GP inferences then
3  |  s_f ← s_f * safetyMargin;
4  end
5  p ← nil ;          // The selected resource profile
6  m_p ← ∞ ;     // SLO metric value under profile p
7  c_p ← ∞ ;                  // Cost under profile p
8  PP_f^x ← GetPerfProfilesForInputSize(PP_f, |x|);
9  forall pp_i ∈ PP_f^x do
10 |  m_f ← GetSloMetric(pp_i);
11 |  c_f ← GetCost(pp_i);
12 |  if m_f ≺ s_f then
13 |  |  if c_f < c_p OR (c_f = c_p AND m_f ≺ m_p) then
14 |  |  |  p ← GetResourceProfile(pp_i);
15 |  |  |  m_p ← m_f;
16 |  |  |  c_p ← c_f;
17 |  |  end
18 |  end
19 end
20 if p = nil then
21 |  p ← GetFastestProfileForInputSize (PP_f, |x|);
22 end
23 return p;
24 function ComputeSubSLO(f, s_w, m_curr, M_avg)
25 |  cp ← FindSloCriticalPath(f, FinalNode(W));
26 |  if M is additive then
27 |  |  m_f ← GetAvgSloMetric(M_avg, f);
28 |  |  m_remaining ← s_W − m_curr;
29 |  |  contrib_f ← m_f / (GetSloMetric(cp) + m_f);
30 |  |  return m_remaining * contrib_f;
31 |  else
32 |  |  return min(s_w, GetSloMetric(cp))
33 |  end
34 end function
```

f_i's proportional contribution to it. This proportion used on the remaining workflow SLO yields the sub-SLO. For a min-metric, we take the minimum of the f_i's SLO metric value and the aggregated SLO metric value of the critical path. Algorithm 2 outlines the Proportional Critical Path heuristic:

Step 1: Line 1 computes the sub-SLO of function f using `ComputeSubSLO()`. For additive SLO metrics we use Dijkstra's shortest path algorithm to find the critical path. The weight of each edge (f_i, f_j) is the negative mean SLO metric of f_j. All weights are negative, so Dijkstra's algorithm works normally and we get the longest path. For an additive metric we compute the sub-SLO using f's proportional contribution to the critical path, while for a min-metric we use the minimum of f's and the critical path's SLO metric values. After returning the sub-SLO we multiply it with a safety margin (line 3) if the performance profiles contain inferences from BO. This ensures that imprecisions resulting from the inferences do not affect SLO adherence.

Step 2: Lines 5–8 initialize the selected resource profile p to nil and f's SLO metric and cost to infinity. Then, f's performance profiles for the current input data size are retrieved. Performance profiles are stored in buckets, according to the input

data size they were computed for. Input x matches the bucket with the smallest input data size that is greater than or equal to the size of x. For inputs that are greater than the largest bucket input data size, that greatest bucket is taken.

Step 3: Lines 9–19 iterate over f's performance profiles for the current input data size. For each performance profile pp_i we check if its SLO metric value allows meeting the sub-SLO (line 12). If that is the case, the cost of pp_i is examined (line 13). If pp_i is cheaper than the currently selected profile p or, if their costs are equal, but pp_i has a better SLO metric value, the selected profile is updated to the resource profile in pp_i.

Step 4: If no resource profile meets the sub-SLO, we fall back to the fastest profile for the input size, irrespective of its metrics, hoping that subsequent functions meet their SLOs. Finally, the selected resource profile is returned.

V. Implementation & Experiments Design

To evaluate ChunkFunc we focus on the quality of the Workflow Optimizer results, i.e., whether its resource profile selection meets the workflows' response time SLOs and how much the total cost is. We compare ChunkFunc to two state-of-the-art approaches. All code and data needed to run the experiments, as well as, additional results can be found in our repository.[4]

A. Implementation

We implement ChunkFunc Profiler in Go as an open source[5] Kubernetes controller and target serverless functions realized with Knative. Without loss of generality, the Profiler currently triggers functions via HTTP requests, since this is a common and flexible invocation method. Our trigger mechanism abstraction allows for adding other trigger types, e.g., storage events, in the future. ChunkFunc-specific function metadata is passed to the Profiler, using a Kubernetes Custom Resource Definition (CRD), i.e., a custom type of object that can be stored in the cluster. Each such `FunctionDescription` object contains a reference to the Knative function definition object and a list of typical inputs. Once the Profiler detects a new `FunctionDescription` it automatically starts profiling the referenced Knative function and, upon completion, adds the performance profiles to the `status` subresource of the `Function -Description`. To evaluate the Workflow Optimizer we design various workflows and for each we replay real-life function traces from our performance profiles in our custom simulator. Our simulation with real-life traces is deterministic, so it needs to be executed only once for each configuration, which enables faster exploration of a large range of SLOs.

B. Experiments Setup

To evaluate ChunkFunc we use three real-world and six synthetic serverless workflows. The real-world workflows are written for our research, but are similar to production use cases. They are

1) a log processing workflow (*LogPro*),
2) a video processing workflow (*VidPro*), and
3) an ML-based face detection workflow (*FaceDet*).

They represent typical examples of serverless workflows with variable input data size, while exhibiting different response time characteristics. LogPro takes a log file from a distributed cluster scheduler [22] from an S3-compatible storage bucket as input. The workflow consists of a sequence of four serverless functions that validate the log and extract various statistics. VidPro cuts out an unwanted segment of a video from S3, and encodes the rest in a predefined format for social media. The workflow consists of four functions that validate the video, cut and encode the two segments (two parallel instances of the same function), and merge the encoded segments. FaceDet detects and marks faces in a video from S3. It consists of a sequence of four functions: validation, transformation of the video to a standardized resolution, face detection, and marking of all faces in an output video.

Additionally, we use six synthetic workflows, which are assembled using profiling results from the real-world workflows. Like the real-world functions, the response time of the functions in the synthetic workflows is dependent on the input size, as determined by the profiling results. During generation of the workflows, each function's output is chosen from the set of supported input sizes of the successor function. For each workflow there are three input size configurations: small, medium, and large. The synthetic workflows are: i) *homogeneous*, a sequence of functions with the same (medium) resource requirements, ii) *LoHiRes*, a sequence of functions with low resource requirements, followed by a sequence of functions with high resource requirements, iii) *HiLoRes*, high resource functions, followed by low resource functions, iv) *random*, a random sequence of functions with low, medium, and high resource requirements, v) *cyclic*, a low resource function, followed by a medium resource, followed by a high resource function, repeated in cycles, and vi) *staircase*, a sequence of low resource functions, followed by a sequence of medium resource functions, followed by a sequence of high resource functions. The first four workflows consist of 40 functions each, while the last two consist of 42 functions.

