With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions

Muhammad Bilal*
IST(ULisboa)/INESC-ID and UCLouvain

Marco Canini
KAUST

Rodrigo Fonseca
Azure Systems Research

Rodrigo Rodrigues
IST(ULisboa)/INESC-ID

Abstract

Current serverless offerings give users limited flexibility for configuring the resources allocated to their function invocations. This simplifies the interface for users to deploy serverless computations but creates deployments that are resource inefficient. In this paper, we take a principled approach to the problem of resource allocation for serverless functions, analyzing the effects of automating this choice in a way that leads to the best combination of performance and cost. In particular, we systematically explore the opportunities that come with decoupling memory and CPU resource allocations and also enabling the use of different VM types, and we find a rich trade-off space between performance and cost. The provider can use this in a number of ways, e.g., exposing all these parameters to the user; eliding preferences for performance and cost from users and simply offer the same performance with lower cost; or exposing a small number of choices for users to trade performance for cost.

Our results show that, by decoupling memory and CPU allocation, there is the potential to have up to 40% lower execution cost than the preset coupled configurations that are the norm in current serverless offerings. Similarly, making the correct choice of VM instance type can provide up to 50% better execution time. Furthermore, we demonstrate that providers have the flexibility to choose different instance types for the same functions to maximize resource utilization while providing performance within 10-20% of the best resource configuration for each respective function.

*Currently with Unbabel. Work done in part while author was interning at KAUST.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567506

CCS Concepts: • **Computer systems organization → Cloud computing**.

ACM Reference Format:
Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567506

1 Introduction

The serverless programming model has flourished in the last few years, mainly because it allows developers to concentrate on the application logic and not worry about scalability and resource management. Developers only have to create cloud functions, and have little to no control over the infrastructure where those functions run. Cloud providers take care of provisioning, deployment, scalability, and maintenance of the resources required for each function invocation.

A particular aspect of this lack of control is the fact that the current serverless computing interface typically couples memory and CPU resource allocations together. Both AWS [4] and GCP [25] provide preset resource allocation configurations: AWS assigns a CPU share proportional to the amount of requested memory (in a fine-grained way, but up to 10 GB); GCP provides seven preset resource allocation options to choose from. Azure Functions [36] guarantees at least 1 vCPU core to each function instance and allows up to 1.5GB of memory per function instance (charging based on the actual memory consumption). Additionally, most cloud providers do not provide users the option to select the type of CPU for their serverless functions,[1] even though the VM type used to run serverless functions is not always the same [34, 53].

This simple interface is one of the defining characteristics of serverless computing, with the advantage of removing the configuration burden from the user. This is in stark contrast with the complexity of selecting and provisioning machines from the *hundreds* of configuration options and VM types in

[1] Lately AWS has started offering Lambda functions on ARM processors [5].

regular IaaS offerings [2, 55]. However, it also comes with significant disadvantages.

First, as we show later, in most cases, there are configurations that achieve better performance, better cost, or both, than the ones from current offerings. Second, when there are knobs such as the ones mentioned above, they are low-level: it is difficult for most users to translate memory and CPU to performance and cost. Third, the lack of full transparency hurts predictability, and even raises the question of whether the cloud provider is making the right choices to optimize resource usage, translating into lower costs for the users.

In this paper, we take a step back and rethink, from a clean slate, the allocation of resources for the execution of serverless functions. To achieve this goal in a principled way, we conduct a feasibility study to precisely characterize the gains that can be obtained by taking fine-grained control over the individual allocation of CPU, memory, and VM type to each serverless invocation. The key challenge of this study is to understand how these allocation decisions influence the trade-off between performance and monetary cost, the predictability of these metrics, and the ability to meet target execution times. Furthermore, we need to understand what is the minimal resource allocation that is required to meet such targets, if possible leveraging idle resources, whose type and availability may vary with time.

Despite the benefits of flexibility, simply offering a much larger configuration space to users negates the advantages of simplicity. Therefore, we also provide a thorough study of the effectiveness of *black-box* optimization algorithms to automatically determine the right resource configuration, to remove the need for the user to deal with the complexity of profiling their functions and choosing the right resources. The output of these algorithms can then be used in two possible ways: either by a cloud provider to *automatically* allocate resources for a user's functions, or by the serverless end-user if the cloud provider provides a set of *configuration knobs*. In addition, we address the problem of determining what is the right interface when these configuration knobs are exposed to the end-user. The challenge here is to strike a good balance between maintaining simplicity and giving the user control of both cost and execution time. Our proposed interface allows an auto-tuning framework built using black-box optimization algorithms to determine a set of optimal points in the trade-off space, allowing the user to only choose between a small number of possibilities.

Finally, if automatic resource configuration is provided by the cloud vendor, we study the benefits of using spare resources in a principled way. To this end, we determine whether cloud providers can allocate functions on different VM types, including those that leverage spare resources, thus minimizing cost while providing predictable performance.

Our experimental analysis shows that a flexible resource allocation provides up to 40% better execution time and 50% better execution cost than the restrictive resource allocation strategy that cloud providers currently use. We also show that black-box optimization algorithms can reach within 10% of the performance of the best configuration in our resource allocation search space within 20 optimization trials. Finally, we show that cloud providers can reduce their costs (or conversely increase resource utilization) by utilizing different instance types for the same function, while providing performance within 10% of the best configuration.

Resource Configuration	Values
CPU share	[0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
Memory limit (MB)	[128, 256, 512, 768, 1024, 2048]
Instance types	[c6g, m6g, c5, m5, c5a, m5a]

Table 1. Resource allocation search space (288 configurations). A configuration is a value for CPU share, memory limit and instance type.
Instance type prefix: 'c' = compute-optimized; 'm' = general-purpose / suffix: 'g' = Graviton2 ARM-based; 'a' AMD-based / no suffix = Intel-based.

Contributions. We make the following contributions:

- We determine the ground truth about the execution time and execution cost of 6 serverless applications across 288 resource configurations and multiple inputs (§2). Our benchmarks and data are available as open source [10].
- We analyze the potential benefits of enabling a more flexible resource allocation for serverless functions (§4).
- We analyze the accuracy of four Bayesian Optimization algorithms for determining the best resource allocation for two optimization objectives: execution time and cost (§5).
- We verify whether the serverless functions of our study have data-dependent performance characteristics (§5.4).
- We propose a set of possible interfaces for enabling the user to benefit from multi-objective optimization (§6.1).
- We evaluate the cost reduction opportunities for the cloud providers by using different instance types while providing predictable performance (§6.2).

2 Motivation

Resource allocation decisions can significantly impact the performance and execution cost of serverless functions. To characterize this performance and cost variation, we exhaustively test six benchmark functions on the resource configuration space defined in Table 1. (We defer to §3 the detailed setup for these experiments, including a description of the functions.) We run these benchmarks on AWS and adopt their nomenclature of instance families. As AWS does not provide information about the per-core or per-GB price, we calculate the execution cost based on a set of assumptions encapsulated in a methodology defined in §3.

A resource allocation choice specifies a CPU share, a memory limit, and the instance type.[2] The CPU share is the time-share of a vCPU allocated to the function. For example, a

[2]Throughout the paper, we use the term instance type to refer to the type of VM instance, which in practice corresponds to a given type of CPU, since the remaining differences between VM instances are irrelevant in this work.

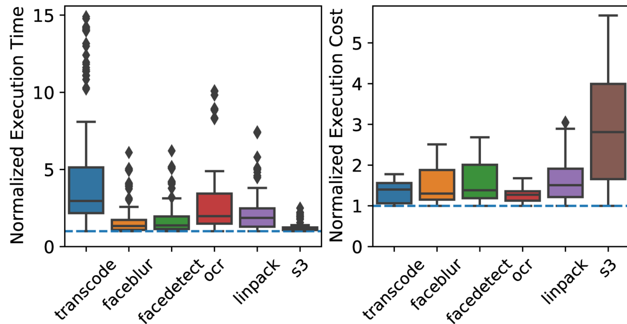

Figure 1. Execution time and cost of each function across the entire configuration space defined in Table 1, normalized w.r.t. the best configuration of each function.

share of 0.25 means that a quarter of vCPU time is allocated to the function whereas a share of 1 or 2 implies allocations of 1 and 2 entire vCPUs, respectively. The memory limit, expressed in MB, is simply the amount of memory allocated to the function. In addition, we run functions on a diverse range of instance types. The caption of Table 1 lists their nomenclature.

Figure 1 shows a boxplot[3] of the normalized execution time and execution cost for each function across the entire configuration search space of Table 1. For each function, the normalization is done w.r.t. the best (minimum) execution time and execution cost for that function in the search space.

