Parrotfish: Parametric Regression for Optimizing Serverless Functions

Arshia Moghimi
University of British Columbia
Vancouver, Canada

Joe Hattori
University of Tokyo
Tokyo, Japan

Alexander Li
University of British Columbia
Vancouver, Canada

Mehdi Ben Chikha
INSAT
Tunis, Tunisia

Mohammad Shahrad
University of British Columbia
Vancouver, Canada

ABSTRACT

Serverless computing is a new paradigm that aims to remove the burdens of cloud management from developers. Yet rightsizing serverless functions, primarily through setting memory limits, remains a pain point for developers. Choosing the right memory configuration is necessary to ensure cost and/or performance optimality for serverless workloads. In this work, we identify that using parametric regression can significantly simplify function rightsizing compared to black-box optimization techniques currently available. With this insight, we build a tool, called Parrotfish, which finds optimal configurations through an online learning process. It also allows users to communicate constraints on execution time, or to relax cost optimality to gain performance. Parrotfish achieves substantially lower exploration costs (1.81-9.96×) compared with the state-of-the-art tools, while delivering similar or better recommendations.

ACM Reference Format:
Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. 2023. Parrotfish: Parametric Regression for Optimizing Serverless Functions. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624654

1 INTRODUCTION

Serverless computing has gained significant traction in the past few years. It removes a considerable portion of the provisioning burden from the shoulders of developers, offers a pay-per-use pricing model, and accelerates the deployment of scalable cloud-based applications. Today, all major cloud providers have serverless offerings in a variety of forms, such as Function-as-a-Service (e.g., AWS Lambda, Azure Functions, and Google Cloud Functions), Container-as-a-Service (e.g., AWS Fargate and IBM Code Engine), and other domain-specific services (e.g., Google BigQuery). According to Datadog's June 2022 analysis of cloud user telemetry [19], over 70% of organizations using AWS and over 50% of Azure and Google Cloud users have adopted serverless offerings.

Over the past few years, the research community has proposed intricate solutions to major limitations of serverless systems. Yet the issue of rightsizing, which every serverless developer deals with on a day-to-day basis, has received relatively low attention. It involves setting the correct configuration for each building block (e.g., function or container) of a serverless application, and is an artifact of serverless's pay-per-use pricing model. To maximize cost and performance efficiency, developers need to rightsize their serverless functions when they modify the source code, update libraries, experience a different request mix from end-users, or when the provider modifies the pricing parameters. These necessitate frequent, prompt, and cost-efficient rightsizing.

Currently, developers are equipped with rudimentary and inefficient tools for function rightsizing. Many have to resort to manually testing their functions with different sizes and various inputs to find the most cost- or performance-optimal configuration. Some rely on automated configuration sweeping tools [14], which as we show are costly and do not scale. The rightsizing burden goes against the serverless philosophy that aims to simplify provisioning aspects for developers. Rightsizing functions can lower the cost dramatically [57], so there is real value in building low-cost, high-quality, and automated function rightsizing tools.

In this work, we uncover a new approach to configuring serverless functions. We identify that there exists an intrinsic relation between execution time and resources allocated to serverless functions. By building models relating the two

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624654

and leveraging them for parametric regression, we show that function rightsizing can be done with fewer samples compared with naive configuration sweeping or black-box ML-based techniques. Furthermore, we show how the sampling strategy can also be made aware of the sampling cost. We combine these ideas in a new serverless rightsizing tool, called Parrotfish (PARametric Regression for OpTimizing Functions In Serverless). By building an accurate cost model using parametric regression and with its cost-aware sampling strategy, Parrotfish can reduce the configuration exploration cost by 1.81-9.96× compared to three state-of-the-art optimization tools. This enables more frequent optimizations, compounding long-term savings. Developers can use Parrotfish in their nightly builds and CI/CD pipelines. Similarly, providers can use it to rightsize functions at scale.

The execution time and cost models that Parrotfish builds with a limited number of samples are used for function rightsizing in this work. The application of this approach is much broader, especially when resource management relies on execution time predictions. Schedulers and load balancers can use the accurate, per-function execution time models when controlling tail latency [25], leveraging heterogeneity [41], or opportunistically harvesting idle resources [42, 59]. Similarly, adaptive lifetime management policies [41, 50] can leverage these accurate models to fine-tune keep-alive parameters.

Parrotfish is publicly available at https://github.com/ubc-cirrus-lab/parrotfish.

2 BACKGROUND AND MOTIVATION

2.1 Serverless Pricing Model

One of the most appealing attributes of serverless for developers is its pay-per-use pricing model. Developers are charged for the resources consumed during the execution of their functions (capacity cost) as well as a fixed fee for each function invocation (request cost). It is also common for serverless providers to offer monthly free tiers, for instance, the first 400,000 GB-s and the first one million requests per month for AWS Lambda.

Inspired by a similar study [11], Table 1 compares the base pricing model for three popular serverless offerings: AWS Lambda [5], Azure Functions [10], and IBM Cloud Functions [30]. IBM Cloud Functions charges only for capacity cost. The capacity cost for these offerings is charged per GB-s. The monthly capacity usage (GB-s) is calculated as the sum of capacity usage for function executions in a subscription, where the usage for each execution is determined as the multiplication of memory (GB) in execution duration (s). The minimum billed duration, duration rounding resolution, or even static vs. dynamic metering of memory differs across providers, as shown in the table. Moreover, there can be discounts to incentivize using alternative architectures

	Cost						
	Request Cost		Capacity Cost				
Provider	Free Tier	Cost per Million	Free Tier	Cost per GB-s	Min Billed Duration	Round-up Resol.	Memory Metering
AWS Lambda [5]	1 M	$0.2	4e5 GB-s	$1.67e-5	1 ms	1 ms	Static
Azure Functions [10]	1 M	$0.2	4e5 GB-s	$1.6e-5	100 ms	1 ms	Dynamic
IBM Cloud Functions [30]	∞	$0	4e5 GB-s	$1.78e-5	100 ms	100 ms	Static

Table 1: The base pricing model for three popular public serverless offerings as of June 4, 2023.

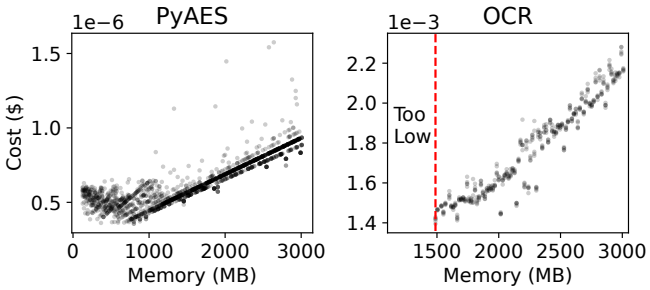

Figure 1: High variability of execution times in serverless hinders modeling the cost function with limited samples. The unknown minimum memory requirements pose an additional challenge.

or for high volume customers [5, 10]. However, at its core, charging for the resource-time is the common practice.

For developers, using Parrotfish makes sense if the total monthly usage is expected to surpass the provider's free tier. Providers, however, can enhance resource efficiency by rightsizing serverless functions, irrespective of usage tier. Lastly, to optimize costs, the key focus is on modeling the average cost per invocation (total cost divided by request count), rather than predicting tail behavior.

2.2 Why is serverless rightsizing hard?

Finding optimal configurations for serverless functions is not a straightforward task due to a few practical challenges:

(1) *High execution time variability:* Serverless execution times are short. Datadog has reported [18] that the median AWS Lambda execution time dropped to 60 ms in 2020, from 130 ms in 2019. At such short execution times, performance variations due to shared resources (memory bandwidth [39], last-level cache [49], etc.) have an amplified impact on relative jitter. Diurnal load variations of serverless providers have been shown to have a quantifiable impact on the performance of serverless applications [26, 45]. Needless to say, the degree of performance variation depends on the application [26, 34], invocation pattern [44], and the host architecture [34]. This performance variation translates into measurement uncertainty for any configuration optimizer. Figure 1 shows the cost measurements for two benchmark applications.