Two sets of workflows allow us to demonstrate how ChunkFunc behaves with real-life applications and to use the longer and more complex synthetic workflows to evaluate ChunkFunc's scalability. Our workflows are mostly sequential, because ChunkFunc relies on the critical path in a workflow and even in a massively parallel workflow, the critical path is always sequential. The response times of our functions is within the range of the current state-of-the-art, e.g., AWS imposes a default function timeout of 3 seconds, which can be changed to a maximum of 15 minutes [32]. The average end-to-end durations (base SLOs) of our workflows cover a wide range, starting at 12 seconds for LogPro, approx. one minute for VidPro, 5 minutes for FaceDet, extending to 75 minutes for the synthetic HiLoRes workflow. The number of functions per workflow is representative of most serverless workflows currently in use. A large scale study [33]

[4]https://polaris-slo-cloud.github.io/chunk-func/experiments/

[5]https://github.com/polaris-slo-cloud/chunk-func and https://doi.org/10.5281/zenodo.14174081

showed that 59% of workflows consisted of 2–10 functions, 19% of 10–1000 functions, and 3% more than that (19% could not be categorized).

We implement all real-world functions in TypeScript, except for face detection and marking, for which we use Python. We deploy them using Knative v1.10 on a Kubernetes v1.27 cluster. For video processing we wrap ffmpeg[6] v6.0 and use the x264[7] and AAC codecs. Face detection and marking relies on the OpenCV[8] library.

We run the experiments with two sets of resource profiles. The first set of profiles and their costs per 100 ms is coarse-grained and resembles the 128 MB – 16384 MB profiles available on Google Cloud Functions (GCF) [25] (Tier 2 prices). Since for GCF there are eight profiles in this memory range, we use exhaustive profiling for this set of resource profiles. The second set of resources profiles and their costs per 1 ms is fine-grained and resembles the 128 MB – 10240 MB range available on AWS Lambda [34]. Every memory size maps to the CPU core count defined by AWS [26]. AWS uses a continuous memory range, which we divide into 64 MB steps, which results in 159 resource profiles, for which we use BO-guided profiling.

We implement six heuristics:

1) Fastest configuration,
2) Cheapest configuration,
3) ChunkFunc Proportional Critical Path heuristic (ChunkFunc),
4) ChunkFunc with known function output sizes (CF-Oracle),
5) SLAM [11], and
6) StepConf [10].

CF-Oracle is identical to ChunkFunc, except that the former knows all function output sizes from an "oracle" when computing the critical path – this is only used in comparison to ChunkFunc to assess the effectiveness of function output size estimates compared to the actual output sizes when determining the critical path. Both heuristics compute an average across all resource profiles for the critical path. SLAM and StepConf both rely on offline profiling to build a performance model of the functions. We execute all experiments using the exhaustive profiling results and using the BO predicted profiling results.

SLAM precomputes all function configurations prior to executing the workflow. It inserts all functions using their response times under the lowest resource configuration into a max-heap. SLAM pops the slowest function off the heap, increases its resources to the next higher profile, and reinserts it into the heap. If the resources cannot be increased further, the function's configuration is frozen, and it is not reinserted into the heap. SLAM continues until an SLO-compliant configuration is found or the heap is empty. A second version of the algorithm checks if the percentage of decrease in response time is greater than the percentage of cost increase before returning a function to the heap. We use the cheaper of the two results.

[6]https://www.ffmpeg.org
[7]https://www.videolan.org/developers/x264.html
[8]https://opencv.org

TABLE II
REAL-WORLD WORKFLOW SCENARIOS

Workflow	Input Sizes	SLO Interval (sec) for Profiles	
		Coarse-grained	Fine-grained
LogPro	2.4 MiB 54.3 MiB 95.8 MiB	12.1s +/- 15% [10.3; 14.0]	-
VidPro	360p - 40 MiB 720p - 227 MiB 1080p - 500 MiB	77.4s +/- 35% [50.3; 104.5]	51.0s +/- 35% [33.2; 68.9]
FaceDet	20s, 720p - 6.51 MiB 40s, 720p - 26.5 MiB 60s, 1080p - 73.5 MiB	330s +/- 35% [214.5; 445.4]	302.2s +/- 35% [196.4; 408.0]

TABLE III
REAL-WORLD WORKFLOWS SLO COMPLIANCE FOR COARSE-GRAINED RESOURCE PROFILES

Workflow	Input Size	SLO Adherence				
		ChunkFunc	SLAM		StepConf	
LogPro	Small	100%	100%	67%	100%	91%
	Medium		100%		100%	
	Large		0%		74%	
VidPro	Small	100%	100%	69%	100%	78%
	Medium		100%		100%	
	Large		6%		34%	
FaceDet	Small	100%	100%	68%	100%	75%
	Medium		87%		73%	
	Large		17%		51%	
Overall		100%	68%		81%	

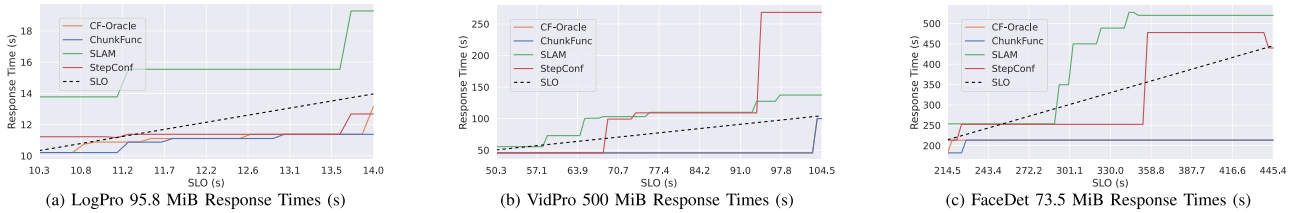
StepConf chooses each function's resource profile directly prior to its execution using an NP-hard algorithm or a heuristic on a DAG and is a representative for state-of-the-art graph-based algorithms. The heuristic we implemented for our experiments, computes a sub-SLO for each function step, based on its contribution to the critical path until the end of the workflow and the remaining time until SLO violation. For computing the critical path, the response time of the most cost-effective resource profile is used for every function.

Since SLAM and StepConf are unaware of different input data sizes, we use the profiling results for each function's median input data size for these strategies.

VI. EXPERIMENTAL RESULTS

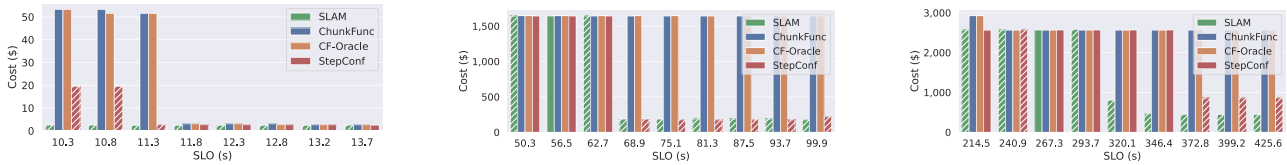
A. Real-World Workflows

For each real-world workflow we create and profile scenarios with a small, a medium, and a large input size. To define MRT SLOs we use the fastest and the cheapest configurations as the lower and upper bounds. For example, for the largest input data size for VidPro the lower and upper bounds for the response time on the coarse-grained profiles are 44.788 seconds and 110.089 seconds. We define the $baseSlo = lowerBound + \frac{upperBound - lowerBound}{2}$, e.g., 77.4 s for VidPro. We explore the SLO interval of $baseSlo \pm N\%$ in one-percent steps, i.e., $N + 1$ distinct SLOs. We chose N s.t. the interval does not exceed the bounds given by the fastest and cheapest configurations. Since the available resources in the lowest and the highest profiles differ between the coarse-grained and the fine-grained resource profile sets, also the lower and upper response time bounds and, hence, the base SLOs differ. All workflow configuration scenarios are shown in Table II .