We observe that, in the worst case, selecting the wrong configuration can lead to up to 14.9× worse execution time and 5.6× worse execution cost than the best configuration. A somewhat different effect happens in the s3 application, which has a small variation in execution time but a significant variation in execution cost. This is because s3 is a network bound application, and therefore increasing the resources (CPU and memory) assigned to it leads to minor improvements in the execution time beyond a certain point, but to an increase in the cost for the additional resources.[4]

3 Experimental setup

To understand the impact of different resource allocation decisions on the cost and performance of running serverless functions, we use the OpenFaaS [39] serverless framework to execute and measure the performance of a diverse range of benchmark functions across the resource allocation search space. We deploy OpenFaaS atop the k3s distribution [29] of Kubernetes as a cluster running on AWS EC2 instances. We prepare multi-architecture Docker containers (amd64 and arm64) for all benchmark functions using *docker buildx* without using any platform-specific optimized libraries.

Data. We collect ground truth performance and cost data as follows. We execute each function using multiple input data samples on every resource allocation in the configuration search space of Table 1. For every resource allocation, a single instance of each function is deployed in isolation on the desired instance type to measure its performance. To minimize the impact of performance outliers, we execute each function at least 5 times on a given configuration. We use a function execution timeout of 600s, which is comparable to the timeouts in current serverless offerings [54]. The input samples were randomly chosen from publicly available datasets. We set one input sample as the default, but we analyze how the performance depends on input data (a modest 20% at most, §5.4). We run over 5,000 combinations of resource configurations, benchmarks and input samples. We identify the overall best configuration for each function with regard to both execution time and execution cost.

Benchmark functions. From publicly available code, we select a diverse set of applications based on their characteristics and resource usage, i.e., combining applications that are single-threaded, multi-threaded/multi-process, require storage access, and have different memory requirements. In particular, these benchmarks are based on functions from FunctionBench [30] and from the OpenFaaS function store [38]. Note that we do not use the entire OpenFaaS function store repository: we exclude applications with extremely short execution time and low resource requirements because resource allocation optimization for these applications is not useful. Table 2 details the chosen benchmark functions.

transcode's and ocr's request handlers are in Python but use internal bindings to invoke C and C++ code, respectively. facedetect and faceblur use Go libraries [46, 47]. linpack uses an application from FunctionBench [31]. Both transcode and ocr effectively utilize more than 1 vCPU, as shown in Column P of Table 2. Since s3 downloads and uploads objects to an AWS S3 bucket, its performance mainly depends on network resources.

These six benchmarks provide a good mix of applications that have different resource requirements, as shown in Table 2, based on our experiments. We determined these resource requirements based on the observed performance characteristics of the applications, i.e., resources required to run the application (memory configuration) and how their execution time varies with an increase of different resources. Table 2 also shows the best configuration for both the execution time (ET) and execution cost (EC) objective functions in the search space of Table 1, as reference.

Cost model. To perform a cost analysis in our resource allocation setting, we need a pricing function per-vCPU and per-GB of memory for different instance types. However, we cannot infer this information from the pricing models of the current serverless offering because FaaS providers either

[3]All boxplots in the paper show median, 1st and 3rd quartiles with whiskers showing the distribution 1.5× IQR past the high and low quartiles and anything beyond is shown as outliers.

[4]This effect can also be observed from the best configuration for s3 for the execution time and execution cost objectives in Table 2.

Name	Purpose	Language	Important Resources				Best Configurations					
							ET			EC		
			C	P	M	N	CPU	Mem	Inst.	CPU	Mem	Inst.
facedetect	Image face detection	Go	•				1.0	2048Mi	c5	1.0	512Mi	m6g
faceblur	Image face blurring	Go	•				1.0	768Mi	c5	0.5	256Mi	c5a
transcode	Video transcoding	Python & C	•	•	•		2.0	1024Mi	c5a	2.0	512Mi	m6g
ocr	Optical character recognition	Python & C++	•	•			2.0	512Mi	c5	2.0	256Mi	m6g
linpack	Solve linear equations	Python	•		•		2.0	512Mi	c5a	0.5	512Mi	c5a
s3	Download then upload an object from one S3 bucket to another	Python				•	2.0	256Mi	c5a	0.25	128Mi	m6g

Table 2. Benchmark serverless functions and their best configurations for Execution Time (ET) and Execution Cost (EC) objectives (using median values) in our configuration search space. Each benchmark has certain resources that are more important than others, either for performance or to avoid function failure. Resources: C: CPU, P: parallelism, M: memory, N: network.

bundle the pricing of CPU share and allocation memory in a single price per unit of time or provide the CPU share pricing for only a single CPU type. For example, AWS lambda charges a fixed price for each GB-second (per 1ms) of function execution [6][5], while Google Cloud Functions provides the decomposition of the total price of Cloud functions into the price for each GB-second and GHz-second, the latter being provided without specifying the CPU type [24]. Thus, we cannot determine per-vCPU, per-GB of memory, or per-instance type pricing using the current pricing schemes of FaaS providers. Consequently, we must create a reasonable pricing model that allows for the fine grained accounting of CPU and memory consumption.

To this end, and since we deploy OpenFaaS on top of EC2, we use the AWS EC2 instance pricing so that, based on the overall pricing of various different instance types, we infer a per-vCPU and per-GB memory cost for those types. In particular, we calculate per-vCPU and per-GB memory cost by solving a system of linear equations for instances with the same instance type. Each equation is of the form below, defining the instance price $P_{instance}$ (given as input) as a sum of per-vCPU cost X and per-GB of memory cost Y:

$$\alpha X_{vCPU} + \beta Y_{mem} = P_{instance} \quad (1)$$

To create a well-defined system of equations, we use publicly available information from AWS to determine the number of vCPUs α and the amount of memory in GB β for a sufficient number of equations. For example, x86 instances m5, c5, and r5 are assumed to have the same per-GB memory cost Y_{mem}. Moreover, m5 and r5 instances have same CPU. Thus, we have X^1_{vCPU} for c5 and X^2_{vCPU} for m5 and r5. This yields a system of equations with 3 unknowns and 3 equations, using $\alpha = 2$ and $\beta = 4, 8, 16$ for c5, m5, and r5 instances, respectively. While we do not use r5 instances, its pricing information is also used to solve the system. (The same approach was used for ARM and AMD instances.)

[5] AWS lambda now offers ARM-based serverless functions, but that still does not give us the complete pricing information we need.

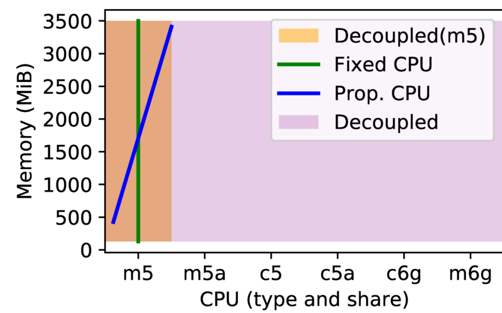

Figure 2. Four strategies for configuring serverless functions. Each strategy defines a search space, corresponding to a subset of the possible resource allocation configurations.

Note that we are not arguing for cloud providers to adopt this specific pricing scheme. Instead, our goal is to come up with reasonable pricing for an execution in our scenario, given current instance costs. Generally, cloud providers add a premium for managed services on top of the infrastructure costs. Prior work [33] provide guidelines to create the optimal pricing for serverless from the provider's perspective.

4 How good is flexible resource allocation?

To understand the opportunities that current serverless offerings are missing, we start by characterizing the best execution time and cost for different resource allocation options.

4.1 Larger search spaces yield advantages...

Setup. We consider four strategies for resource allocation, with an increasing level of flexibility. Each strategy corresponds to a subset of the configuration search space, as depicted in Figure 2. The first three strategies assume a fixed instance type, corresponding to the EC2 m5 instance in our experiments.

Fixed CPU allocates a single-vCPU for each function instance, whereas the memory is charged based on the average actual consumption. This strategy is inspired by Azure Functions.

Prop. CPU allocates a share of CPU proportional to the

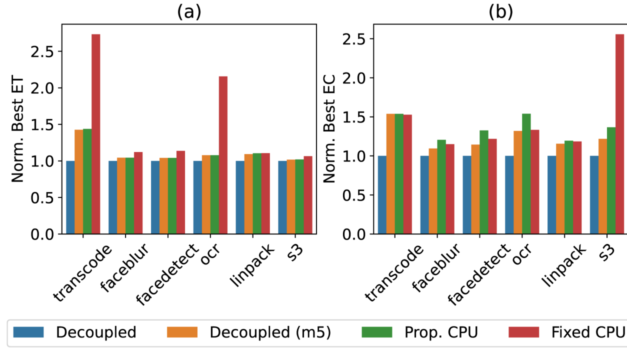

Figure 3. Potential gains within each search space. The graphs show the best (a) Execution Time (ET) and (b) Execution Cost (EC) of each function across different search spaces, normalized to the overall best configuration.

amount of memory selected. This strategy is inspired by AWS Lambda and Google Cloud Functions.

Decoupled (m5) decouples CPU and memory allocations. In this case, the search space includes all CPU and memory values in Table 1, but it uses only the default m5 type.

Decoupled has the largest search space, encompassing all other strategies and covers the search space in Table 1.

Results. We measure the best execution time and execution cost for each benchmark function in each of the different resource allocation search spaces. Figure 3 shows the best execution time (ET) and execution cost (EC) of each strategy's search space normalized w.r.t. the best possible ones in *Decoupled*, since its search space includes all others.