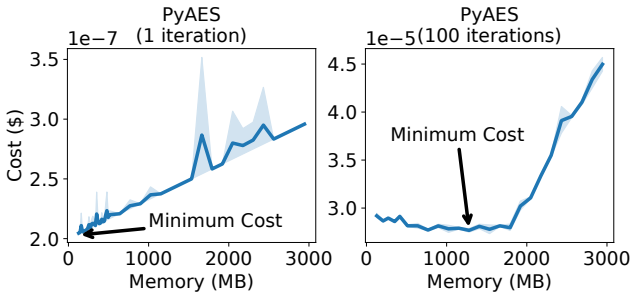

Figure 2: The cost-optimal configuration for the same function can vary based on input.

PyAES [32] is a Python function designed for AES block cipher calculations, while OCR [56] is a Python function used to extract text from a specified PDF file. The degree of measurement uncertainty is typically exacerbated for functions with shorter execution times (PyAES here). Note that cold start executions are excluded here, and the observed performance jitter is just for warm executions. Cold starts introduce even higher variations [24, 50, 54].

(2) *Minimum memory requirements unknown a priori:* Function execution can fail if sufficient memory is not provided [9]. The minimum memory requirement is not known a priori to the configuration optimizer or even to the developer, is function-specific, and can vary with different inputs to the same function [36]. The OCR function shown in Figure 1, for instance, needs at least ~1.5 GB of memory to run for the specified input.

(3) *Input dependence:* The execution time and consequently optimal configuration of functions can be heavily input dependent. Figure 2 depicts the cost measurements for the PyAES benchmark using two distinct inputs, resulting in varying numbers of encryption iterations. The cost-optimal configuration varies depending on the computational requirements of the input. This is well documented by prior work [13, 23, 28, 36].

2.3 State of Serverless Cost Optimization

Choosing a suitable configuration for serverless functions is necessary to ensure performance and cost optimality. Currently, there are three approaches used for rightsizing serverless function configurations:

(1) **Manual Search:** Developers can manually search for memory configurations suitable for their functions. While crude, this approach is used by some today and is guaranteed to result in cost savings [12, 33]. The main drawback of this approach is reduced developer productivity [17]. Manual reconfiguration hinders adaptability to function changes, fast-evolving workloads, and diverse inputs.

(2) **Automated Exhaustive Search:** Here, several configurations are tested, and the best is chosen based on the optimization goal (e.g., lowest cost). Tools such as AWS Lambda Power Tuning [14] fall in this category. This approach eliminates the need for developers to do the configuration sweep themselves. However, in the face of high performance variability in serverless functions [24, 26], high quality results require an exhaustive configuration sweep, leading to slow and costly explorations.

(3) **Learning-Based Search:** In this category, the goal is to estimate the performance model of serverless functions without prior knowledge. A number of strategies have been studied in this space, including using Bayesian optimization [1, 60] to estimate the cost and execution time of serverless functions, and training a machine-learning model on thousands of serverless functions to build a performance model [22]. These methods offer higher efficiency (recommendation quality per exploration cost) compared to exhaustive search approaches. However, due to their reliance on black-box performance models, they still need a substantial number of samples to construct an accurate model, particularly in the presence of significant execution time variations. In §5.1, we demonstrate how incorporating knowledge of the underlying model, coupled with a cost-aware sampling strategy, can effectively minimize exploration costs without compromising the quality of results.

3 MODELING SERVERLESS FUNCTIONS

Memory serves as the main configuration parameter for most serverless offerings. The allocation of other resources such as CPU and I/O shares depend on the memory configuration. This is unlike the traditional virtual machine (VM) and container configurations, where various resource ratios are available. AWS Lambda, for example, allocates CPU in proportion to the amount of memory in such a way that at 1,769 MB, a function has the equivalent of one vCPU [6]. As a result, the performance of a serverless application depends heavily on the choice of memory value, even though the application might not be memory-intensive itself. Even after the function receives all its required resources, increasing the memory size leads to fewer co-tenants, reducing the noisy neighbor effects and slightly improving performance.

Prior work treats serverless functions' behavior as black-box [1, 22, 60]. We leverage the one-dimensional configuration space of serverless functions, modeling performance and cost as a function of memory. As we show later in the paper, such a model can simplify the configuration exploration.

3.1 Modeling Execution Time and Cost

The cost of running a serverless function consists of constant request cost, which is independent of the function's execution, and capacity cost (§2.1). The capacity cost depends on

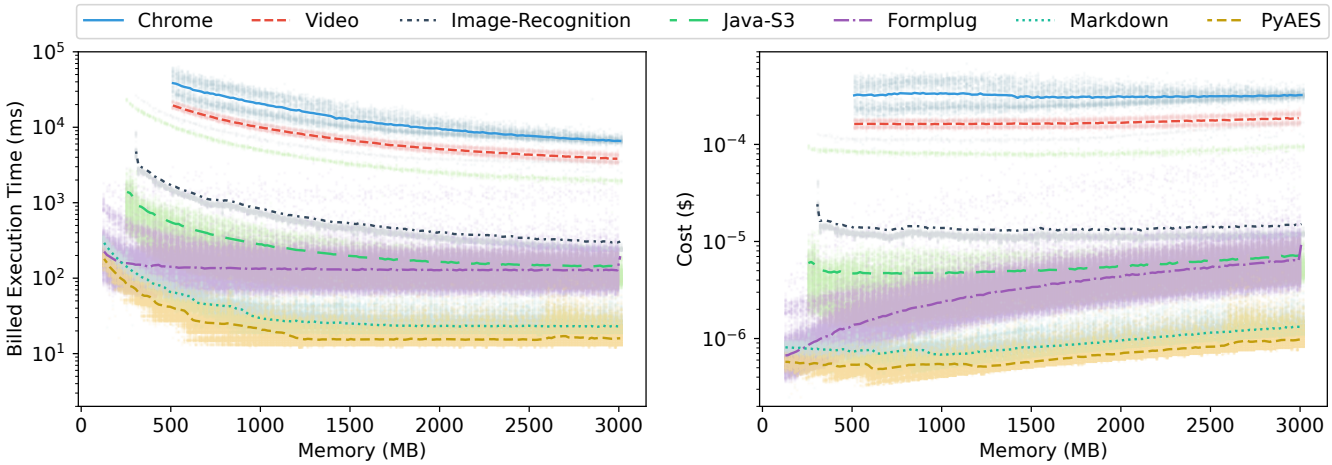

Figure 3: Execution time and cost of different serverless applications. Each line indicates the average of the data for that benchmark. (Execution time and cost are in log scale.)

the amount of configured memory and the execution duration. The execution time of a function itself depends on the memory size (m), which determines a number of other resources, as mentioned earlier:

$$Cost(m) \propto m \times ExecTime(m) \quad (1)$$

Modeling either the cost function or the execution time for a serverless function, we can reason about rightsizing it. In this section, we investigate whether either can be accurately and reliably modeled by a family of mathematical functions. For this, we gathered an extensive number of execution logs from seven different serverless functions on AWS Lambda:

(1) Chrome-Screenshot [3]: Written in JavaScript, it takes the screenshot of a headless Chrome browser for a given URL and saves it as a file with given display size.
(2) Formplug [31]: An HTML form forwarding service written in JavaScript.
(3) Image-Recognition [15]: A Python ML inference application classifying an input image uploaded to a bucket.
(4) Java-S3 [8]: A Java application that reads an image from an S3 bucket, shrinks it, and saves it in the bucket.
(5) Markdown-to-HTML [48]: A Python script that converts Markdown to HTML.
(6) PyAES [32]: AES block cipher calculation in Python.
(7) Video-Processing [15]: A Python script that adds a watermark to a given video and converts it to a GIF.