(a) LogPro 95.8 MiB Response Times (s) (b) VidPro 500 MiB Response Times (s) (c) FaceDet 73.5 MiB Response Times (s)

Fig. 3. LogPro, VidPro, and FaceDet Maximum Response Time SLO compliance for large inputs for coarse-grained resource profiles.



(a) LogPro 95.8 MiB Costs per 10,000 Invocations (b) VidPro 500 MiB Costs per 10,000 Invocations (c) FaceDet 73.5 MiB Costs per 10,000 Invocations

Fig. 4. LogPro, VidPro, and FaceDet Costs per 10,000 invocations for large inputs for coarse-grained resource profiles.

1) Coarse-Grained Resource Profiles - Exhaustive Profiling: Fig. 3 shows the SLO compliance results as response time graphs for the large input data sizes (the other sizes are available in our repository). The dashed black line denotes the MRT SLO, i.e., to fulfill the SLO, the workflow's response time must be equal to or below this line. Table III shows details for all input data sizes.

All heuristics exhibit long periods of straight lines in the response time graphs, because they use a certain set resource configurations until the SLO relaxes enough to use a less powerful resource profile on one function – this behavior causes a straight line in the graph. Additionally, the relatively short workflows allow only few functions to be adapted, thus increasing the length of the straight lines; the synthetic workflows exhibit many more "steps" in the graphs.

ChunkFunc (standard and CF-Oracle version) is the only heuristic that meets the SLO in all cases across all input sizes. SLAM and StepConf work well for one or two input sizes, but fail a substantial amount of SLOs in the rest. SLAM fulfills two thirds of the LogPro SLOs, 69% of VidPro, and 68% of FaceDet. SLO violations occur for medium and large inputs for FaceDet and only for large inputs for the other two. Compared to SLAM, ChunkFunc increases SLO adherence by 45% to 49% . StepConf fulfills 91% of the LogPro SLOs, 78% for VidPro, and 75% for FaceDet. Most violations occur for large input sizes, but for FaceDet StepConf also misses 27% of the SLOs for medium inputs. Compared to StepConf, ChunkFunc increases SLO adherence by 10% to 33% . Across all workflows, SLAM fulfills 68% of the SLOs, while StepConf meets 81%, this amounts to a mean increase in SLO adherence of 47% and 23% respectively, when using ChunkFunc instead.

Fig. 4 shows the costs for 10,000 workflow invocations. If an algorithm violates an SLO the respective cost bar is shown with a hatch pattern, because if the SLO is not met, evaluating the cost is pointless. To ensure comparability we show the costs for each algorithm only where it meets the SLO. To avoid bias from SLO violating configurations, when analyzing the costs, we conduct a one-on-one comparison, where we consider only the cases where

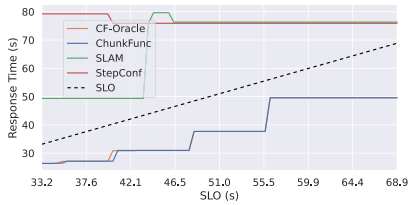


Fig. 5. VidPro 500 MiB MRT SLO compliance for fine-grained profiles.

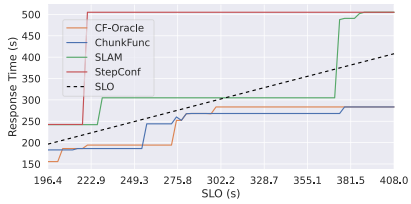


Fig. 6. FaceDet 73.5 MiB MRT SLO compliance for fine-grained profiles.

both strategies meet an SLO. We compare the mean costs of these cases. When comparing ChunkFunc to SLAM, ChunkFunc is 4% cheaper for LogPro, 54% cheaper for VidPro, and 19% cheaper for FaceDet. When comparing to StepConf, ChunkFunc is 165% more expensive for LogPro, 29% cheaper for VidPro, and 22% cheaper for FaceDet. For LogPro ChunkFunc is more expensive than StepConf for almost all SLOs (for some they are even). This is because ChunkFunc often picks faster resource profiles, because it knows that it needs to fulfill every sub-SLO for the large input, while StepConf assumes the medium input, for which the sub-SLO can be fulfilled with cheaper resource profiles. While this approach allows StepConf to save costs, it also causes it to miss the tight SLOs for large inputs. In the general case, ChunkFunc fulfills more SLOs than StepConf. In workflows with long-running functions, such as VidPro and FaceDet, ChunkFunc allows saving up to 48% of the costs over StepConf.

2) Fine-Grained Resource Profiles - BO-Guided Profiling: Figs. 5 and 6 show the SLO compliance results as response time

TABLE IV
REAL-WORLD WORKFLOWS SLO COMPLIANCE FOR BO-INFERRED FINE-GRAINED PROFILES

Workflow	Input Size	SLO Adherence				
		ChunkFunc	SLAM		StepConf	
VidPro	Small	100%	100%	67%	100%	62%
	Medium		100%		86%	
	Large		0%		0%	
FaceDet	Small	100%	100%	74%	100%	36%
	Medium		89%		7%	
	Large		32%		0%	
Overall		100%	71%		49%	

graphs for the large input data sizes. Table IV shows details for all input data sizes. The straight lines in the graphs are caused by the same reasons as for the coarse-grained resource profiles. ChunkFunc is the only heuristic that meets the SLO in all cases across all input sizes. Its SLO adherence is completely unaffected by whether we use the exhaustive profiling results or the BO-inferred profiling results. SLAM and StepConf work well for one or two input sizes, but fail a substantial amount of SLOs in the rest. SLAM fulfills two thirds of all VidPro SLOs and 74% of FaceDet, with SLO violations occurring mostly for large inputs. Compared to SLAM, ChunkFunc increases SLO adherence by 35% to 50% . StepConf fulfills 62% of the VidPro SLOs and 36% for FaceDet. Only for small input sizes all the SLOs are met, while as for medium input sizes there are already considerable violations for VidPro and almost entirely violated for FaceDet. Compared to StepConf, ChunkFunc increases SLO adherence by 61% to 178% . Across all workflows, SLAM fulfills 71% of the SLOs, while StepConf meets 49%, this amounts to a mean increase in SLO adherence of 41% and 104% respectively, when using ChunkFunc instead. We have excluded LogPro from the experiment with fine-grained resources profiles. This is because its functions are single-threaded (Node.JS) with low memory requirements. Since the fine-grained resource profiles all contain at least one vCPU, there is almost no performance difference between the resource profiles, hence the omission of LogPro from this experiment.