Observations. Figure 3a shows that, by using different instance types, *Decoupled* can provide 5%-40% better execution time than *Decoupled (m5)* and *Prop. CPU*. In most cases, the execution time improvement is not significant, but the decoupled strategy achieves improved execution times at lower cost compared to a proportional resource allocation. These gains are primarily due to the use of different instance types. In addition, decoupling memory and CPU in *Decoupled (m5)* is sufficient to provide 10%-50% better execution cost compared to *Prop. CPU*, as shown in Figure 3b.

Fixed CPU leads to 2.7× and 2.1× higher execution time for transcode and ocr, respectively, and 2.6× higher execution cost for s3. This is because, for transcode and ocr, having 1 fixed vCPU per function invocation does not exploit available parallelism opportunities. For s3, *Fixed CPU* has higher execution cost because the function is not compute intensive and its execution time already plateaus with a CPU share lower than 1.

Takeaways: Using different instance types allows for improving the execution time, while decoupling CPU and memory enables potential improvements in execution cost compared to currently deployed resource allocation strategies.

4.2 ... and potential to use other instance types

The compute infrastructure of a cloud provider includes dozens of different instance/VM types. Another potential advantage of flexible resource allocation for serverless functions is that a cloud provider might be able to use underutilized instance types, even if the instance type is not the best for a given function. *Decoupled* gives us the opportunity to use different instance types, when doing so provides sufficient performance for a given objective.

Setup. To quantify the number of different instance types (from our search space) that provide satisfactory performance, we use multiple performance thresholds (θ) w.r.t. the best configuration and multiple performance objectives based on weighted combinations of execution time and cost. We use the execution time data collected across all the configurations in our search space to calculate the value of each performance objective for each configuration and benchmark function.

Results. Table 3 shows the number of instance types that have at least one configuration that is within θ% of the best *Decoupled* configuration. The results demonstrate the potential of using alternative instance types while providing comparable performance for different performance objectives for each function. The objectives are execution time (left), execution cost (right), and three weighted combinations of the two (denoted with W_t and W_c for execution time and cost, respectively).

Observations. We highlight two types of special cases in Table 3. The cells in red indicate cases where there is no other instance type able to reach within θ% of the best configuration. The cells in blue denote cases where all instance types have at least one configuration that provides performance within θ% of the best configuration. The likelihood of being able to use available resources of an alternative type clearly increases with the number of alternative instance types and depends on the function as well as the parameter θ. The results show a prevalence of white and blue cells, indicating the existence of alternative deployment options for serverless functions, at a small sacrifice in the objective function value.

Takeaways: In our experiments, we found that in most cases (28 out of 30 settings in Table 3) there are opportunities to use idle or underutilized instances of different types while providing performance within 10% of the best configuration.

5 Is automatically discovering good configurations possible?

While more flexibility in the choice of resources is beneficial, the task of determining the right configuration from a large search space can be daunting, requiring performance modeling and/or profiling. This is in contrast with the goal of serverless — to relieve the user from the burden of managing resources. Thus, we explore the effectiveness of using

Benchmark	Execution Time			W_t = 0.25 & W_c = 0.75			W_t = 0.5 & W_c = 0.5			W_t = 0.75 & W_c = 0.25			Execution Cost		
	Threshold (θ)			Threshold (θ)			Threshold (θ)			Threshold (θ)			Threshold (θ)		
	5%	10%	20%	5%	10%	20%	5%	10%	20%	5%	10%	20%	5%	10%	20%
ocr	2	4	5	1	1	2	1	1	4	1	3	4	1	1	2
transcode	0	0	2	2	2	2	2	2	2	0	2	2	1	2	2
faceblur	1	2	2	0	1	3	1	1	3	1	1	2	0	3	4
facedetect	1	4	4	1	1	3	2	3	3	2	3	3	1	1	3
linpack	2	2	4	0	2	3	0	2	3	1	1	4	1	1	3
s3	3	5	5	0	1	5	1	4	5	0	4	5	0	0	3

Table 3. Number of alternative instance types with one or more resource allocation configurations with the performance metric within threshold (θ) of the best configuration in the *Decoupled* search space. The cells in red indicate cases where there is no alternative instance type able to reach within θ% of the best configuration. The cells in blue denote cases where all instance types have at least one configuration that provides performance within θ% of the best configuration.

black-box optimization algorithms to automatically determine the right allocation of resources (either transparently by the provider or by a library used by serverless users).

5.1 Background on optimization techniques

Recent works developed techniques for automatic cloud configuration [2, 11, 12, 28, 51]. At the core of many of these approaches there lie various optimization techniques ranging from model-based optimization algorithms to sampling-based search techniques. These approaches are also called black-box optimization algorithms, as they consider the objective function as a black-box: one may evaluate it at specific points (using profiling runs), but the techniques make little or no assumptions about the function. In contrast, other works (e.g., Ernest [51]) rely on analytical modeling to create a mathematical model dependent on the characteristics of the application and thus varies with each application.

We believe that black-box optimization methods are a good fit for serverless, since the search space is large, and function invocations can provide good performance indicators, while keeping the objective function as a black-box. Even though the resulting configurations may be, in some cases, worse than using analytical models, we believe that the advantage of being able to quickly reuse existing algorithms outweighs those limitations.

We next briefly review a range of optimization techniques that we use and point out a relevant change when using these methods for the serverless scenario.

Model-based algorithms build a model of the underlying black-box objective function. Following a comprehensive study [12], we adopt the best-performing method, which is Bayesian Optimization (BO). We consider four variants of the surrogate model (required for approximating the objective function): (1) Gaussian Processes (GP), (2) Gradient Boosted Regression Trees (GBRT), (3) Random Forests (RF), and (4) Extra Trees (ET). In all cases, we use the popular Expected Improvement (EI) as the acquisition function. We use the

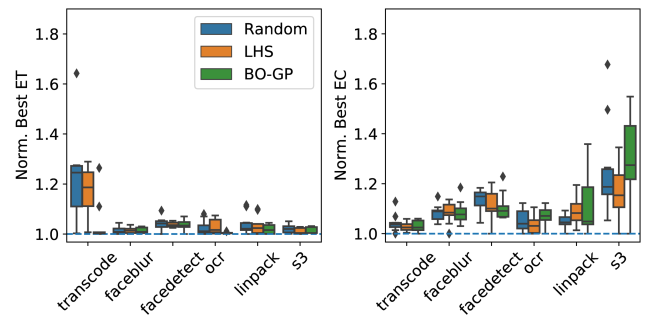

Figure 4. Performance of the best-found configurations for sampling-based and model-based algorithms.

Scikit-Optimize Python library [44] with its default parameters unless otherwise stated. By default, based on previous findings [2], we use three random initial samples.

Sampling-based search techniques sample the search space to find good configurations. These methods are easy to parallelize. We use both Random Sampling as well as Latin Hypercube Sampling (LHS) [35]. LHS samples the search space using a space filling design to create samples from a given search space. We use pyDOE [41] to generate LHS samples.

5.2 Adapting to the serverless scenario

An issue that arises when using the above techniques out-of-the-box is that they can produce configurations on which the function fails because not enough memory is allocated. At first, we attempted to address this by assigning a large value to represent the performance objective (e.g., execution time) of a failed function invocation. However, that created a non-smooth underlying function, which affects the quality of the optimization.

To address this, we resorted to pruning the search space to remove the resource configurations with memory less than or equal to every memory configuration for which we determine that the function failed. This is based on the simple property that if a function fails for a certain memory limit, it will likely fail with a lower memory limit. Thus,

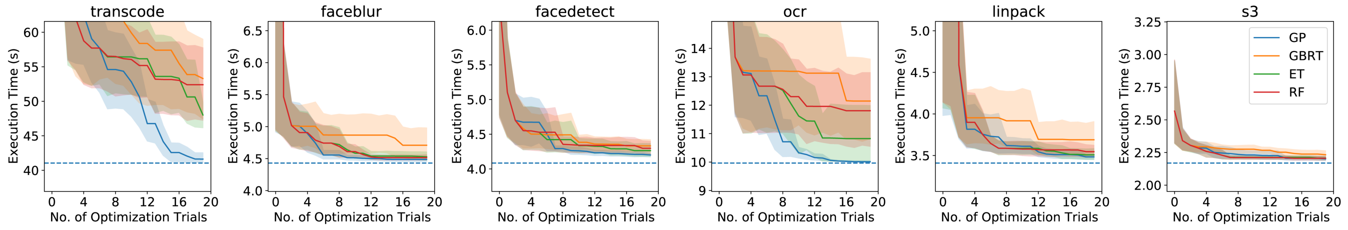

Figure 5. Execution time performance of the best found configuration as optimization by different BO variants progresses.