We chose benchmarks written in Python, Node.js, and Java as they are the most widely used languages in AWS Lambda, accounting for nearly 90% of functions as of 2021 [18]. These benchmarks have various characteristics that enable us to evaluate our ideas in a number of different scenarios. For instance, Formplug is a lightweight benchmark that requires minimal computations. It performs sufficiently fast even with the minimum allocated memory (128 MB). On the contrary, Chrome-Screenshot and Video-Processing benchmarks are computationally intensive, exploit parallelism, and have longer execution durations. They require a minimum of 512 MB of memory to successfully execute. Furthermore, these two benchmarks, alongside Formplug, make external network calls during execution. These include opening a webpage, downloading files from external storage, and sending emails, respectively. PyAES and Markdown-to-HTML benchmarks require moderate computing power and do not involve any external communication.

We incrementally configured each benchmark's memory and collected the billed execution time and cost for each invocation using a fixed input. We discuss how it is possible to handle variable inputs in §5.2. To account for the variations in the cloud, we sampled each memory configuration more than 100 times. We collected the execution time and cost data multiple times a day over the span of two weeks to compensate for diurnal and hourly variations [45]. Figure 3 shows the result of this experiment. As expected, due to the resource allocation scheme of serverless functions, increasing memory decreases the execution time for all the functions. The degree of decrease, however, depends on the function's characteristics. The cost of running the function, on the other hand, does not follow any specific trends.

We used the OriginPro [38] data analysis tool to check the fitness of more than two hundred built-in mathematical functions against data collected for execution time and cost. For simplicity, we categorized different functions of the same family under the same name, for example, quadratic and cubic functions are categorized under the *Polynomial* family, and exponential functions with different degrees all fall under the *Exponential* family. To enhance result differentiation,

	Chrome-Screenshot			Formplug			Image-Recognition			Java-S3			Markdown-to-HTML			PyAES			Video-Processing		
Fit	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE
Best Fit	Poly	0	1.13	Exp	0	1.02	Exp	0	7.69	Exp	0	1.27	Exp	0	1.95	Exp	0	3.90	Poly	0	1.56
2nd Best	Exp	44	1.29	Poly	128	2.18	Log	532	15.52	Log	280	2.94	Poly	142	17.48	Poly	119	17.80	Exp	36	0.41
3rd Best	Asymp	158	1.63	Asymp	289	2.03	Poly	567	10.67	Asymp	518	10.07	Asymp	488	7.30	Asymp	312	7.87	Recip	556	2.74
4th Best	Recip	276	2.48	Log	349	1.22	Asymp	705	15.18	Poly	611	11.72	Recip	565	18.86	Recip	484	21.67	Asymp	605	2.57
5th Best	Log	476	1.04	Recip	656	4.14	Recip	727	10.36	Recip	491	13.13	Log	918	37.74	Log	764	37.33	Log	789	5.07

Table 2: Top five families of functions for modeling *billed execution time* of benchmark serverless functions on AWS Lambda. We use the MAPE (lower is better) and *BIC* (lower is better) metrics to evaluate model fitness. (Poly: Polynomial, Exp: Exponential, Asymp: Asymptotic, Recip: Reciprocal, Log: Logarithmic)

	Chrome-Screenshot			Formplug			Image-Recognition			Java-S3			Markdown-to-HTML			PyAES			Video-Processing		
	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE	Family	ΔBIC	MAPE
Best Fit	Poly	0	0.88	Linear	0	1.05	Exp	0	3.77	Exp	0	1.08	Poly	0	1.55	Poly	0	2.01	Poly	0	0.33
2nd Best	Sine	34	1.26	Exp	1	1.04	Rational	17	3.18	Sine	229	1.51	Sine	12	1.52	Sine	36	2.01	Exp	37	0.38
3rd Best	Exp	124	1.65	Poly	5	1.08	Poly	45	2.44	Poly	27	1.52	Exp	277	1.74	Exp	53	2.10	Sine	38	0.38
4th Best	Asymp	258	2.04	Log	1076	2.09	Asymp	60	3.40	Recip	469	3.68	Recip	429	5.42	Recip	346	4.65	Recip	407	1.37
5th Best	Log	271	1.14	Recip	1298	74.59	Log	70	2.21	Linear	538	5.03	Linear	568	8.31	Linear	515	8.06	Linear	434	1.48

Table 3: Top five families of functions for modeling *cost* of benchmark serverless functions on AWS Lambda. Like before, we use the MAPE (lower is better) and *BIC* (lower is better) metrics to evaluate model fitness. (Poly: Polynomial, Exp: Exponential, Asymp: Asymptotic, Recip: Reciprocal, Log: Logarithmic)

we excluded linear functions from the polynomial family categorization. Tables 2 and 3 show the five best-fitting families of functions for each benchmark's execution time and cost. We used two metrics to identify the families of functions that best fit our data. The first metric is Mean Absolute Percentage Error (MAPE). MAPE is a measure of the accuracy of a forecast model [20] and is defined as:

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} |\frac{A_i - F_i}{A_i}|, \tag{2}$$

where A_i and F_i are the actual and forecast values, respectively, and n is the number of fitted points. Lower MAPE value depicts a better forecast.

We also use the Bayesian Information Criterion [46] metric to incorporate both goodness of fit and model complexity. BIC values in themselves are not interpretable and are only for comparing different models fitted to the same data. Hence, we compare *BIC* values in the table as

$$\Delta BIC_i = BIC_i - BIC_{min}, \tag{3}$$

with BIC_{min} being the minimum BIC in each column. ΔBIC depicts the difference in the goodness of fit between different models. The larger the ΔBIC value in the table, the less plausible its corresponding model is at being the best approximating model. The results shown in Tables 2 and 3 show high accuracy for the models and demonstrate the feasibility of modeling the execution time and cost.

3.2 Sample-Limited Model Performance

In §3.1, we used millions of data points to search for the underlying models of execution time and cost. We were interested in determining the underlying average models. In reality, however, collecting that much data is not feasible when rightsizing every function due to substantial cost overhead. More importantly, rightsizing may be needed frequently with any code change, library update, or workload change. Collecting 100 samples for thousands of memory configurations will be both costly and time-consuming. The question we pose here is as follows: *Can the parameters for the models identified in the previous subsection be robustly trained with a limited number of samples?*

To evaluate the resilience of different fitting functions when using limited samples, we select five random samples from each benchmark's data and use only those to train the fitting functions that exhibited the highest accuracy in §3.1. We then compare these poorly fitted functions against the original full dataset and measure the MAPE values. To ensure statistical significance, we repeat this process 100 times for each benchmark to consider different random scenarios.

Figure 4 shows the cumulative distribution function (CDF) of MAPE values for each benchmark. For modeling execution time, the exponential family of functions provided the best fit to the full dataset (Table 2) and continues to deliver superior performance with partial data here, too. On the contrary, for modeling cost, the polynomial family of functions provided the best fit to the full dataset (Table 3), but exhibits significantly inferior relative fit performance compared to other models. This discrepancy is in part due to the different nature of the cost and execution time functions. As depicted in Figure 3, the average execution time is monotonically decreasing, while the cost function has varying patterns.

As described in §3.1, modeling either of execution time or cost would be sufficient, as they are dependent functions. Given the consistent superiority of estimating the execution time with exponential functions, whether with full or partial data, we will use the execution time model in the subsequent sections. The cost can be predicted using the execution time model, with the pricing details provided in §2.1.

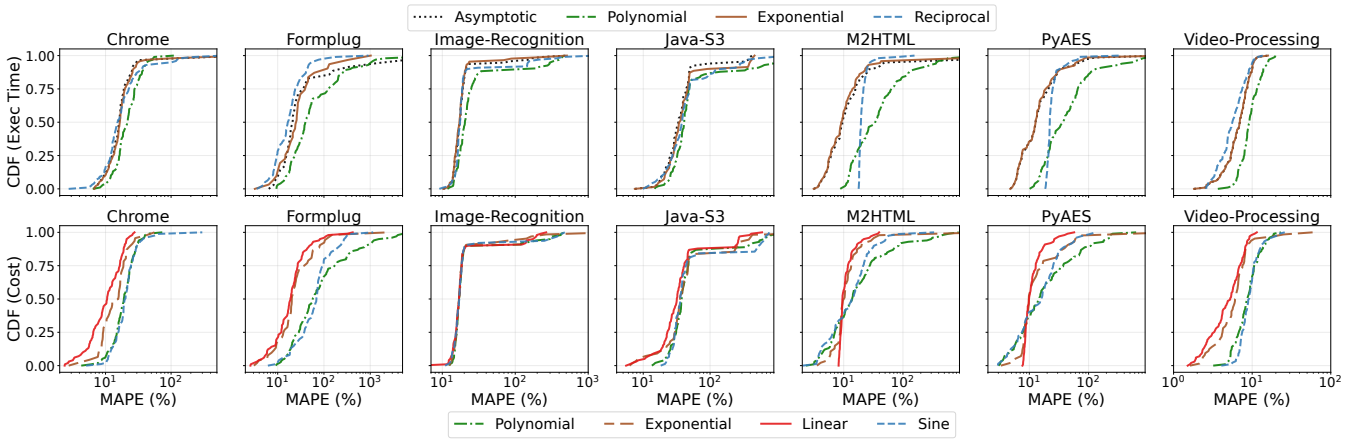

Figure 4: Distribution of MAPE values for the best fitting functions given only 5 data points.