We do not show the cost graphs here, because SLAM and StepConf fail to meet almost all of the SLOs for large inputs. When comparing ChunkFunc to SLAM one-on-one across all input sizes, where both heuristics meet the SLO, ChunkFunc is 48% cheaper for VidPro and 6% more expensive for FaceDet. When comparing to StepConf, ChunkFunc is 36% more expensive for VidPro and 42% more expensive for FaceDet. However, ChunkFunc fulfills many more SLOs than SLAM and StepConf. This justifies a slight increase in cost for one workflow with respect to SLAM. With respect to StepConf, the cost increases are more substantial. However, these increases cover less than two thirds of the SLOs for VidPro and only slightly over one third of the SLOs for FaceDet; for the remainder StepConf fails to meet the SLO.

B. Synthetic Workflows

1) Coarse-Grained Resource Profiles - Exhaustive Profiling: The synthetic workflows are used to evaluate ChunkFunc's scalability in longer, more complex workflows. The homogeneous, LoHiRes, HiLoRes, and random workflows consist of 40 functions in sequence. The cyclic and staircase workflows use a short-running, medium-running, and a long-running function, each of which appears 14 times in the workflow, hence they consist of a total of 42 functions. For all synthetic workflows we simulate scenarios with a small, a medium, and a large input.

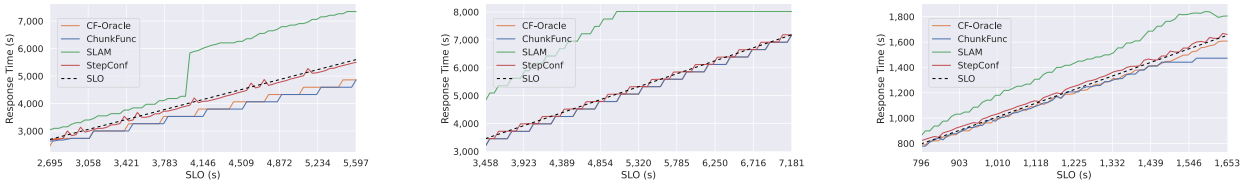
Fig. 7 shows the SLO adherence for the coarse-grained profiles for the large inputs to the cyclic, HiLoRes, and homogeneous workflows, which we use as a representative examples (for other graphs please see our repository). The SLO adherence of the heuristics shows three pattern categories: For the cyclic, random, and staircase workflows the heuristics exhibit the pattern exemplified in Fig. 7(a). The LoHiRes and HiLoRes workflows show the pattern in Fig. 7(b). The SLO adherence for homogeneous workflow has its own distinct pattern shown in Fig. 7(c).

ChunkFunc's pattern shows only minor differences between the workflows. It is the only heuristic that meets all SLOs for all input sizes. StepConf's pattern remains consistent across all workflows. For large inputs, it varies closely between fulfilling and violating the SLOs. Across all six synthetic workflows and three input sizes, it meets 79% of the SLOs, with the lowest value being 60% for the homogeneous workflow and the highest being 96% for the cyclic workflow. SLAM exhibits the largest differences in its patterns. It violates all large input SLOs, but fulfills all SLOs for the other inputs, yielding an average adherence of 67% . For the cyclic, random, and staircase workflows, SLAM's response times are first close to the SLO line and diverge at some point from it. For HiLoRes and LoHiRes the response times are always far from the SLO until they plateau out at some point. For the homogeneous workflow, SLAM's response times are closer to the MRT SLO line. For ChunkFunc the results yield an increase in SLO adherence of 27% over StepConf and 50% over SLAM.

For costs we perform the same one-on-one comparison for fulfilled SLOs that we did for the real-world workflows. For the cyclic workflow ChunkFunc is 48% cheaper than SLAM and 27% cheaper than StepConf. For the staircase workflow ChunkFunc only requires 39% of the costs of SLAM, making it 61% cheaper. On average ChunkFunc is 38% cheaper than SLAM and 10% cheaper than StepConf.

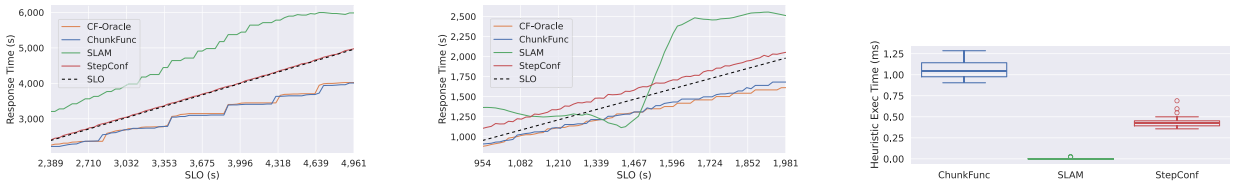
2) Fine-Grained Resource Profiles - BO-Guided Profiling: Fig. 8 shows the SLO adherence for the fine-grained resource profiles with large inputs to the cyclic and homogeneous workflows, which we use as a representative examples (for other graphs please see our repository). We omit the costs for the cyclic workflow for these resource profiles, because only ChunkFunc manages to fulfill all SLOs for large inputs. The SLO adherence of all but one workflow follows the pattern shown in Fig. 8(a), where ChunkFunc meets all SLOs, StepConf meets some, but closely misses most SLOs, and SLAM misses all SLOs. The exception is the homogeneous workflow, shown in Fig. 8(b), where ChunkFunc fulfills all SLOs, StepConf misses all SLOs, and SLAM fulfills a little less than a quarter of the SLOs.

ChunkFunc is the only heuristic that meets all SLOs for all input sizes for the BO-predicted profiles. Across all six synthetic



(a) Cyclic WF Large Input Response Times (s) (b) HiLoRes WF Large Input Response Times (s) (c) Homog. WF Large Input Response Times (s)

Fig. 7. Representative results of synthetic workflow experiments for coarse-grained resource profiles.



(a) Cyclic WF Large Input Response Times (s) (b) Homog. WF Large Input Response Times (s) (c) Cyclic WF Large Input Heuristic Execution Times.

Fig. 8. Representative results of synthetic workflow experiments for fine-grained resource profiles.

workflows and three input sizes, StepConf meets 53% of the SLOs, with the lowest value being 45% for the homogeneous workflow and the highest being 59% for the random workflow. SLAM meets 65% of all SLOs, with 62% and 74% being the lowest and highest values respectively. For ChunkFunc this yields an increase in SLO adherence of 89% over StepConf and 54% over SLAM.