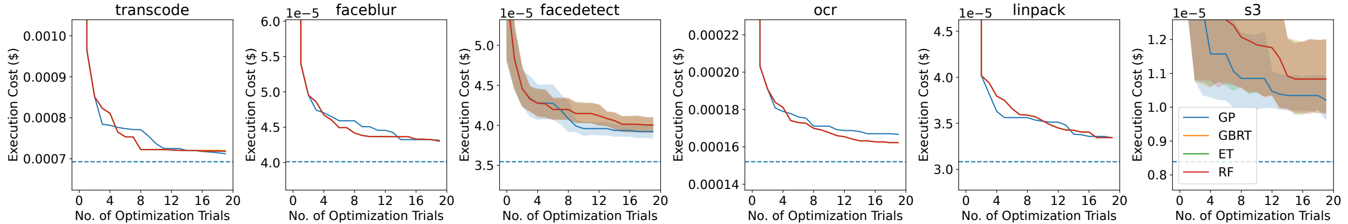

Figure 6. Execution cost performance of the best found configuration as optimization by different BO variants progresses.

every time we record a function failure, the search space is dynamically pruned, removing all configurations that would lead to failure due to a lower memory limit.

External failures (e.g., bugs or unreliable external services) can also contribute to function failure, but these can be hard to detect without application-specific knowledge. The easiest way to handle such cases is to repeat the experiments, which increases cost at the expense of providing more reliable data when the services are unreliable.

Next, we present the methods and findings of our study.

5.3 Optimization techniques are effective

Sampling-based vs. model-based algorithms. We first study how well black-box algorithms optimize the resource allocation as compared to the ground-truth best configuration in the *Decoupled* search space. We fix a budget of 20 trials for each method and report on the best execution time and cost found. For the sampling-based methods (Random Sampling and LHS), this entails generating 20 samples. For the Bayesian Optimization (BO) methods, we start with 3 initial samples and then use the acquisition function to repeatedly sample from the search space a series of configurations to test until the budget is matched. The BO methods learn a mapping from the resource allocation of the function (input features) to the function's execution cost or time (objective function). We repeat the process 10 times using different random seeds (for sampling) and initial samples (for BO). The search space sampling and the optimization runs can be done either online, based on the first set of user requests (§5.5) or based on a profiling run, which can use either a sample of past user requests, or inputs that were created specifically for that purpose. Different methods have their pros and cons (e.g., real user requests may generate side effects during profiling).

Figure 4 shows the best-found execution time and execution cost normalized w.r.t. the best configuration in the search space. The boxplot captures the variation in the best-found configuration across different repetitions. For these results, we only show the values for BO with GP, because this method outperforms other BO variants.

We observe that both sampling methods and BO with GP perform comparably in most cases. While BO with GP finds a better execution time for transcode (Figure 4 left), both sampling methods find better execution cost for s3 (Figure 4 right). We got similar results for weighted combinations of execution time and cost (omitted due to space limitations).

Thus, while sampling- and model-based methods perform comparably in finding the best configuration for an objective function, model-based methods have the additional benefit of generating performance models that can provide predictions for yet-unseen configurations. Therefore, given the overall good performance of model-based methods and the importance of predicting configurations with larger search spaces, we focus on model-based methods next.

Convergence speed of model-based algorithms. We now analyze how fast model-based methods converge towards the best configuration in the search space for each benchmark function. This analysis offers an early indication of how many optimization trials would be necessary as a baseline, as we later turn to online optimization in §5.5.

We run the four BO variants for 20 steps (including the 3 initial samples), using execution time and cost as the performance objective. Figures 5 and 6 show the execution time and execution cost, respectively, of the best-found configuration as the optimization progresses. The dashed lines denote the overall best execution time (resp. execution cost) in the search space. We repeat the experiment 10× for each optimization method using different random initial samples. The shaded area is the 95th percentile confidence interval.

With regard to execution time, while almost all optimization algorithms perform comparably, BO with GP overall tends to outperform other methods and reaches within 5% of the best execution time in 20 optimization trials, in all cases.

When optimizing for execution cost, the best configurations are harder to find and thus there is a larger gap between the best-found configurations and the overall best ones in the search space. This is particularly true for s3 and facedetect. In 5 out of 6 benchmarks, BO with GP finds configurations that are within 20% of the best execution cost in the search space. However, for s3, it only finds configurations that are within 30%. As with execution time, BO with GP either outperforms or performs comparably to other BO variants.

Takeaways: While both sampling-based search and model-based optimization methods find good resource allocations, we favor model-based methods as they provide predictions for untested configurations. In that context, different BO variants perform comparably when optimizing for execution time and execution cost, with an advantage for BO with GP.

5.4 Input data variation has modest influence

The performance of a function in general depends on its input data. For instance, transcode significantly depends on the input video dimension whereas facedetect and faceblur depend on the input image size. Indeed, all of our benchmark functions have input data-dependent execution time.

Dealing with data dependence is challenging. One approach would be to create data-specific performance models that account for the input data characteristics. However, this adds significant complexity to the optimization framework. First, the model needs to be sophisticated enough to encapsulate these performance-determining characteristics, which is especially difficult if the relation between input data and execution time is complex (e.g., if it has many modalities that require advanced profiling [42]). Second, even utilizing, e.g., a performance model is difficult because, at invocation time, the serverless framework would need to evaluate with minimal overhead the input data and route accordingly.

A simpler approach instead is to use a generic model using representative input data samples. This is based on the intuition that, even though the absolute performance may vary, *a good configuration for one input data sample might also be a good configuration for other input data samples.*

We therefore study to what extent the resource configuration depends on input data and the effectiveness of this latter approach. To this end, we use the methods in §5.3 so that, for each function, we use the default input to create a single generic model as well as 10 input-specific models, one for each input sample. The different input samples used are available in our artifact repository [10]. The input samples vary in size from 74KB to 1.4MB for images, 4.3KB to 634KB for OCR images, and 3.1MB to 18MB for videos. Input samples for facedetect and faceblur also vary in terms of the number of faces in the images.

Figure 7 contrasts the performance of the best-found configuration for the two model types – generic (blue) vs. input-specific (orange) – for each input, together with the overall best configuration (green) in the search space (for that input). The input-specific scenario represents an artificial best-case scenario where the same input that is arriving was used to train the model. For clarity, we only show the results of 5 out of 10 input samples (except for linpack). We show the optimization scenario for execution time and note that optimizing for execution cost obtains similar results.

In our experiments, using input-specific models provided up to 20% better execution time. linpack with a $N = 7500$ matrix is a special case since it requires more memory and the optimization process using a default input leads to lack-of-memory failure in 3 out of 10 repetitions of the optimization process (which we exclude). However, this performance improvement comes with significantly increased complexity. In addition, it is unrealistic for each input to have its own model, and therefore we leave it as an open research direction to create more sophisticated performance models. Consequently, we adopt the generic models in the rest of the paper.

Note that these results are specific to our benchmarks and input data samples. It is easy to conceive scenarios where some input data samples would require more memory. For example, a 1GB video input for the transcode application would require significantly more memory than a 10MB video input. As a consequence, the memory allocation might need to be adjusted based on the input data, or, alternatively, should be set based on the biggest input video size expected. Finally, an alternative way that could, in some applications, handle this kind of scenario gracefully would be to split the data into smaller fixed-size chunks and process them independently. This technique will provide predictable performance without requiring adjusting configurations based on input data size. It is important to note that this problem is independent of whether resource allocation is automated or manual.

Takeaways: In general, the performance of a serverless function, depends on its input data. But, in our experiments, good configurations for one input sample were also good for others: i.e., the configuration that works best with input I_1 in terms of ET or EC is also the one that works best or close to best for other inputs I_2, I_3, ..., I_N. However, the absolute value for the objective function would naturally vary with different inputs in general. Input-specific optimization can improve ET, but since input-specific performance models are more complex to create and maintain, this trade-off must be considered carefully.

5.5 Online optimization is feasible

The optimization of resource configurations can occur in two ways: (1) offline or (2) online. Offline optimization requires running the optimization process described in §5.3

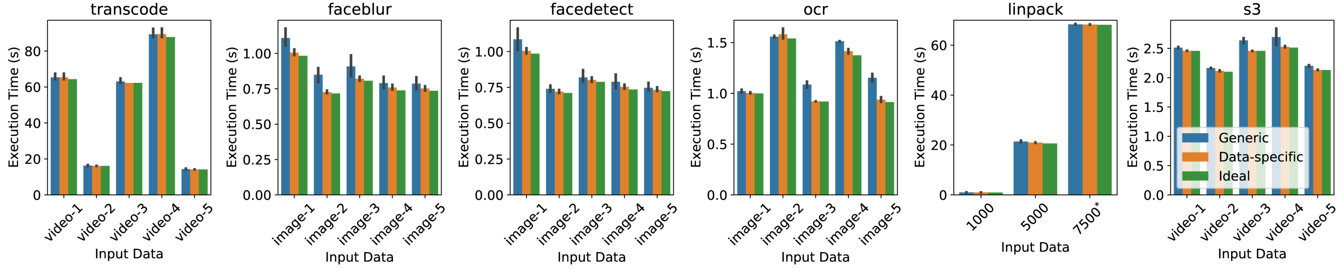

Figure 7. Execution time of the best configurations found by a generic optimization process (blue), a data-specific optimization process (orange), and the overall ideal configuration (green).