Figure 4 also reveals another insight: compute-intensive functions such as Video-Processing and Image-Recognition exhibit a high level of accuracy in the resulting models. On the other hand, for short-lived functions like PyAES and functions that rely on third-party APIs such as Formplug, the data shows considerable variation, and all fitted functions perform below expectations. This highlights the limitation of relying solely on random sampling when building a reliable model and motivates the need for a better sampling strategy.

Some recent studies have assumed suitability of exponential functions to model the relation between allocated memory and execution time of serverless functions [57], or observed it with limited data and without comparison to other families of functions [1]. Researchers have reported the same relation between resource allocation and execution time for serverless queries of SCOPE big data workloads at Microsoft [40]. In our search for the best performance and cost models, we took a data-driven approach, using millions of execution logs and evaluating the fitness of more than two hundred families of functions. To the best of our knowledge, it is the first work presenting a characterization-driven modeling of serverless functions at this scale. Our results confirm observations of these prior studies and provide more concrete evidence to the community. In §5.7 we will show that this approach can also be used to model the tail execution time of serverless functions.

Intuitively, as resource allocation is bound to the choice of memory, when the memory size of a function is decreased, there is an increase in the function's execution time. Conversely, increasing the memory allocation provides the function with more resources, which can potentially reduce the execution time. However, there is a point where the function cannot effectively utilize the excess resources, leading to a plateau in the execution time. For example, the execution time of a single-threaded compute-bound function would have little improvement for memory allocation beyond 1,769 MB of memory, which maps to 1vCPU in AWS Lambda. Exponential functions are well-suited to estimate such behavior with few model parameters.

Finally, we do not incorporate the input of each function in our model, as prior work has shown that the correlation between input size/type and resource usage is highly function-dependent and may not exist [36].

3.3 Significance of Sampling Strategy

We used random sampling in §3.2 and observed moderate results for many benchmarks. We now investigate the potential for improving model accuracy under limited samples by using an active sampling technique. In particular, an alternative approach to sampling is binary search. In binary search we select search space ends, midpoint, divide the space, and repeat until sufficient samples are gathered. For example, for the default memory range for AWS Lambda, which spans from 128 MB to 3,008 MB[1], the binary search sampling strategy would sample the configurations in the following order: 128, 3008, 1568, 848, and 2288. This method provides a systematic and efficient way to explore the search space. Note that the order of samples does not affect the fit accuracy, as we do the fitting after gathering all samples.

As an experiment, we selected five samples from the entire dataset for each benchmark, just like in the previous subsection. However, this time we also used binary search. The process was repeated 100 times. Figure 5 illustrates the CDF of the MAPE values obtained from these experiments. The results indicate that using the same number of samples, binary search leads to much more reliable models compared with the random sampling. From an alternative perspective, it shows that binary search requires fewer samples to achieve

[1]Developers can request a quota increase up to 10,240 MB. However, serverless functions' memory usage is typically within the default range [50].

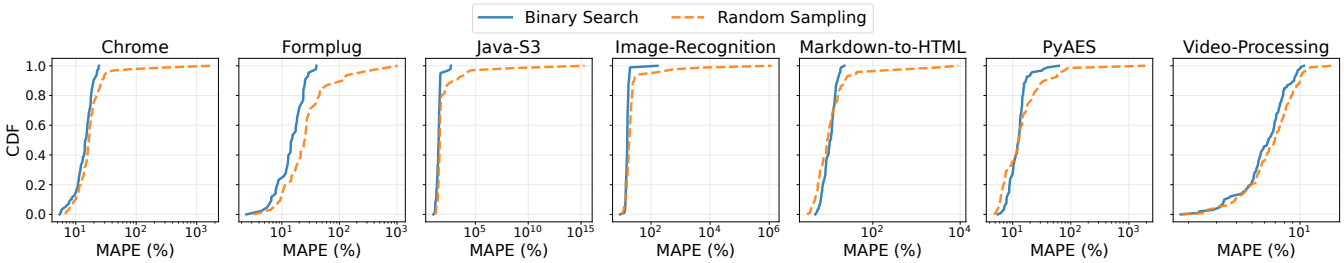

Figure 5: Distribution of MAPE values for two sampling strategies.

the same model accuracy as random sampling. Requiring fewer samples means a faster and more cost-effective right-sizing process. The goal of this experiment was to emphasize the significance of sampling strategy.

A major factor that has been ignored in our analysis so far is that samples from different memory configurations have different costs. A good sampling strategy should balance the cost of exploration with the information gain from each sample. In designing Parrotfish, we take these considerations into account and leverage the knowledge on execution time's shape (exponential family of functions).

4 PARROTFISH

Let us apply the insights we gained from modeling execution time and cost of benchmark serverless functions.

4.1 Online Parametric Regression

Parrotfish combines two seemingly obvious, yet novel ideas: 1) using execution time models instead of black-box optimization to gather fewer samples, and 2) employing a cost-aware sampling strategy. Parrotfish employs online parametric regression. Online means that the decision of where and how many samples to collect is based on the previous samples, and parametric regression means that the samples collected up to each point are fitted to a known function shape (exponential in this case) to train the model parameters. In this section, we walk the reader through the online parametric regression process of Parrotfish and explain its components.

4.1.1 Initiating the sampling process. At first, Parrotfish has no information about the target function. It needs at least a few samples to be able to build a preliminary execution and cost model. Exploring at least three memory configurations is necessary to build an estimation of the decay factor for the exponential execution time function. This is because the most basic exponential decay function has three parameters to be learned, b, N_0, and τ below:

$$ExecTime(m) = b + N_0.e^{\frac{-m}{\tau}} \tag{4}$$

If the user has not specified any constraints on the memory range of the target function, initial memory values of 128 MB, 1,024 MB, and 3,008 MB are used. These values are driven by the binary search strategy (§3.3) on the default memory configuration range for AWS Lambda (128 MB - 3,008 MB). Once the first three initial memory values have been sampled, an iterative process is triggered to determine the subsequent memory size to explore or decide whether to terminate the sampling process and present the results to the developer.

4.1.2 Leveraging the continuous nature of average execution time function. The average execution time as a function of allocated memory is a continuous function, i.e., it has no discontinuities within the operational memory range. For a continuous function, sampling a memory configuration reveals valuable information regarding its surrounding configurations. Particularly, this can be well leveraged as the execution time is monotonic decreasing. For instance, we expect somewhat similar average execution time with 460 MB and 462 MB configurations for a function, as long as the minimum memory requirement is less than the smaller value. As such, sampling one, perhaps the cheaper one, would suffice. On the other hand, inferring information about the 1000 MB configuration based on a sample at 460 MB seems may be a stretch. This qualitative insight can be modeled using a belief function [51] over the search space (the range of possible memory values). The belief function is constant zero before any configuration is explored. Upon exploring a configuration, we can add a unit belief distribution centered around that configuration to the belief function. Empirically, we found a normal distribution with a standard deviation of 200 MB adequately represents the variation in acquired knowledge. We evaluate the impact of this width of this unit normal distribution in §5.6.