For the costs of the cyclic workflow, ChunkFunc amounts to only 49% of the costs of SLAM and 98% of the costs of StepConf. For the homogeneous workflow ChunkFunc requires 77% more costs than StepConf, but in any other case, ChunkFunc is cheaper. On average ChunkFunc reduces costs by 52% compared to SLAM. Compared to StepConf ChunkFunc is 5% more expensive overall, because of the homogeneous workflow. For the other five workflows, ChunkFunc is on average 9% cheaper than StepConf.

The cost difference between ChunkFunc and CF-Oracle, which knows all function outputs when computing a critical path, is negligible. Across all experiments with all workflows and resource profiles ChunkFunc is only 1% more expensive on average, which shows that its critical path estimation works well for keeping costs low, while fulfilling the SLOs.

Fig. 8(c) examines the execution times of the three heuristics for the cyclic workflow. We log the execution time for computing each resource profile in a simulation and, then, compute the mean time for determining a single resource profile. We accumulate these values across all SLOs for an input size. Since SLAM only performs max-heap operations it is the fastest. ChunkFunc and StepConf both compute paths through a DAG and show a similar performance, with median values close to 1 ms and 0.5 ms respectively. Since ChunkFunc fulfills all SLOs, the slight increase in computation time over StepConf is justifiable and since it is marginal, it does not affect the user experience when invoking a workflow.

C. Takeaways

While automatic profiling causes some up front costs, workflows are typically executed for months or years in production. For example, profiling the `merge-videos` function in the VidPro workflow took 106 minutes, which on a GCP `c2-standard-30` VM with SSD amounts to a one time cost of about $2.27, which amortizes quickly since ChunkFunc may reduce function execution costs by up to 61% . New versions of a function can reuse existing performance profiles. Reprofiling is only necessary if the changes affect the function's performance. This can be revealed using a performance test in the continuous integration pipeline.

The number of inputs that should be profiled for a particular function to obtain the best resource optimization results depends on the function and its typical uses. A suitable approach for a production system is to monitor a function's live usage for a representative period, e.g., one week. A clustering of inputs can be used to identify the ideal number of inputs for profiling and to obtain sample input data as well. An automation of this step is currently out of scope, but should be considered as a future expansion.

State-of-the-art approaches for resource optimization do not consider input sizes, causing them to underestimate function response times, especially for large inputs. This leads to the selection of too weak resource profiles, often violating the SLO. ChunkFunc is the only heuristic that always meets the SLO because its input size-aware heuristic provides more accurate estimates for function response times. The analysis of all results shows that the more accurate critical path estimation and input data size awareness of ChunkFunc fulfills the SLOs in all test cases, an increase of a factor of 1.04 to 2.78, with respect to the state-of-the-art and a maximum cost saving of 61% . The advantage of input data size awareness becomes more apparent as the input data size-dependent response time of the functions increase, i.e., ChunkFunc performs better in processing intensive workflows, such as video encoding, where a badly chosen resource profile has a large effect.

In some cases the input data size is not the most decisive factor for function response time because other properties of the input are more important. For example, a video's file size is determined by its length and bitrate. However, when encoding a video, as we do in the VidPro workflow, the video's resolution has a

much greater effect on the encoding duration than its bitrate. Another example is earth observation data from satellites: the image resolution and raw data size are always the same, but the processing complexity can change depending on whether the image shows the ocean or an urban area. To encompass such cases, ChunkFunc's input size parameter can be generalized to an abstraction that represents an arbitrary numeric property of the input, which affects processing time the most. In many cases this is the file size, but in some cases it may be another property. For example, for VidPro and FaceDet we use the product of $resolution \times length$ as the "input size." For satellite imagery a preprocessing function can be used to determine the complexity, which will be used as the "input size" for the next function.

Our Workflow Optimizer uses bucketing for selecting a performance profile for an input that does not exactly match one of the pre-computed performance profiles. Doing this instead of linear interpolation between profiles makes it easier to fulfill the SLOs. In the future a Gaussian Process could be bootstrapped with the pre-computed profiles and, then, used to infer the resource profile for such unknown inputs.

VII. Related Work

A. Resource Configuration Optimization

Solutions most similar ChunkFunc, which aim to optimize the resource configurations of functions can be divided into two categories: i) approaches that build a performance model offline using a-priori profiling and ii) approaches that build the performance model online using monitoring data. Each category can be further subdivided depending on whether it supports single functions or entire workflows and by the algorithm type used to determine function configuration(s).

Offline Performance Modeling using A-priori Profiling: A-priori profiling typically executes the serverless function or workflow using a representative input or set of inputs under different resource configurations to build a performance model in an offline fashion, which is used to tune the function configuration(s) for production execution.

Approaches for single functions use a wide variety of algorithms. AWS Lambda Power Tuning [35] executes profiling runs and graphs the response times and costs to let users manually pick a configuration. CPU-TAMS [14] relies on regression modeling to create a "vCPU-to-memory model" for a particular platform. Subsequently, a single profiling run for a function using the maximum resources configuration suffices to perform optimization. CherryPick [13], albeit originally developed for Big Data analytics jobs, uses BO to reduce the number of profiling runs needed to find a configuration that matches, e.g., a response time SLO. MAFF [12] uses linear, binary, or gradient descent search to find a suitable configuration. It supports an active mode (a-priori profiling) and a passive mode (using monitoring data only).

Optimizing a workflow to meet an SLO is much harder, because the performance of one function can affect the available resource choices for subsequent functions. Most approaches for serverless workflows use graph algorithms on the workflow's call graph, or on a graph derived form it, to find suitable configurations. StepConf [10] estimates function execution times using a piece-wise fitting model, based on results from an "offline", i.e., profiling, phase and a quantile regression model for data transmission delays between functions. These estimates are used in combination with a workflow graph in an NP-hard algorithm and in a heuristic to find function configurations that fulfill the SLO, while minimizing cost. Lin and Khazaei [9] augment a workflow graph with information, such as profiling results and probabilities of taking a certain edge after executing a function node. After transformations, such as removing cycles, they obtain a "probabilistic DAG", on which they run a Probability Refined Critical Path (PRCP) Algorithm that progressively refines the transition probabilities, while determining function configurations. Costless [8] assesses, in addition to resource configurations, the possibility to fuse multiple functions into a single function and whether to execute them in the Cloud or on the Edge. It utilizes a "cost graph", which contains paths through all possible function fusion options, with each edge weight containing the execution time and cost of running the succeeding function node.

Some a-priori profiling approaches do not use graph algorithms, such as SLAM [11], which places all functions with their lowest resource configurations in a max-heap ordered by response time. It pops off the top function from the heap, increases its resources, and checks if the workflow's SLO is fulfilled now. If not, it reinserts the function into the heap (if further resource increases are possible) and continues.