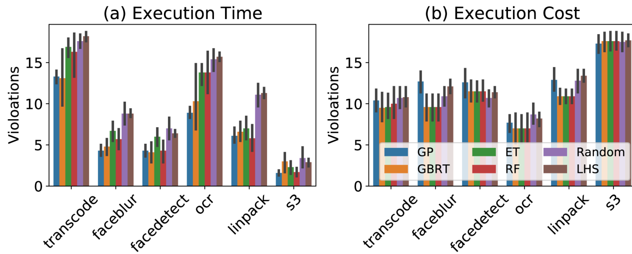

Figure 8. Average number violations during online optimization for (a) execution time (ET) and (b) execution cost (EC).

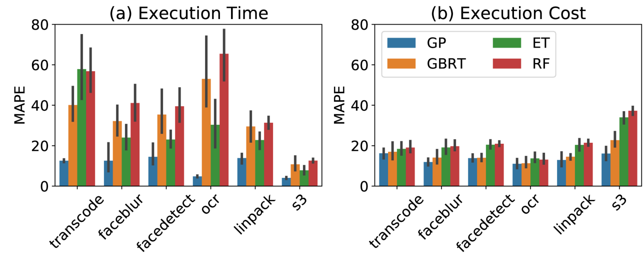

Figure 9. MAPE for different benchmarks and optimization methods for (a) execution time and (b) execution cost.

upon function deployment with representative input samples. Conversely, online optimization exploits function invocations in production as trials of the optimization process. However, in the online scenario, it is important to reduce the possibility of performance degradation due to trials with bad configurations. Thus, we now analyze which optimization methods lead to fewer degraded runs during optimization.

Figure 8 shows the average number of violations during 10 repetitions of the optimization process. Here we consider a violation when the performance objective is at or above 1.5× the objective value[6] for the best configuration in the search space. On average, BO with GP has a number of violations comparable to sampling-based search techniques for execution cost, but it has a lower number of violations for execution time. Overall, sampling-based search techniques incur in slightly more violations than model-based methods.
Takeaways: BO with GP has slightly lower average number of violations for execution time compared to other methods. For execution cost, it has slightly higher number of violations compared to other BO variants, but overall it leads to fewer violations than sampling-based online optimization.

5.6 Resource allocation models can predict performance of untested configurations

Predicting the performance of an untested configuration is harder than converging to the best performing configuration during the search. A low prediction accuracy for models built using Bayesian Optimization would suggest that sampling-based search methods, which are simpler, may be sufficient after all. Therefore, we now analyze how well model-based methods can predict the performance objective across configurations in two different scenarios: (1) over the entire *Decoupled* search space (except the failed runs), and (2) when the configuration prediction is restricted to match a particular instance type. This second scenario is relevant in the context of helping the cloud provider utilize idle resources of different instance types while providing predictable performance (as we elaborate in §6.2). We repeat every measurement 10 times and report average and 95th percentile confidence interval of the error metric.
Scenario 1. Figure 9 shows, for each BO variant, the Mean Absolute Percentage Error (MAPE) across all configurations between the actual execution time/cost and the predicted value. We observe that BO with GP has lower error than other optimization algorithms. Compared to other variants, BO with GP, on average, has up to 16× and 2.3× lower MAPE for execution time and execution cost objective, respectively.
Scenario 2. Figure 10 shows the MAPE across the best predicted configuration for each instance type. Similar to the previous case, BO with GP generally outperforms other BO variants in most cases, except for transcode and ocr when optimizing for execution cost. BO with GP, on average, has up to 7× and 3.5× lower MAPE than other variants for the execution time and execution cost objective, respectively.
Takeaways: BO with GP not only works well when it comes to convergence towards the best configuration (as shown

[6] 1.5× is an arbitrary objective threshold, chosen for illustrative purposes.

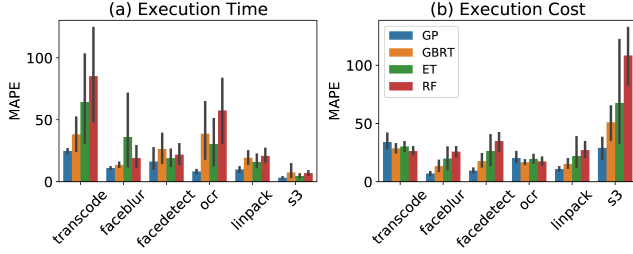

Figure 10. MAPE for different benchmarks and optimization methods when comparing the best configuration for each instance type, for (a) execution time and (b) execution cost.

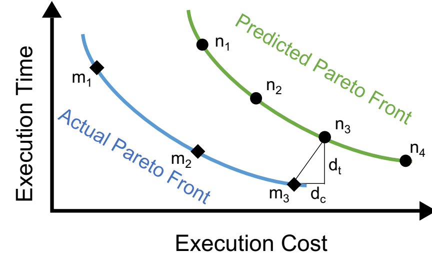

Figure 11. Distance between the configurations in predicted and actual Pareto fronts, with time and cost components.

in §5.3) but overall has lower error compared to models created using other BO variants. In our experiments, models built with BO with GP provide up to 16× lower MAPE than models build with other BO variants. This means that the performance models built with BO with GP are better at predicting the performance of untested configurations.

6 Automatic resource allocation from the provider's perspective

Having looked at the potential gains from flexible resource allocation, and the effectiveness with which we can exploit these gains using black-box optimization algorithms, in this section we discuss how the cloud provider can not only expose this to users without unduly complicating the 'serverless' interface, but also leverage model predictions to opportunistically select available instance types that can reduce costs without significantly sacrificing performance.

6.1 On the interface between user and provider

Given the performance and cost benefits of flexible resource allocations, the provider could expose the knobs for fine-grained resource configuration (selecting instance type, memory and CPU allocation separately). While possible, this would, however, shift the complexity of configuration selection back to the user, and negate one of the big advantages of serverless: its simplicity. Alternatively, the provider could abstract away that interface and allocate resources automatically and transparently. Furthermore, as we describe in the next contribution of this analysis, our automatic exploration of the configuration space additionally allows for the best of both worlds: a simple, high-level interface that exposes to the user in a simple way the cost/performance benefits of decoupled resource allocation.

More specifically, we describe three ways to select a trade-off between execution time and cost: 1) Providing configurations from the predicted Pareto front, 2) Weighted multi-objective optimization [9] to provide best configurations for different weights, and 3) Hierarchical multi-objective optimization [9] to satisfy a user-provided trade-off constraint.

Pareto front: We use the predictions from the black-box model to create a Pareto front and expose the configurations that have different trade-offs between execution time and cost. The predicted Pareto front is created by normalizing the execution time and cost so that the values of the two objectives are on a similar scale. Since we do not know the actual minimum value of the objectives, we use the respectively observed minimum values to perform normalization. Thus, two models have to be trained to create a Pareto front.

To assess the effectiveness of using the configurations from the predicted Pareto front, we measure the distance of those configurations to the nearest configuration in the actual Pareto front, as shown in Figure 11. We measure the distances in terms of normalized execution time (d_t) and cost (d_c) separately. We normalize d_t and d_c for each configuration using the corresponding objective value for the nearest configuration in the actual Pareto front.

Figure 12 shows, for each function, the average distance between the points in the predicted Pareto front and the actual Pareto front for the default input. In our experiments, the average difference between the configurations in predicted Pareto front and actual Pareto fronts is up to 20% (cost) and 25% (time). This difference is because of the prediction error in the model and thus it leads to $d_t > 0, d_c > 0$.

In this option, the interface to the user exposes the small set of configurations (between 2 and 10) in the Pareto front, with the corresponding predicted cost and execution times.

Weighted multi-objective optimization: With weighted multi-objective optimization, the cloud provider can select relative weights for execution time (W_t) and execution cost (W_c), where $W_c = 1 - W_t$. Using these weights, the cloud provider can form a weighted objective function $F_w(X)$ for a configuration X, from normalized objective functions for execution time ($F_t(X)$) and cost ($F_c(X)$):

$$F_w(X) = W_t \times \frac{F_t(X)}{B_t} + W_c \times \frac{F_c(X)}{B_c} \quad (2)$$

where B_t and B_c are the minimum values for execution cost and execution time objectives, found during the optimization process for $F_t(X)$ and $F_c(X)$. To simplify the process for the user, we pre-train three models with $W_t \in \{0.25, 0.5, 0.75\}$. The two models trained first for execution time and execution

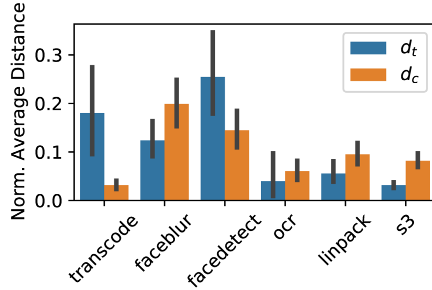

Figure 12. Normalized average distance between the points in the predicted Pareto front and the actual front for the default input. We show normalized execution cost (d_c) and execution time (d_t) components of the distance separately.

cost translate to $W_t = 1$ and $W_t = 0$, giving a total of five models and five best configurations (each suggested by every optimization process) for the user to choose from.