4.1.3 Choosing the next memory configuration to explore. At each iteration, the optimal next exploration is at a memory configuration that maximizes gain in belief with minimum required exploration cost. A simple way to account for these two factors is selecting the memory configuration m that minimizes the following joint objective function:

$$\min_m Objective(m) = Cost(m) \times Belief(m) \tag{5}$$

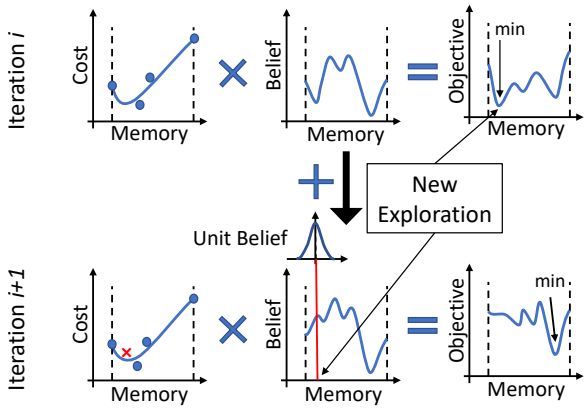

Figure 6: Exploration Cost-Aware sampling strategy in action. The red cross shows the value we have just sampled and the red line shows where the normal distribution is added.

Intuitively, we need the next configuration to be selected where both cost and belief are low. Figure 6 shows this process in action, as well as how the belief function gets updated after an exploration. As discussed earlier in the paper, the capacity cost is a function of execution time and the invocation cost is not optimizable through rightsizing. Therefore, effectively, we are optimizing for the following objective:

$$\min_{m} Objective(m) = k \times m \times ExecTime(m) \times Belief(m), \quad (6)$$

with k being the constant capacity cost defined by the cloud providers. As we accumulate more samples, the execution time model becomes more accurate (refer to §3).

4.1.4 Termination logic. In Figure 3, we demonstrated different cost function shapes and variations. As Parrotfish should be able to handle very different functions and explore each with minimal samples count and cost, a fixed exploration count for all functions cannot be used. Instead, the sampling should be adaptively terminated based on the confidence gained from prior samples. We leverage the belief distribution function described earlier to guide us in determining when to terminate. As unit belief functions get added to the belief function in each iteration, we check if the value of the belief function at the cost-optimal configuration exceeds a threshold (B_{th}), and if so, we conclude the optimization process. The insight is to make sure that our accumulated belief for the recommended configuration is high enough. Empirically, we found B_{th} equal to 2 to strike the right balance between exploration cost and the improvement in locating the cost-optimal configuration. We evaluate the impact of using different termination thresholds in §5.5.

4.1.5 Adaptive sample count per exploration. In order to account for the high variability of serverless execution times, Parrotfish employs adaptive sampling. At least two samples are collected per explored memory configuration. If these two samples show significant variability in their execution times (with a coefficient of variation (CoV) greater than 0.05), an additional sample is collected. This process is repeated as long as the high CoV persists, but it stops once the maximum sample count per configuration is reached (3 by default, tunable by the user).

4.1.6 Optional user constraints. The core objective of Parrotfish is to find the most cost-optimal configuration for the target serverless function. However, cost is not the only criteria developers care about. Parrotfish has two optional constraints to broaden its applicability:

(1) **Maximum Execution Time Constraint:** To accommodate latency limitations, it is common for serverless functions to have an upper threshold on their execution time. To address this consideration, users have the option to specify the maximum acceptable execution time for the function when interacting with Parrotfish. Parrotfish leverages the execution time model to recommend a configuration that minimizes cost while adhering to the specified execution time constraint.
(2) **Cost-Tolerance Best-Performance Constraint:** Parrotfish is agnostic to the execution time of the serverless functions and focuses solely on identifying the configuration with the minimum cost. However, users have the option to define a cost tolerance window to obtain the most performant configuration within a specified range. For example, if the user passes a cost tolerance window of 5% to Parrotfish, the Recommendation Engine's result will be the configuration with the least execution time that incurs a cost no more than 5% higher than the minimum cost. This approach enables users to strike a balance between cost optimization and meeting performance requirements within the specified tolerance.

4.1.7 Putting it all together. We have reviewed various design aspects of Parrotfish's optimization strategy. Let us go over the formulation of the optimization flow, shown in Algorithm 1. The algorithm outlines the sequential steps involved in the tool, beginning with obtaining the initial samples and concluding with the termination of sampling and the return of the final result. Figure 7 shows an overview of Parrotfish's components. The Recommendation Engine is at the heart of Parrotfish, containing the optimization algorithm. It gets the optimization settings as a configuration file from the user. These settings include the function's endpoint that the user wants to optimize, a representative input, the user's cloud access keys (needed to collect logs and configure the function), and the operating memory range of the function if known. The Recommendation Engine, by default, finds the configuration that minimizes the cost of running the

Algorithm 1: Parrotfish's Optimization Flow.

```
Input  : constraints, payload
Output : minimum-cost memory
Function sample: set function's memory and invoke,
  updating belief values
Function curveFit: fits samples with the exp function
defaultMemoryRange = [128, 3008]
initialMemories = [128, 1024, 3008]
/* User-provided range alters defaults   */
beliefValues, samples = [], []
samples += sample(initialMemories, beliefValues)
costModel, executionModel = curveFit(samples)
minCostMemory = argmin(costModel)
beliefPeak = beliefValues[minCostMemory]
while beliefPeak < terminationThreshold do
    /* Default termination threshold is 2 */
    nextSample = argmin(costModel × beliefValues)
    samples += sample(nextSample)
    costModel, executionModel = curveFit(samples)
    minCostMemory = argmin(costModel)
    beliefPeak = beliefValues[minCostMemory]
end
if maxExecutionTimeConstraint then
    executionTimes = executionModel[memories]
    filter = executionTimes < constraint
    return argmin(costModel, filter)
end
if costToleranceWindowConstraint then
    executionTimes = executionModel[memories]
    filter = cost < min(costModel) + constraint
    return argmax(executionTimes, filter)
end
return argmin(costModel)
```

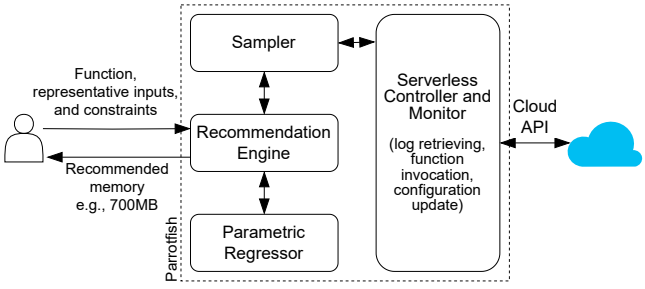

Figure 7: High-level overview of Parrotfish.

serverless function, however, we allow the user to specify optional constraints to get a configuration that meets certain criteria. We explain these next.

4.2 Input Variability

Serverless functions are typically invoked with various inputs. Parrotfish allows developers to specify multiple inputs representative of their production workload. Parrotfish then builds execution time and cost models for each input. Using these models, Parrotfish reports optimal per-input and aggregate configurations. The aggregate recommendation uses the weighted average cost model with the developer-defined weights. For additional savings, developers can use the per-input results as insight to deploy different versions of the function and split the load.

Parrotfish can be used in different settings. If used by developers, it is up to them to identify representative inputs in time periods that suit them and their end-users. If integrated within a serverless platform, invocations can be statistically sampled asynchronously. Ultimately, we focus on solving each problem instance as efficiently as possible.

4.3 Practicality and Versatility

In designing Parrotfish, the usability, versatility, and reliability of the tool was one of our primary concerns. These aspects may not be particularly novel in terms of research. However, we believe that the lack of attention paid to these factors is the reason why developers tend to favor the use of well-designed exhaustive search tools like AWS Lambda Power Tuning [14], instead of more cost-optimal advanced academic serverless rightsizing tools. In fact, when attempting to run some of the prior work on our benchmark functions, we encountered several issues that served as valuable lessons for our tool's development.