Contrary to ChunkFunc, these approaches use either a typical input data size for profiling or an aggregation of profiling results over multiple input data sizes, but they do not differentiate between different input data sizes. While CherryPick can detect a large gap between expected performance and actual performance, e.g., due to changed input data sizes, and trigger a reprofiling, the current performance profile does not support multiple input data sizes. StepConf's approach is similar to ChunkFunc, however, it relies on piece-wise fitting to determine function performance, while we use BO and its GP. StepConf uses the number of requests to a function to determine inter-function and intra-function parallelism. Through intra-function parallelism the number of requests indirectly influences the resources available to a function instance, however, the input size or complexity of a request does not. In accordance with pure serverless principles, ChunkFunc assumes each function instance processes one request at at time. Thus, the number of requests are only relevant to the autoscaler of the serverless platform and do not influence the resource configuration. Instead for ChunkFunc, the input size or complexity of each request influences the resource configuration, which leads to superior results, as shown in our evaluation. The approach of fusing functions in addition to optimizing their resources, as done by Costless, can serve as a complimentary strategy for finding SLO-compliant resource configurations. However, it cannot replace the awareness of input data sizes. The repercussions of not being aware of different input data sizes are exemplified by our evaluation of SLAM, which fails to meet the SLOs for inputs that do not match the expected size. Additionally, SLAM and PRCP precompute all configurations before executing the

workflow, SLAM using a max-heap and PRCP on a graph. This entails that they cannot adjust if some functions take longer than expected. ChunkFunc executes its heuristic directly before invoking each function, which allows it to leverage information about the current status of the workflow and react if a previous function was slower or faster than expected.

Some systems tackle the resource configuration problem specifically for ML workflows and rely on the request frequency to influence the optimization. AsyFunc [36] reduces memory usage of Deep Learning (DL) inference workflows by not loading the entire model into every function and tuning intra-function parallelism. It uses the number of requests per second to determine the number of CPU cores to be assigned to a function to achieve efficient memory usage. HarmonyBatch [37] reduces response time and costs for model inference operations. It batches infrequent requests of different applications with the same model together on the same function instance. The request frequency and application SLOs are used to determine the resources and the batching. λDNN [38] optimizes the resources and number of serverless functions used for training a Deep Neural Networks (DNN) model, based on the model parameters and a time SLO. It iterates over all possible memory profiles, similar to ChunkFunc. However, even though λDNN tunes the entire training workflow, all functions are the same and use the same resource configuration in the end, which simplifies the problem. λGrapher [39] computes the memory configuration of serverless functions for Graph Neural Network (GNN) serving as the sum of the memory required by the runtime, the graphs, and the embeddings; CPU configuration is determined using Bayesian Optimization to minimize costs. These systems, which work well for ML workflows, can leverage a-priori knowledge about the functions and/or assumptions that ChunkFunc cannot use, cause it is designed for generic serverless workflows. AsyFunc can decide to not not load the entire model into every function, which is not possible for a generic system like ChunkFunc. HarmonyBatch can batch requests that use the same model on the same function instance. While ChunkFunc could do this too if the same input data is used, it would depend on the function type if this would provide a benefit, e.g., a video encoding function will always encode the video even if it is the same function instance. Contrary to ChunkFunc λGrapher can leverage prior knowledge about the memory requirements of the GNNs. While ChunkFunc uses BO to reduce profiling time, λGrapher uses it to find the CPU configuration with the minimal costs. This is possible, because λGrapher can leverage more a-priori knowledge than ChunkFunc has available.

Online Performance Modeling: Approaches that do not use a-priori profiling typically use historical or live monitoring data to build performance models in an online fashion.

Solutions for single functions use various algorithms. AWS Compute Optimizer [16] analyzes function invocations, their duration, errors, and the number of throttled invocations and uses ML (exact technique is unspecified) to make recommendations for configurations, but does not optimize automatically. Sizeless [15] uses a multi-target regression model trained on a large dataset obtained from monitoring synthetically generated functions. This allows it to predict the execution time of a function with monitoring from a single memory configuration only. Aquatope [24] relies on BO to learn the most suitable configuration that fulfills an SLO more quickly and aims to reduce cold starts as well. FaasDeliver [18] applies a new resource configuration to a function after every execution until its model is complete. It uses a Tree-structured Parzen Estimator to reduce the number of configurations that need to be explored. Libra [40] harvests unused resources from function instances and assigns them to other instances that require more resources. It uses the first input to profile the function and to bootstrap multiple ML models; subsequent monitoring data updates these models. For every invocation, Libra predicts the required resources and the execution time based on the input size and harvests or adds resources based on these predictions.

Systems for optimizing workflows also use very diverse approaches. Eismann et al. [41] use Mixture Density Networks and Monte-Carlo simulations to predict costs of serverless workflows, based on their input sizes, but they assume that the functions' resources are already assigned and do not propose a solution to optimize them. COSE [17] relies on Bayesian Optimization to pick the resource configuration to apply to the next function execution, while it is building its performance model. Once it has sufficient data, it computes configurations and placements (Cloud or Edge) by solving an Integer Linear Programming (ILP) problem. Orion [42] optimizes resource profiles, function co-location, and cold starts. It models function response times as distributions (one for each observed resource profile) to account for variability and finds correlations between the latencies of functions in a workflow. FireFace [23] initially does not rely on monitoring data, but uses static code analysis to extract internal features to allow it to estimate execution time under various resource configurations using a prediction model. Adaptive Particle Swarm Optimization using Genetic Algorithm Operators is, then, used to find the function configurations that harmonize SLO satisfaction and cost minimization. The prediction model is regularly updated using monitoring data. Jolteon [43] uses monitoring data to build its models and formulates a chance constrained optimization problem, which is solved by a convex optimizer after converting it using Monte Carlo sampling. Astra [21], relies on graph algorithms to approximate the solution to an optimization problem for analytics workflows. Like FireFace, Astra also does not use monitoring data, but it determines function execution times with a formula that uses the input data size and the computation time on a "unit size object" for a given resource configuration.

Online performance modeling does not require profiling or configuration of typical inputs, because it monitors the running system. However, while the performance model is incomplete, the SLO will likely be violated. Statistical methods, such as those employed by COSE, FaasDeliver, and Sizeless reduce this time, but they cannot eliminate it. Except for Astra and Libra, none of these solutions account for different input data sizes. Libra reassigns unused resources, but it does not directly support SLOs and it is limited to tuning a single function. While ChunkFunc uses profiling results from different inputs, Astra needs to determine the computation time on a "unit size object". This may be hard to do complex functions and its approach is

limited to analytics workflows. The Mixture Density Networks and Monte-Carlo employed by Eismann et al. [41] could be an alternative to BO for creating performance profiles in the ChunkFunc Profiler. However, their approach also requires a collection of function monitoring data to train its model. The required volume of monitoring data is not specified, but the authors state that micro-benchmarks can be used to generate the data. This suggests that the volume is likely more than what BO needs during ChunkFunc profiling. Orion models response times as distributions to account for variability, but, contrary to ChunkFunc, Orion ignores that some variability may come from different input data sizes. Similar to SLAM, the approaches employed by COSE, FireFace, Orion, and Astra precompute the set of resource configurations prior to executing the workflow. Thus, unlike ChunkFunc, they cannot react to unexpectedly slower or faster function executions.