Figure 13 shows the best configurations found by the weighted multi-objective optimization (using BO with GP) as we perform more optimization trials. We can see that even in the weighted multi-objective setting, for most cases, the optimization process is able to find configurations that are within 20% of the best configurations in the search space after 20 optimization trials. While not shown here, we increased the number of trials and observed that within 40 trials, BO with GP is able to find configurations with performance within 5-10% of the best configuration, for all cases.

In this option, the interface is similar to the Pareto front one: in this case, the user chooses between at most 5 configurations, based solely on their predicted cost and performance.

Hierarchical multi-objective optimization: In hierarchical multi-objective optimization, we first optimize one of the objective functions (primary objective). Then, the optimized model can be used to find configurations that minimize the value for the second objective function (secondary objective) while degrading the primary objective value by at most a user-defined amount (θ). This allows for showing the users the best value for the primary objective function, and then they can decide if they are willing to degrade that value by up to θ% to improve the secondary objective.

Figure 14 shows the normalized value for execution time and execution cost metrics after the hierarchical optimization (satisfying user's constraints) for the two combinations of primary and secondary objective functions. We used a threshold of $\theta = 20\%$ for this experiment. ET/EC and ideal-ET/ideal-EC represent the best configuration using the prediction model and oracle-like knowledge. The normalization is done w.r.t. the best configuration found after optimizing the primary objective only. The dashed line shows the user-specified degradation threshold for the primary objective. ET and EC are both included in each scenario to indicate the execution cost and execution time of the best configurations found by the hierarchical optimization. In some cases, the prediction error leads to a higher degradation of the primary objective than the threshold. But in other cases, hierarchical optimization using the performance models performs comparably to the ideal case.

Unlike weighted multi-objective optimization, only one model needs to be trained for the hierarchical counterpart.

Takeaways: While certainly not a definitive answer, these three options serve as a starting point for a much needed discussion on how to give the user access to a broader space of configurations in serverless offers, without requiring the user to deal with complex resource allocation choices. In fact, these proposals do so while shifting the language from *resources* to *outcomes*: performance and cost. They also represent different trade-offs in terms of simplicity and effectiveness. With the Pareto front and the weighted multi-objective optimization, users simply get a small set of cost-performance tuples to select from. In our experiments, both methods found configurations that were only up to 30% worse than the best in both cost and performance. In turn, the hierarchical optimization offers a more explicit prioritization, but users have to choose the threshold themselves. It also has good results: for an increase of roughly 20% in the primary metric, a reduction of up to 50% in the other. They also present different costs for the provider: for Pareto front and hierarchical multi-objective optimization, we need to train 2 and 1 models, respectively. In turn, for the weighted multi-objective optimization, the number of models we need to train depends on the number of weighted combinations of ET and EC. A full evaluation of these interfaces would require user studies and a more thorough cost analysis, and we leave it as an open direction for future work.

6.2 Cost vs. performance of different instance types

Table 3 shows that there is potential for using different instance types while providing performance within a certain margin of the best found configuration. We now evaluate how effectively we can utilize that potential. In particular, we measure this benefit by translating utilization of idle resources into a cost decrease. Similarly to spot instances, we assume that a serverless instance type with many idle instances is assigned a lower cost, to incentivize the utilization of the idle resources. We assume that the spot pricing for the serverless instance will decrease the per-CPU and per-GB cost to a fraction of the original price.

Figure 15 shows the decrease in deployment cost that the cloud provider can observe by utilizing the best configurations for each instance type predicted by the model. For this figure, we assume that spot pricing is 20% of the normal pricing. Figure 15 shows the decrease in execution cost while the performance model is predicting configurations that are within 10% of the execution time (marked by the dashed line) of the best found configuration. The figure shows the

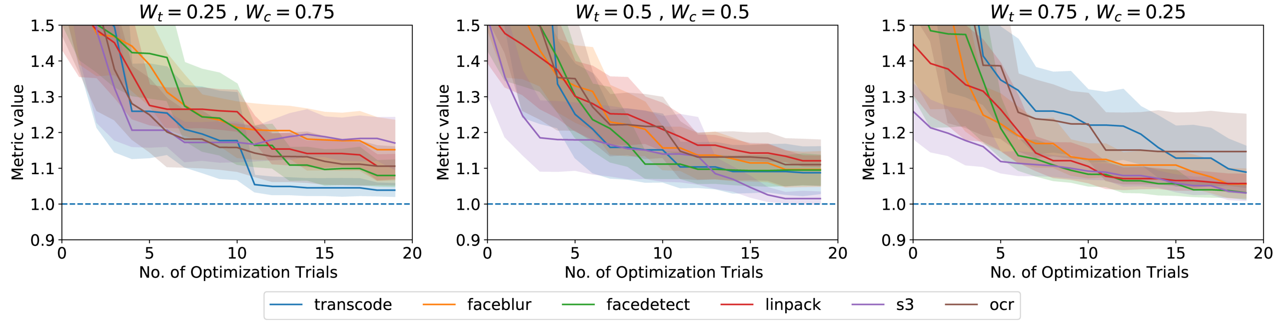

Figure 13. Convergence of the optimization process (BO with GP) for all the benchmarks and different weights for execution time and execution cost.

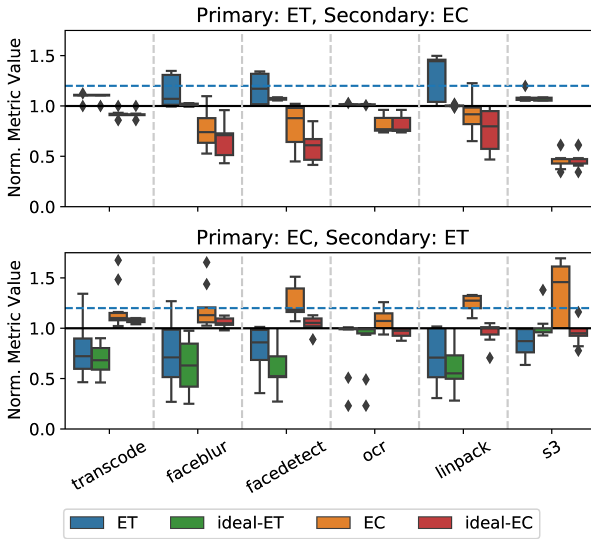

Figure 14. Normalized values for execution time and execution cost after a hierarchical optimization. ET/EC and ideal-ET/ideal-EC represent the best configuration using the prediction model and with oracle-like knowledge, respectively.

normalized value for execution time and execution cost w.r.t. to the best configuration found by the optimization process.

We can see from Figure 15 that, by using the predicted best configurations of other instance types, we can achieve between 25-75% reduction in execution cost, on average, for different benchmarks. This execution cost reduction comes at < 10% increase in execution time, on average. There are outliers where the execution time penalty is up to 50% because of prediction error. The main exception is transcode, for which there are very few execution cost reduction options available (cf. Table 3).

Takeaways: In our experiments, we found that some configurations utilize different instance types but provide performance similar to the best configuration in the search space. A cloud provider can exploit this behavior and use prediction models to achieve lower execution costs (by using idle resources) while providing comparable execution time to the best found configuration. Even with prediction error, we

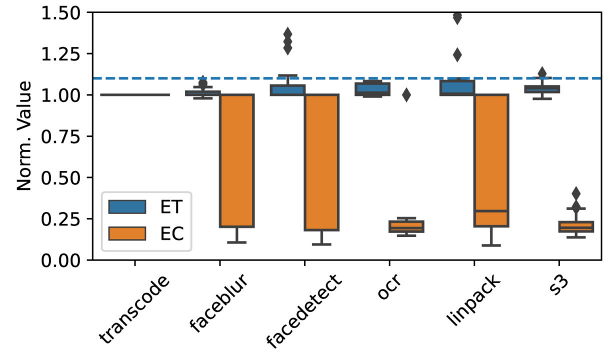

Figure 15. Reduction in cloud provider costs while keeping the objective function value within 10% of the best found configuration (assuming an 80% discount for idle resources).

show that it is possible to significantly reduce costs while delivering performance within 10% (on average) of the execution time of the best found configuration.

7 Design space

Figure 16 shows the design choices that should be considered when developing an automatic resource allocation system for serverless functions. First, the designer has to decide between providing offline or online optimization. If offline optimization is desired, then both search-based (sampling) and model-based optimization are valid choices. But with online optimization, we recommend a model-based approach. We found that Bayesian Optimization with Gaussian Processes performs better than other optimization algorithms that we tested, as it converges to the best configuration faster and it is better at predicting the objective function value for unseen configurations because it builds more accurate models as shown by the lower prediction error in Figure 9 and 10.