One crucial aspect is determining the operational memory range of functions. All existing tools require users to define a working memory configuration [14], or assume that the function operates under all [1] or a predefined [22] set of memory values. Parrotfish automatically determines the operating memory range of the target function based on invocation logs. In situations where out-of-memory errors or timeout errors occur, the tool adjusts the lower bound of the memory range search space. This allows graceful handling of error scenarios and adapting the sampling process accordingly.

Parrotfish, unlike some other tools [22], does not require any changes to the user's code or the use of external libraries to capture runtime behaviors. Instead, we leverage the cloud provider's log collection to retrieve the execution time of the invoked function. This ensures a seamless optimization experience for developers.

It is important to note that Parrtofish is designed for today's serverless offerings, which do not support configuration knobs beyond memory. The memory configuration dictates other resources. However, for future serverless systems with more knobs, Parrotfish can be extended by adding

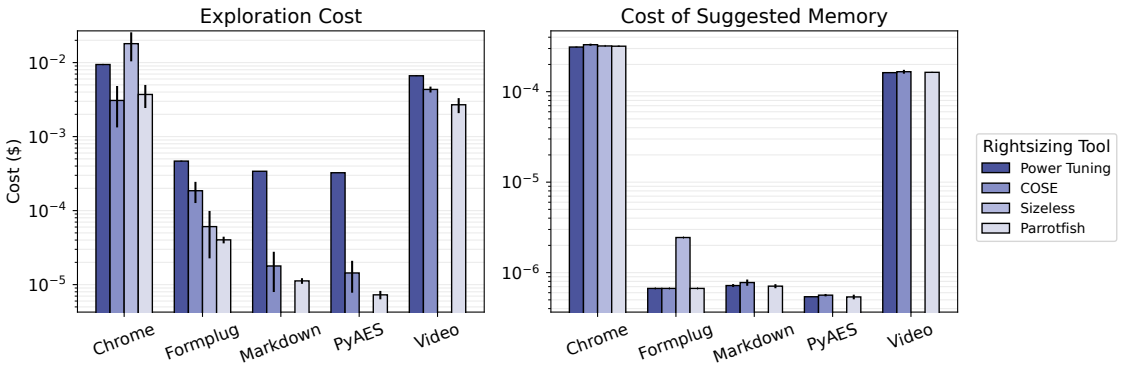

Figure 8: Parrotfish's exploration cost and suggested configurations' real cost compared to Power Tuning, COSE, and Sizeless. Sizeless only supports Node.js functions. Cost is in log scale.

more variables to the model, and changing the sampling strategy to search along multiple dimensions.

5 EVALUATION

We chose AWS Lambda as our serverless provider and deployed our benchmarks on this platform. Parrotfish interacts with Lambda functions, retrieving their execution logs to analyze and optimize their performance. Parrotfish is developed in Python and is platform-independent. For the experiments presented in this paper, we ran Parrotfish on an Ubuntu 22.04 machine. We have also used Parrotfish on an Apple Silicon Mac without any issues. For brevity, we used five of the seven benchmarks introduced in § 3.1 in our experiments.

5.1 Effectiveness of Parrotfish

To evaluate the effectiveness of Parrotfish, we compared it with three state-of-the-art tools:

(1) AWS Lambda Power Tuning [14]: This tool is the recommended tool by AWS to rightsize serverless functions [7]. By default, it employs a process of sampling six predetermined memory configurations (128, 256, 512, 1,024, 1,536, and 3,008 MB) ten times, ultimately identifying the configuration from the presets that offers the lowest average cost. This tool belongs to the category of automated exhaustive search tools.
(2) COSE [1]: Bayesian optimization (BO) has been used by prior work for rightsizing serverless functions [1, 60]. With every sample, BO tries to maximize the confidence in its model. COSE is a complete serverless configuration tool using BO. We compare Parrotfish against it.
(3) Sizeless [22]: Sizeless trains a machine-learning model on thousands of synthetic functions to model the execution time. When applied to real-world functions, it uses this trained model to predict the execution time for previously unseen memory configurations, relying on samples from just a single memory configuration.

We ran each tool ten times and collected its optimization results and exploration costs.

Comparison baseline deployment notes: Sizeless requires a pre-trained model to provide configuration recommendations. This is unlike other tools that learn for each function independently. We use the extensively trained model open-sourced by the Sizeless authors [21], to be faithful to their work. Additionally, the implementation of Sizeless requires code change and using a wrapper that gathers runtime data for the target. This wrapper, however, exclusively supports functions written in Node.js. We were thus limited to evaluate Sizeless only for our Node.js benchmarks, Chrome and Formplug. When it came to our demanding benchmarks that require a minimum of 512 MB of memory (Chrome and Video), Sizeless, COSE, and Power Tuning did not work out of the box. This is because they cannot handle out-of-memory and timeout errors automatically, as discussed in §4.3. To overcome this issue, we had to manually set the memory range for them to obtain results for these benchmarks.

5.1.1 Exploration Cost. Figure 8 shows the exploration cost and the cost of the function with the suggested configuration for Parrotfish, COSE, Sizeless, and Power Tuning. Parrotfish demonstrates significantly lower exploration costs compared to Power Tuning, Sizeless, and COSE, with a geometric average reduction of 9.96×, 2.70×, and 1.81×, respectively. We used geometric mean instead of arithmetic mean as we are averaging the reductions for each benchmark [29].

Power Tuning uses 10 invocations per configuration to mitigate cloud variations, resulting in more consistent configurations but higher exploration costs. Sizeless uses a total of 50 invocations for its inference process. In the case of Chrome, these 50 samples fall within the expensive segment of the application, resulting in a significant exploration cost. Conversely, in Formplug, these 50 samples happen to be positioned close to the minimum cost, resulting in a reduced exploration cost compared to Powertuning and COSE, despite having a larger number of samples. Note that sizeless

relies on a pre-trained model. We did not consider the cost of training the model in the exploration cost. In reality, the exploration cost of sizeless will be much more substantial.

COSE only samples once for each configuration, and requires additional explorations to boost model confidence, especially for highly variable functions like Formplug. In contrast, the white-box model of the execution time, limits the changes of the function, enabling it to converge with fewer explorations (often converging with just four samples in most benchmarks). Moreover, Parrotfish is more consistent in terms of exploration costs compared to COSE.

> Parrotfish outperforms Power Tuning, Sizeless, and COSE in terms of exploration cost, achieving reductions of 9.96×, 2.70×, and 1.81×, respectively.

Cheaper rightsizing enables and encourages frequent reconfigurations, leading to compounded long-term savings.

5.1.2 Suggested Memory Cost. We have made several observations regarding the cost of the suggested memory configurations. Firstly, for benchmarks where the minimum-cost configuration closely aligns with the previously explored samples, all the tools provide similar suggestions. For instance, the minimum-cost configuration for Formplug is 128 MB, a value that all tools except Sizeless have sampled during their exploration.

Secondly, for benchmarks with minimum-cost configurations that differ from the previously sampled configurations, our system (Parrotfish) performs slightly better. For instance, Markdown's optimal memory configuration is around 1100 MB, and Parrotfish accurately captures this, resulting in cost reductions of 8.80% and 1.52% compared to Power Tuning and COSE, respectively.

In case of high variability, such as Chrome-Screenshot, COSE, Sizeless, and Parrotfish yield comparable results, but their suggestions are generally inferior to those of Power Tuning. This is because Power Tuning takes more samples, enabling it to reduce observation noise.

On average (geometric), our approach reduces the cost of running benchmarks by 47.83% and 3.76% compared to Sizeless and COSE, respectively. However, compared to Power Tuning, our suggested cost is slightly worse by 0.18%.

> Parrotfish finds optimal or near-optimal configurations for all the benchmarks, and suggests comparable or better configurations compared to other tools.