B. Vertical Scaling Approaches

Vertical scaling can be seen as the counterpart to resource configuration optimization for serverless functions, which is typically used in (micro)service-based applications. Many solutions use machine learning on historical and/or live monitoring data to predict scaling targets for combined vertical and horizontal scaling [44], [45], [46]. Other techniques, such as control theory [47] are also used. Approaches that focus solely on vertical scaling also rely on a variety of techniques, such as reinforcement learning [48], [49], rule-based [50], fuzzy logic [51], or regression [52], [53]. Vertical microservice autoscalers try to predict a configuration to fulfill a demand consisting of many user requests, whereas serverless configuration tuning, such as ChunkFunc, is applied on a per-request basis. Thus, traditional vertical scaling offers less flexibility since it can update resources only at coarser grain. Since Libra [40] allows harvesting unused resources from serverless function instances and assigning them to instances in need, it can be seen as a vertical scaler for serverless, albeit without direct support for SLOs.

C. Scheduling & Miscellaneous

Proper placement/scheduling of serverless functions can also play an important role in meeting SLOs. Many systems rely on monolithic schedulers. Knative uses the default Kubernetes scheduler [54], which uses a greedy multi-criteria decision making approach to find suitable nodes for the pods, but it is not SLO-aware. FnSched [55] relies on a greedy algorithm to place function instances on as few nodes as possible to allow unused nodes to be turned off. Skippy [56] is a scheduler for data-intensive serverless applications at the Edge. FaaSRank [57] uses reinforcement learning to automatically learn scheduling policies to to optimize function completion time. Owl [20] allows overcommitting physical resources with multiple serverless functions to improve resource utilization, while carefully monitoring and preventing service degradation. Monolithic schedulers have limited capacity, which means that the high scheduling frequency in serverless systems necessitates at some point a distributed scheduler to keep up with the load. Hydra [58] uses a federation of 2-level schedulers to achieve up to 40 K scheduling decisions per second. Hermod [59] supports a distributed mode and uses early binding and hybrid load balancing to reduce slowdown compared to vanilla OpenWhisk scheduling. AuctionWhisk [60] adopts a distributed scheduling approach based on an auctioning mechanism. YuanRong [61] is a complete serverless platform that is used in production. Its uses a highly-scalable multi-level hierarchical scheduler that reduces cross-node communication. Scheduling is orthogonal to the SLO-aware resource configuration provided by ChunkFunc. ChunkFunc and other resource optimization frameworks rely on schedulers to place new function instances on the most suitable nodes to deliver the required performance.

Cold starts are known to affect the response times of serverless functions [62]; mitigation of cold starts is another complimentary approach to resource tuning to ensure SLO adherence of serverless functions. Caching or keep-alive guided by probability distributions is a common strategy for cold start avoidance and employed, e.g., by FaasCache [63] and O-RDC [64]. Pre-warming, as done by StepConf [10], IceBreaker [65], or Orion [42], is a complimentary strategy that often uses workflow context information to predict which functions will be called next. Another approach is to reduce the function startup time with alternative runtimes. Catalyzer [66] restores checkpoints of previously running functions instead of starting completely new instances. The Firecracker [67] microVM relies on a lightweight Virtual Machine Monitor and a stripped down Linux kernel that boots in 125 ms. WebAssembly runtimes allow multiple functions to be hosted in the same container and, hence, allow for faster startup than a container or VM [68].

Some works are dedicated to a detailed study of serverless functions and platforms, such as [69], which deeply analyzes compute and memory performance, scheduling, and the overhead of containers. Jindal et al. [70] use profiling, statistical methods, and Deep Neural Network methods to estimate how many concurrent invocations a function can support without violating an SLO – such information can be intergrated into the profilers of resource configuration optimizers. Liu and Niu [71] examine the current pricing practices of serverless providers and formalize them into a model. As an alternative to the current static pricing, they propose a dynamic auction-based pricing model. If providers decide to adapt their pricing models, this orthogonal research can be used to update the current pricing models used by ChunkFunc and similar solutions.

VIII. Conclusion

We presented ChunkFunc, a framework for input data size-aware resource configuration in serverless workflows. We formulated an optimization problem to find function configurations that meet performance-based SLOs, while minimizing cost. The ChunkFunc Profiler executes functions to create input-size dependent performance models for them, guided by BO and partially inferred by a GP to reduce the number of profiling runs. We also showed ChunkFunc Workflow Optimizer, which adapts the configuration of functions in a workflow to meet performance-based SLOs. We evaluated ChunkFunc against

SLAM and StepConf and showed that it increases SLO adherence by a factor of 1.04 to 2.78, while reducing costs in many cases. This shows that input data size-aware resource configuration provides a significant advantage in serverless workflows with highly fluctuating input sizes.

In the future, we intend to

1) adapt ChunkFunc for cost-based SLOs,
2) investigate the use of a GP to infer more precise resource profiles for input sizes that are not part of the performance profile,
3) design a serverless-native framework for the development of serverless applications, which supports the definition and enforcement of SLOs, cold-start optimizations, and optimizations for inter-function communication, and
4) extend our cluster scheduler [22] with input-size awareness for serverless functions.

REFERENCES

[1] Amazon Web Services, Inc., "Aws lambda," 2023. [Online]. Available: https://aws.amazon.com/lambda/

[2] Microsoft, "Azure functions," 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[3] LLC Google, "Cloud functions," 2023. [Online]. Available: https://cloud.google.com/functions

[4] IBM Corp., "IBM cloud functions," 2023. [Online]. Available: https://cloud.ibm.com/functions/

[5] Datadog, "The state of serverless," 2023. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[6] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.

[7] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Trans. Serv. Comput.*, vol. 16, no. 2, pp. 1522–1539, Mar./Apr. 2023.

[8] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. 2018 IEEE/ACM Symp. Edge Comput.*, 2018, pp. 300–312.

[9] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, Mar. 2021.

[10] Z. Wen, Q. Chen, Y. Niu, Z. Song, Q. Deng, and F. Liu, "Joint optimization of parallelism and resource configuration for serverless function steps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 560–576, Apr. 2024.

[11] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "SLAM: SLO-aware memory optimization for serverless applications," in *Proc. IEEE 15th Int. Conf. Cloud Comput.*, 2022, pp. 30–39.

[12] T. Zubko, A. Jindal, M. Chadha, and M. Gerndt, "MAFF: Self-Adaptive Memory Optimization for Serverless Functions," in *Proc. Service-Oriented Cloud Comput.*, 2022, pp. 137–154.

[13] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for Big Data analytics," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 469–482.

[14] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with CPU time accounting," in *Proc. 2022 IEEE Int. Conf. Cloud Eng.*, 2022, pp. 104–115.

[15] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 248–259.

[16] Amazon Web Services, Inc., "Aws compute optimizer," 2023. [Online]. Available: https://aws.amazon.com/compute-optimizer/

[17] A. Raza, N. Akhtar, V. Isahagian, I. Matta, and L. Huang, "Configuration and placement of serverless applications using statistical learning," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 2, pp. 1065–1077, Jun. 2023.