After deciding on the optimization algorithm, a designer would need to decide whether to create a data-specific or a generic optimization model. A data-specific optimization process might provide better performance but with added complexity. Irrespectively of whether the model is data-specific or not, one has to decide whether to provide single objective

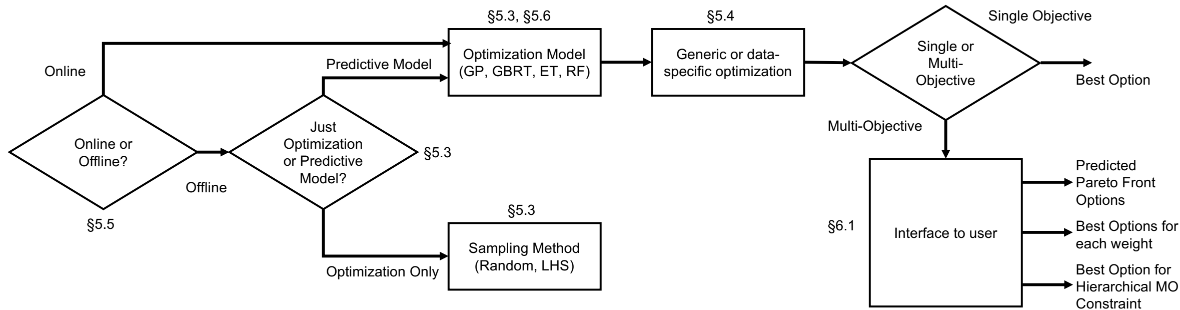

Figure 16. Systematic set of design choices of a system providing automatic resource allocation for serverless functions.

or multi-objective optimization options. A single objective optimization would provide the user with the option of either finding the configuration with the lowest execution time or cost. For multi-objective optimization, in turn, we discussed three options: 1) Pareto front, 2) weighted multi-objective optimization, and 3) hierarchical multi-objective optimization. Hierarchical optimization is the most intuitive for the end-user to reason about, since it allows the user to select a given trade-off the user is willing to accept.

8 Related work

Serverless computing. Recent work demonstrated the advantages of serverless computing for several applications, ranging from data analytics [37, 40], DAG processing [14], video transcoding [23], compilation [22], machine learning [13, 21, 52] and more [8, 48]. A few proposals aimed to optimize the cost of serverless functions [19, 20, 45, 49]. Sizeless [19] and [45] use a regression model to minimize execution cost of AWS lambda functions. [49] uses memory tracing information to collect utilization metrics and find the right memory allocation. COSE [1] uses BO to select a serverless configuration for running a single or a chain of functions, while satisfying customer objectives. Costless [20] uses function fusion, splitting the function between edge and cloud, and allocating memory resources for a sequence of functions to optimize cost. However, these are limited to the resource allocation strategy that AWS Lambda or similar services expose at the moment. In contrast, we take a step back and analyze the space of possible fine-grained configurations, and rethink the interface of services like AWS Lambda.

Several works gain insights into the characteristics of public serverless offerings by creating experiments and benchmarks to test the behavior of these platforms [34, 53, 54, 56]. However, they do not analyze the space of possible configurations beyond the current offerings and their effects.

HarvestVMs [7] create flexible VMs that grow and shrink based on available unallocated resources in an underlying server. Zhang et al. [57] show how to use HarvestVMs for serverless. Our work is complementary since it can inform scheduling functions on the variable resources of HarvestVMs, and enable the use of different instance types, minimizing costs while providing predictable execution times.

Cloud configuration optimization. Cherrypick [2], Arrow [26], Scout [28], Micky [27], Vanir [11] and Lynceus [15] perform cloud configuration optimization for distributed data analytics frameworks. Ernest [51] creates an analytical model for Spark applications and uses that to optimize cloud configurations. PARIS [55] uses historical data and machine learning to quickly choose cloud configurations for tasks that run on single VM instances. Selecta [32] also uses historical data and also it incorporates different storage options into the configuration search space. Our work is not aimed at finding the best optimization algorithm for automatic resource allocation for serverless functions; instead, we deal with design space questions to show the potential opportunities and how they can be utilized.

Google uses Autopilot [43] to configure resources automatically, adjusting both the number of concurrent tasks in a job (horizontal scaling) and the CPU/memory limits for individual tasks (vertical scaling). Autopilot uses ML algorithms applied to historical data about prior executions of a job, plus a set of finely-tuned heuristics, to walk this line. Autopilot changes the amount of CPU and memory but does not provide CPU or machine type recommendations, which is a focus in our work. The black box algorithms discussed in this work could be used as a custom recommendation algorithm in a system like Autopilot.

Resource allocation in data centers. Paragon [17] and its follow-up Quasar [18] propose heterogeneity and interference-aware schedulers for data center workloads. They use collaborative filtering to classify an unknown incoming job to assign resources to it. Similarly, DejaVu [50] also tackles the problem of allocating resources to workloads in a data center, but uses clustering instead of collaborative filtering. Despite tackling a different problem, we note that their use of collaborative filtering and clustering could also replace the black-box optimization methods we have discussed.

9 Discussion

Tail latency. While we focused on median latency in our study, using tail latency as a performance metric should not

change the algorithms or the techniques discussed; however, it could change the best configuration that is found. In addition, it might require more measurements and could potentially increase the time it takes to find a good configuration. More importantly, cold starts, queuing and interference can be major sources of increased tail latency in serverless functions (assuming a scalable FaaS implementation). These issues should be addressed by improving scheduling decisions among other options, more so than by our resource selection. We leave it as future work to devise ways to inform scheduling decisions using the models we propose.

Interference and resource contention. Our central concern is resource allocation for serverless functions. Interference and resource contention can influence this decision. A resource configuration deemed to be the best in isolation might not provide the same performance depending on the interference and noise from other applications sharing the same underlying server. We consider this to be a relevant but complementary issue, and an interesting avenue for future work. We envision that it can be dealt with via interference-aware scheduling techniques such as CloudScope [16], in conjunction with resource allocation decisions.

On cold-start latency. Cold start latency will impact performance of any selected configuration in a serverless environment. Cold-start latency can be measured against the warm-start latency for the same application. This information can then be used by the cloud provider to decide whether to start a new serverless instance for a serverless function with the best configuration or use a sub-optimal but already warm serverless instance, for an incoming request to the serverless application.

Impact on scheduling. The implications of decoupled and flexible resource allocation on scheduling are important and handling it can be challenging. However, cloud providers already offer semi-flexible decoupling of memory and CPU for services like AWS Elastic Container Service with Fargate [3]. In this paper, we similarly use a coarse-grained resource allocation rather than arbitrary value for CPU and memory. Thus, we believe that it is a challenge that cloud providers are equipped to handle.

On concurrent serverless instances. FaaS providers allow users to specify the provisioned concurrency level for their serverless applications. The concurrency setting can improve the latency (by decreasing queuing time and cold-start latency at the expense of cost). But the lower bound on the latency of a single request to the serverless function depends on the resource allocation to each serverless function instance, which we address in this work.

Our study focuses on resource allocation for a single function invocation, and therefore we consider the aspect of concurrent invocations to be an interesting avenue of future work. As a first approximation, each concurrent function instance can be assigned the best configuration found by the optimizer (for that function). Once the prediction models are built, the optimizer can also suggest the top k configurations. The FaaS provider can then make a decision for concurrent function invocations based on several factors, including resources available, performance constraints and cold-start latency.

Limitations. While we employed a diverse set of applications, we note that other applications might have a behavior that changes with their inputs, thus requiring a data-dependent approach. Similarly, BO with GP will not be a good optimization algorithm if the underlying performance model is not smooth, and other optimization algorithms might perform better. Finally, if an analytical model is known for a function, e.g., if a function is embarrassingly parallel, then a mathematical model is better than black-box methods.

We assumed that actual workload inputs to the functions would have similar characteristics to the inputs used to optimize the serverless application. A drift in the predicted and observed performance characteristics over time can be one of the indicators used to retrigger optimization or improvement of the optimization algorithm automatically.

Lastly, we created our own pricing model based on the cost of different EC2 instances. However, if a FaaS provider is to offer a decoupled allocation service, it is likely going to use a different pricing scheme. Cloud providers usually charge a premium for managed services like serverless. The black-box algorithms used in this paper should be equally applicable when used with a different pricing model. Quantitatively, our results in absolute terms will change with a different pricing model. Qualitatively, as long as the ratio of prices across different instance types and between CPU and memory remain the same, our results should also hold true.

10 Conclusion

We highlighted the importance of carefully reasoning about each resource underlying the execution of serverless functions. We thoroughly studied the gains that can be obtained, and propose possible interfaces that allow users to only worry about the essential tradeoffs. The fact that AWS lambda now also offers ARM-based serverless functions (increasing the resource configuration options) further highlights that cloud providers are moving in the direction where automatic resource allocation might become necessary.

Acknowledgments

We thank our shepherds, Redha Gouicem and John Wilkes, and the anonymous reviewers for their feedback. M. Bilal was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (FPA 2012-0030). This work was supported by Fundação para a Ciência e a Tecnologia, under grants UIDB/50021/2020, PTDC/CCIINF/6762/2020.

References

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *IEEE Conference on Computer Communications (INFOCOM '20)*. 129–138.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. 469–482.