5.2 Handling Input Variability

An often overlooked factor in any optimization tool is the impact of input on the execution time and cost of a serverless function. To illustrate this point, we use the Video-Processing benchmark. Video-Processing function accepts a video file

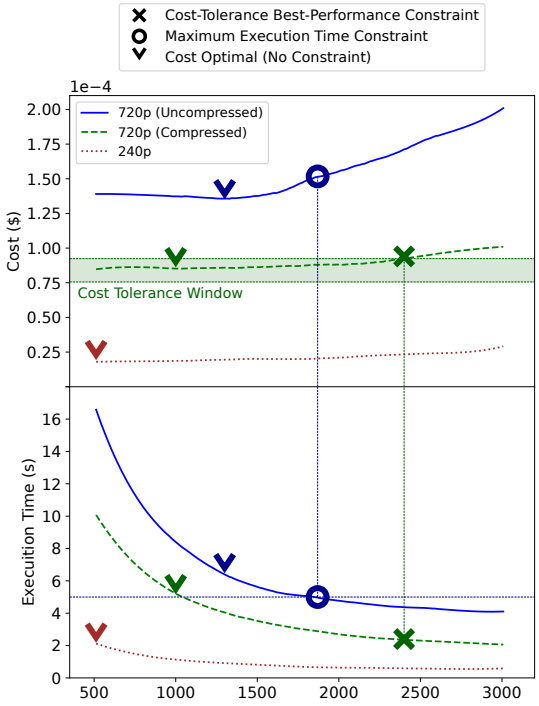

Figure 9: Different inputs to the same function affect the cost optimal configuration ("v" markers). Users can express optional constraints to communicate execution time limit ("o" markers), or their cost optimality tolerance to gain performance ("x" markers).

as input, applies a watermark to the video, and converts it to GIF format. As the video quality improves, the complexity of the task also increases, necessitating more resources. Figure 9 demonstrates this growth in action. When processing a low-quality 240p video, the function requires minimal resources, and the increase in cost outweighs the performance gains. However, as the video quality rises, additional resources enhance the function's performance substantially, pushing the minimum-cost configuration to higher memory values. Parrotfish leverages user-provided input to sample the function and construct a model. When presented with various video qualities as individual inputs, Parrotfish can identify the minimum-cost configuration, as indicated by the caret symbols in the figure. Parrotfish supports weighted inputs that enables developers to replicate real-world inputs and minimize the average cost of running the function.

To demonstrate how Parrotfish supports multiple inputs, we used the Video-Processing and Java-S3 benchmarks. We defined two sets of two inputs consisting of videos with different qualities for Video-Processing and one set of two inputs containing images with various sizes for Java-S3. Table 4 summarizes the results of this experiment. Parrotfish reports both per-input and aggregate optimized configurations.

Function	Input Set	Per Input Optimized Memory Config	Aggregate Memory Config
Video	100 KB 240p Video 265 MB HD Video	512 MB 1331 MB	1226 MB
Video	100 KB 240p Video 3 MB 360p Video	512 MB 512 MB	512 MB
Java-S3	21 KB Image 11 MB Image	1501 MB 128 MB	128 MB

Table 4: Parrotfish optimizes per-input function costs and reports the aggregate optimal configuration based on provided input weight (equal weights used here).

5.3 User Constraints

In §4.1.6, we discussed the two optional constraints that developers can use to communicate goals beyond cost optimization: (1) Maximum Execution Time, which sets a limit on the maximum allowable execution time for the suggested configuration, and (2) Cost-Tolerance Best-Performance, which aims to identify the most performance configuration within a specified tolerance from the optimal cost.

To demonstrate the maximum execution time constraint, we use the Video-Processing benchmark with a high-quality, uncompressed 720p input video file. For this input, the benchmark exhibits a fourfold difference between its minimum and maximum execution times in operable memory range. As an example scenario, this function might be used in a video serving platform, where the team aims for an average latency of no more than 5 seconds (the blue horizontal line). With this large input, the cost optimal configuration is at 1,300 MB. That would result in an average execution time of ~7 seconds (shown as blue "v" markers). Using the maximum execution time constraint, Parrotfish identifies the configuration that meets the user-specified 5-second execution time with the lowest cost: 1,870 MB (blue circle markers).

To demonstrate the cost-tolerance best-performance constraint, we use the same benchmark with a compressed 720p video (size reduced). The cost of this function exhibits a relatively small variation, as depicted by the green line in Figure 9, while its execution times vary significantly, with a fivefold difference. In the absence of any constraints, Parrotfish identifies the cost optimal memory of 1,000 MB for this function, resulting in an execution time of ~5.5 seconds, as indicated by the green "v" markers. By defining a 10% cost tolerance (shown highlighted region), Parrotfish searches for the lowest execution time within 90-110% of the minimum cost. It suggests an optimal memory configuration of 2,400 MB, leading to a 2.4-second execution time, which corresponds to a 2.29-fold decrease in execution time with only a 10% cost increase, as represented by green cross markers.

5.4 Cost-Aware Sampling Strategy

To show the effectiveness of our cost-aware sampling strategy, we conducted a comparative analysis against the random and binary-search approaches, introduced in §3.3. We evaluated both the exploration cost and the cost of the suggested memory. Figure 10 shows the result of this comparison. We conducted 100 iterations of the experiment, using five samples for each. For a fair comparison, we disabled the dynamic sampler (§4.1.5) while sampling, so each sampling strategy invoked the benchmark function exactly five times. Using the cost-aware sampling strategy, we observed a geometric average reduction of 9.36% and 13.18% in exploration costs compared to random sampling and binary search, respectively. This reduction was achieved without compromising the quality of suggested cost, with an improvement of 2.89% compared to random sampling and only 0.74% worse than the binary search strategy.

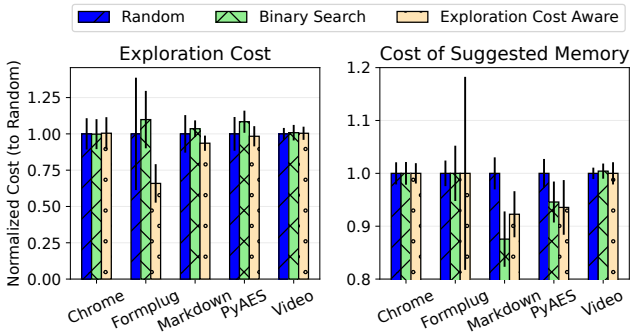

Figure 10: Parrotfish's cost of suggested memories and exploration cost using different sampling strategies. Binary search incurs a higher exploration cost, while random search suggests suboptimal configurations. Note that the y-axis range for the right subplot is magnified.

It is important to note that model accuracy does not always yield proportional cost savings. The cost functions for Chrome and Video-Processing benchmarks are almost flat (Figure 3), so greater memory recommendation noise is tolerable. Conversely, Markdown and PyAES have dominant minima, for which the higher model accuracy leads to lower average costs in Figure 10.

5.5 Termination Logic Sensitivity Study

The decision of when to conclude the exploration process is about hitting a balance between accuracy and exploration cost. Exploring too few configurations reduces the exploration cost, but may lead to premature parametric regression, and thus inaccurate recommendations. On the other hand, excessive explorations can needlessly increase the cost of exploration. As described in §4.1.4, Parrotfish terminates the exploration loop when the belief function at the recommended configuration exceeds a threshold (B_{th}). Here, we evaluate the impact of using different termination thresholds.

Figure 11 showcases the impact of using different belief thresholds. Using a threshold of 1 minimizes the exploration

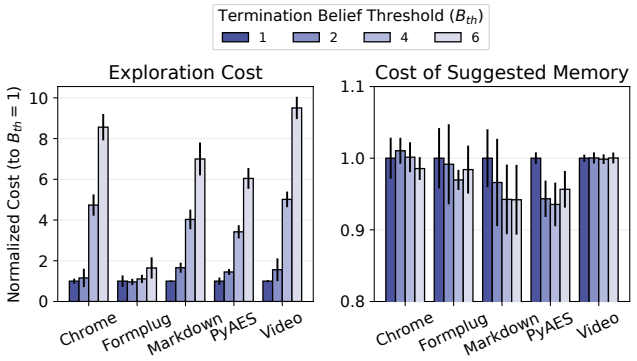

Figure 11: Effect of different termination belief thresholds. Increasing the termination threshold significantly affects the exploration costs. Note that the y-axis range for the right subplot is magnified.