[18] G. Yu, P. Chen, Z. Zheng, J. Zhang, X. Li, and Z. He, "FaaSDeliver: Cost-efficient and QoS-aware function delivery in computing continuum," *IEEE Trans. Serv. Comput.*, vol. 16, no. 5, pp. 3332–3347, Sep./Oct. 2023.

[19] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.

[20] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "OWL: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 78–93.

[21] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astra: Autonomous serverless analytics with cost-efficiency and QoS-awareness," in *Proc. 2021 IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 756–765.

[22] T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, "Vela: A 3-phase distributed scheduler for the edge-cloud continuum," in *Proc. 2023 IEEE Int. Conf. Cloud Eng.*, 2023, pp. 161–172.

[23] M. Li, J. Zhang, J. Lin, Z. Chen, and X. Zheng, "FireFace: Leveraging internal function features for configuration of functions on serverless edge platforms," *Sensors*, vol. 23, no. 18, 2023, Art. no. 7829.

[24] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 1–14.

[25] LLC Google, "Cloud functions pricing," 2024. [Online]. Available: https://cloud.google.com/functions/pricing

[26] C. Munns, "Aws re:invent 2020: What's new in serverless," 2020. [Online]. Available: https://youtu.be/aW5EtKHTMuQ?t=339

[27] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *J. Basic Eng.*, vol. 86, no. 1, pp. 97–106, 1964.

[28] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of Bayesian methods for seeking the extremum," *L Dixon G Szego. Toward Glob. Optim.*, vol. 2, 1978, Art. no. 117.

[29] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Glob. Optim.*, vol. 13, no. 4, pp. 455–492, 1998.

[30] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *J. Glob. Optim.*, vol. 21, no. 4, pp. 345–383, 2001.

[31] Y.-K. Kwok and L. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.

[32] Amazon Web Services, Inc., "Configure lambda function timeout," 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html

[33] S. Eismann et al., "A review of serverless use cases and their characteristics," 2021. [Online]. Available: https://research.spec.org/news/2020-05-29-11-38-technical-report-on-a-review-of-serverless-use-cases-and-their-characteristics-published/

[34] Amazon Web Services, Inc., "Configure lambda function memory," 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

[35] A. Casalboni, "AWS lambda power tuning," 2023. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[36] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. 2023 ACM Symp. Cloud Comput.*, 2023, pp. 324–340.

[37] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, "Harmonybatch: Batching multi-SLO DNN inference with heterogeneous serverless functions," in *Proc. 2024 IEEE/ACM 32nd Int. Symp. Qual. Serv.*, 2024, pp. 1–10.

[38] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λ*DNN*: Achieving predictable distributed DNN training with serverless architectures," *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 450–463, Feb. 2022.

[39] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, "λgrapher: A resource-efficient serverless system for GNN serving through graph sharing," in *Proc. ACM Web Conf.*, 2024, pp. 2826–2835.

[40] H. Yu et al., "Libra: Harvesting idle resources safely and timely in serverless clusters," in *Proc. 32nd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2023, pp. 181–194.

[41] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2020, pp. 265–276.

[42] 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 *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, USENIX Association, 2022, pp. 303–320.

[43] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *Proc. 21st USENIX Symp. Netw. Syst. Des. Implementation*, USENIX Association, 2024, pp. 167–183.

[44] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 329–338.
[45] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Comput. Appl.*, vol. 32, no. 13, pp. 9745–9760, 2020.
[46] K. Rzadca et al., "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
[47] V. Millnert and J. Eker, "HoloScale: Horizontal and vertical scaling of cloud resources," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Comput.*, 2020, pp. 196–205.
[48] X. Bu, J. Rao, and C.-Z. Xu, "Coordinated self-configuration of virtual machines and appliances using a model-free learning approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 4, pp. 681–690, Apr. 2013.
[49] L. Yazdanov and C. Fetzer, "VScaler: Autonomic virtual machine scaling," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 212–219.
[50] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with ELASTICDOCKER," in *Proc. IEEE 10th Int. Conf. Cloud Comput.*, 2017, pp. 472–479.
[51] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the use of fuzzy modeling in virtualized data center management," in *Proc. 4th Int. Conf. Autonomic Comput.*, 2007, pp. 25-25.
[52] P. Padala et al., "Automated control of multiple virtualized resources," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 13–26.
[53] S. Spinner et al., "Runtime vertical scaling of virtualized applications via online model estimation," in *Proc. IEEE 8th Int. Conf. Self-Adaptive Self-Organizing Syst.*, 2014, pp. 157–166.
[54] The Kubernetes Authors, "Scheduling framework | kubernetes," 2024. Accessed: Feb. 19, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/
[55] A. Suresh and A. Gandhi, "FnSched: An efficient scheduler for serverless functions," in *Proc. 5th Int. Workshop Serverless Comput.*, ACM, 2019, pp. 19–24.
[56] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, 2021.
[57] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "Faasrank: Learning to schedule functions in serverless platforms," in *Proc. 2021 IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst.*, 2021, pp. 31–40.
[58] C. Curino et al., "Hydra: A federated resource manager for data-center scale analytics," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 177–192.
[59] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: Principled and practical scheduling for serverless functions," in *Proc. 13th Symp. Cloud Comput.*, ACM, 2022, pp. 289–305.
[60] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, "AuctionWhisk: Using an auction-inspired approach for function placement in serverless fog platforms," *Softw.: Pract. Experience*, vol. 52, no. 5, pp. 1143–1169, 2022.
[61] Q. Chen et al., "YuanRong: A production general-purpose serverless system for distributed applications in the cloud," in *Proc. ACM SIGCOMM 2024 Conf.*, ACM, 2024, pp. 843–859.
[62] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. 2018 USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.
[63] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 386–400.
[64] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1069–1078.
[65] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 753–767.
[66] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 467–481.
[67] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, USENIX Association, 2020, pp. 419–434.
[68] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput.*, 2022, pp. 140–149.
[69] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 1063–1075.
[70] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, "Estimating the capacities of function-as-a-service functions," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput. Companion*, 2021, pp. 1–8.
[71] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 59–72, Jan. 2024.

Thomas Pusztai is currently working toward the PhD degree with the Distributed Systems Group, TU Wien, Austria. Prior to joining TU Wien as a Phd student, he was a senior developer in industry. His research interests include serverless computing, 3D computing continuum, software engineering, and reliability engineering. He has experience in key roles in EU and industrial projects.

Stefan Nastic (Member, IEEE) received the PhD degree in programming, provisioning, and governing IoT cloud systems from TU Wien. He is currently an Assistant professor on a tenure track with Distributed Systems Group, TU Wien, Austria. He has a track record as a lead researcher, consultant, and technical coordinator working on various research and commercial projects for more than a decade. His research interests include serverless computing, 3D computing continuum, Compound AI & Edge AI, and reliability engineering.