[3] Amazon. 2022. *AWS Fargate Pricing Page.* https://aws.amazon.com/fargate/pricing/

[4] Amazon. 2022. *AWS Lambda.* https://aws.amazon.com/lambda/

[5] Amazon. 2022. *AWS Lambda Functions Powered by AWS Graviton2 Processor.*

[6] Amazon. 2022. *AWS Lambda Pricing Page.* https://aws.amazon.com/lambda/pricing/

[7] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 735–751.

[8] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. 263–274.

[9] Jasbir Singh Arora. 2012. *Introduction to Optimum Design (Third Edition).* Elsevier. 657–679 pages.

[10] Muhammad Bilal. 2022. *This Paper's Artifacts Repository.* https://github.com/MBtech/rethinking-serverless

[11] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. 2020. Finding the Right Cloud Configuration for Analytics Clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 208–222.

[12] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. 2020. Do the Best Cloud Configurations Grow on Trees? An Experimental Evaluation of Black Box Algorithms for Optimizing Cloud Workloads. *PVLDB* 13, 12 (2020), 2563–2575.

[13] João Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SoCC '19)*. 13–24.

[14] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 1–15.

[15] Maria Casimiro, Diego Didona, Paolo Romano, Luís Rodrigues, Willy Zwaenepoel, and David Garlan. 2020. Lynceus: Cost-efficient Tuning and Provisioning of Data Analytic Jobs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS '20)*. 56–66.

[16] Xi Chen, Lukas Rupprecht, Rasha Osman, Peter Pietzuch, Felipe Franciosi, and William Knottenbelt. 2015. CloudScope: Diagnosing and Managing Performance Interference in Multi-tenant Clouds. In *IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '15)*. 164–173.

[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 77–88.

[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 127–144.

[19] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the Optimal Size of Serverless Functions. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. 248–259.

[20] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC '18)*. 300–312.

[21] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD '18)*. 334–341.

[22] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. 475–488.

[23] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-latency Video Processing using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. 363–376.

[24] Google. 2022. *GCP Cloud Functions Pricing Page.* https://cloud.google.com/functions/pricing

[25] Google. 2022. *Google Cloud Functions.* https://cloud.google.com/functions/

[26] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. 2018. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS '18)*. 660–670.

[27] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent Freeh. 2018. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD '18)*. 409–416.

[28] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. 2018. Scout: An Experienced Guide to Find the Best Cloud Configuration. *arXiv 1803.01296* (2018).

[29] K3S. 2022. K3S Website. https://k3s.io.

[30] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*.

[31] Jeongchul Kim and Kyungyong Lee. 2022. FunctionBench. https://github.com/kmu-bigdata/serverless-faas-workbench.

[32] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. 759–773.

[33] Kunal Mahajan, Daniel Figueiredo, Vishal Misra, and Dan Rubenstein. 2019. Optimal Pricing for Serverless Computing. In *2019 IEEE Global Communications Conference (GLOBECOM '19)*. 1–6.

[34] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: A Benchmark Suite for Serverless Computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems (DEBS '20)*. 73–84.

[35] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 21, 2 (1979).

[36] Microsoft. 2022. *Azure Functions.* https://azure.microsoft.com/en-us/services/functions/

[37] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 115–130.

[38] OpenFaaS. 2022. OpenFaaS Function Store. https://github.com/openfaas/store.

[39] OpenFaaS. 2022. OpenFaaS Wesite. https://www.openfaas.com.

[40] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. 193–206.

[41] pyDOE. 2022. pyDOE Website. https://pythonhosted.org/pyDOE/.

[42] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. 2020. Analyzing System Performance with Probabilistic Performance Annotations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[43] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[44] Sckit. 2022. Scikit Optimize Website. https://scikit-optimize.github.io/stable/.

[45] Özgür Sedefoğlu and Hasan Sözer. 2021. Cost Minimization for Deploying Serverless Functions. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*. 83–85.

[46] Endre Simo. 2022. Go port of Mario Klingemann's Stackblur algorithm. https://github.com/esimov/stackblur-go.

[47] Endre Simo. 2022. *Pigo: Pure Go Face Detection Library*. https://github.com/esimov/pigo

[48] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. SNF: Serverless Network Functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 296–310.

[49] Josef Spillner. 2020. Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing (WoSC '20)*. 13–18.

[50] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 423–436.

[51] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics.. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. 363–378.

[52] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE Conference on Computer Communications (INFOCOM '19)*. 1288–1296.

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. 133–146.

[54] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Yun Ma, Haoyu Wang, and Xuanzhe Liu. 2020. Understanding Characteristics of Commodity Serverless Computing Platforms. arXiv:2012.00992 [cs.SE]

[55] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. 452–465.

[56] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 30–44.

[57] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 724–739.

A Artifact Appendix

A.1 Abstract

This appendix provides the details of the artifacts available for the research paper. The artifacts include the serverless applications, input data for the applications, the optimization algorithms, analysis notebooks, and the performance data collected for the experimental scenarios described in the paper.

A.2 Description & Requirements

A.2.1 How to access. The public repository containing data and code for the research paper can be found at: https://doi.org/10.5281/zenodo.7143413.

A.2.2 Hardware dependencies. When running the optimization code/notebooks, a multi-core machine is ideal but otherwise, there are no specific hardware requirements.

A.2.3 Software dependencies. A Linux/Mac machine with Python >=3.8 and Jupyter Notebooks (the code has been executed on Mac OS) is required for running python code that is in the repository. Any other specific software requirements (if any) for running specific parts of the code are mentioned in the README files of the sub-directories in the repository.

A.2.4 Benchmarks. Benchmarks/Serverless applications used in the paper are available in the repository in the *benchmarks* sub-directory. These benchmarks are compatible with OpenFaaS.

A.3 Set-up

Using different parts of the repository requires different installation setups. But all of the configuration setups can be done locally (in some cases using python virtualenv). The instructions in the sub-directories of the repository contain the requirements as well (and instructions on how to install them).

A.4 Evaluation workflow

A.4.1 Major Claims. *The major claims of the research work include:*

- *(C1): We determine the ground truth about the execution time and execution cost of 6 serverless applications across 288 resource configurations and multiple inputs. Our benchmarks and data are available as open source.*
- *(C2): We analyze the potential benefits of enabling a more flexible resource allocation for serverless functions*
- *(C3): We analyze the accuracy of 4 Bayesian Optimization algorithms for determining the best resource allocation for two optimization objectives: execution time and cost.*
- *(C4): We verify whether the serverless functions of our study have data-dependent performance characteristics*

- *(C5): We propose a set of possible interfaces for enabling the user to benefit from multi-objective optimization.*
- *(C6): We evaluate the cost reduction opportunities for the cloud providers by using different instance types while providing predictable performance.*

A.4.2 Experiments.

Instructions are provided in the `analysis/README.md` on the setup required to run the notebooks needed for some of the experiments below.

Experiment (E1): [Benchmark Data] [30 human-minutes]: The benchmark data is available under `benchmarks/` subdirectory of the repo with instructions on the naming of the files and metrics that have been used in our paper. This relates to C1.

Experiment (E2): [30 human-minutes + 15 compute-minutes]: For C2, we have two notebooks in the repository to show the benefits of flexible resource allocation strategies.

1. `analysis/decoupled-vs-default.ipynb`
2. `analysis/alternative-analysis.ipynb`

[Results] The notebooks have previously created graphs and outputs that have been used in the paper. Running decoupled-vs-default creates a plot that is Figure 3 in the research paper. Running alternative-analysis provides the alternative configurations that were used in Table 2.

Experiment (E3): [1 human-hour + 2 compute-hour]: For C4, we have the notebook `analysis/optimization-analysis-notebook.ipynb` that runs four Bayesian optimization algorithms that we have discussed in the paper, for execution cost and execution time objective functions.

[Results] The notebooks have the code to create Figures 5-6 and Figures 8-10 of the paper.

Experiment (E4): [30 human-minutes + 1 compute-hour]: For C4, we have the notebook `analysis/optimization-across-data.ipynb` that compared the two scenarios:

1. When a generic performance model is used to predict performance for input data it has not seen before.
2. When the performance model is trained for that specific input.

[Results] The notebooks have the code to create Figure 7 of the paper.

Experiment (E5): [1 human-hour + 3 compute-hour]: For C5, we have three notebooks in the repository that use different optimization techniques to provide possible interfaces for the user as discussed in Section 6.1 of the research paper.

1. `analysis/pareto-front.ipynb`
2. `analysis/multi-objective.ipynb`
3. `analysis/hierarchical-mo.ipynb`

[Results] The notebooks have the code to create Figures 12-14 of the paper.

Experiment (E6): [30 human-minutes + 1 compute-hour]: For C6, we have `analysis/cost-benefit-alternatives.ipynb` in the repository to show cost benefit of using alternative instance type configurations that are within the performance threshold of the best configuration.

[Results] The notebooks have the code to create Figure 15 of the paper.

A.5 Notes on Re-usability

Several components of the repository are standalone in terms of usability. The performance data available (under `data/`) in the repository can be easily used in other research works on optimization without using any other part of the repository. Additionally, only changes to the input parameters and settings (usually in the first section of the notebook) are required to use them for analysis of other benchmarks and additional input data.