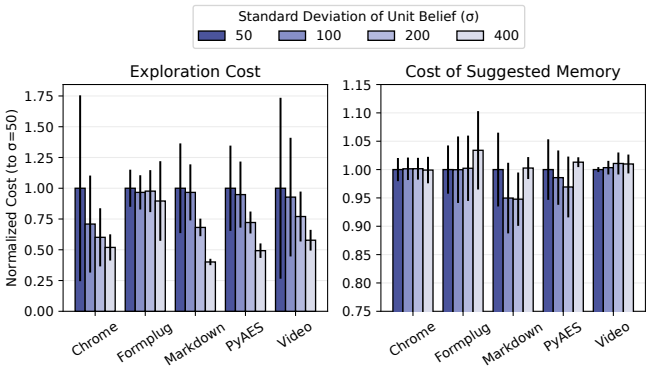

Figure 12: Effect of different standard deviations (σ) for the unit belief distribution. Increasing σ reduces the exploration cost, but causes subpar recommendations. The y-axis range in the right subplot is magnified.

cost. However, the suggested configurations will not optimal for three of the benchmarks, Formplug, Markdown-to-HTML, and PyAES. Increasing the threshold allows for gathering more samples and subsequently recommend higher quality configurations. However, increasing the threshold beyond 2 substantially increases the exploration cost, with marginal improvement in the suggested configurations. This is why we chose a default B_{th} of 2. This parameter is tunable by the user, in case they prioritize higher quality estimates over exploration cost.

5.6 Unit Belief Sensitivity Study

In §4.1.2 we discussed that the continuity of the average execution time function allows us to infer some information on the surroundings of the sampled configuration. The reliability of this information inference decreases as the distance from the sampled configuration increases. That is why the unit belief function has a normal (Gaussian) distribution. The choice of the standard deviation (σ) for this unit belief distribution affects the sampling process. Figure 12 shows the impact of using different σ values. A narrower unit belief distribution (smaller σ) translates into more exploration to gain the required confidence in the optimal configuration. On the other hand, a wider unit belief distribution (larger σ) enables inferring more information on configurations farther from each sampled memory, potentially leading to a premature belief in the optimal memory configuration. We chose 200 MB for σ as it hits a balance between exploration cost and recommendation quality.

5.7 Tail Execution Time Model Accuracy

The application of execution time models in this paper has been for function rightsizing. Given that the capacity cost of serverless functions is determined by the aggregate monthly usage, the appropriate variant of execution time to model in the context of rightsizing is average execution time. This is because multiplying the average cost of an invocation by the number of invocations yields the aggregate monthly cost. In this section, we take a step back and further explore the modeling discussion of §3 outside the context of Parrotfish. In particular, we want to know which family of functions can decently model the tail execution time. While not applicable for rightsizing, this information can be used by other researchers in designing better scheduling and load-balancing in serverless settings to suite latency-critical applications.

Benchmark	Chrome	Formplug	Image	Java-S3	Markdown	PyAES	Video
MAPE	2.71	11.26	12.53	2.33	8.95	11.54	2.27

Table 5: The exponential function had the best fit to the 95th percentile of execution time data. MAPE values indicate a highly accurate fit for all of our benchmarks.

We used the data we had gathered for sampling and used the OriginPro [38] tool again to all supported families of functions. This time, we picked the 95th sample of each configuration (out of 100 samples), and used these values for fitting. The MAPE values for the most optimal fitting function are presented in Table 5, which again turned out to be the exponential function.

6 RELATED WORK

Rightsizing cloud resources have been studied previously in the context of VMs. Rightsizing VMs is a different problem to solve, as the search space has more dimensions, but fewer configurations per dimension compared with serverless functions. Some of the prominent work on rightsizing VMs includes CherryPick [2] and PARIS [58], which both use black-box learning algorithms to estimate the performance model of a VM. Ernest [55], is another approach that uses a linear model to estimate the performance of the jobs running on VMs under different configurations.

Tool/Study	Methodology	Primary Objective	Open-Source Tool
Aquatope [60]	Bayesian Optimization	Resource Management	✗
AWS Compute Optimizer [4]	Machine Learning	Workload Rightsizing	✗
COSE [1]	Bayesian Optimization	Cost Minimization	✓
CPU-TAMS [17]	CPU Time Accounting	Cost Minimization	✗
Lachesis [52]	Supervised Learning	Cost Minimization	✗
ORION [35]	Linear Interpolation	E2E Latency Estimation	✓
Power Tuning [14]	Automated Search	Cost Minimization	✓
SAAF [16]	CPU Time Accounting	Performance Prediction	✓
Sedefoğlu et al. [47]	Regression Analysis	Cost Minimization	✗
Sizeless [22]	Multi-Target Regression	Cost Minimization	✓
SLAM [43]	Automated Search	SLO Compliance	✗
StepConf [57]	Global-Cached Most Cost-effective Critical Path	Cost Minimization	✗
Parrotfish (this work)	Parametric Regression	Cost Minimization	✓

Table 6: Overview of serverless optimization studies.

Additionally, cloud providers offer services to help developers rightsize their resources. For example, AWS Compute Optimizer [4] is a tool offered by AWS, leveraging historical utilization data and ML to rightsize cloud resources (including serverless functions). However, users do not have control over triggering these services, they require many logs, and do not allow expression of complex objectives.

Table 6 summarizes the body of work pertaining to rightsizing serverless functions. Some are not usable by practitioners due to the absence of open-source tools or artifacts. Parrotfish has been extensively tested and is designed for versatile operation with minimal requirements for developers. Next, we detail the objectives and methodologies used by various serverless rightsizing systems.

Exhaustive search: AWS Lambda Power Tuning [14] is a widely adopted, automated exhaustive search tool. It leverages a predefined set of memory configurations to identify the most optimal one. Being endorsed by AWS [7], it is a prominent serverless rightsizing tool used by developers [27, 37, 53]. Sedefoğlu et al. [47] also sample fixed memory configurations in their paper. They suggest using regression analysis to estimate serverless function execution time for cost optimization, without going into specifics. The authors of SLAM [43] propose using a max heap data structure to reduce the time complexity of automated search. Compared to these exhaustive search strategies, Parrotfish offers a significant advantage by having lower exploration cost.

Bayesian Optimization (BO): COSE [1] is a black-box learning-based approach that uses BO to model the performance of serverless functions and rightsize them. BO has also been recently used to prioritize QoS and uncertainty considerations within the platform [60]. In contrast to BO approaches, Parrotfish employs white-box learning, wherein the high-level model is known, and only model parameters need to be trained online. This prevents overfitting, and leads to improved recommendations with fewer samples.

Pre-trained models + profiling: Sizeless [22] represents an alternative learning-based approach. It gathers runtime data from a wide range of synthetic functions and builds a regression model to recommend configurations for new functions using only a limited number of profiling runs. CPU-TAMS [17] and SAAF [16] adopt a similar concept, but with less exhaustive metrics. Unlike Parrotfish, these tools require code modifications or the use of custom libraries to gather runtime metrics. This adds extra effort and may not support all programming languages, thus limiting their applicability. Additionally, these tools often rely on large trained models, resulting in significant cost overheads.

Lachesis [52] is a resource allocation framework that uses supervised learning to predict the resources that a function would need for each invocation based on its input characteristics. Lachesis eliminates the need for users to specify the memory configuration for the function. Parrotfish, unlike Lachesis, is not in the critical path of the function's execution, resulting in less overheads.

Workflow optimization: ORION [35] is designed to estimate the end-to-end (E2E) latency of serverless workflows and improving it by allocating right amount of resources to functions. To build the execution time model of each function, ORION samples it at three fixed configurations where different are expected to saturate. It then uses percentile-wise linear interpolation to approximate the underlying non-linear model. StepConf [57] uses heuristic algorithms to optimize the performance and cost of serverless workflows. Parrotfish can be used to complement these tools, providing them with more accurate per-function execution time models with low sampling cost.

7 CONCLUSION

Parrotfish is a serverless cost optimization tool employing an online parametric regression strategy. Parrotfish can reduce the exploration cost of optimization by an average (geometric) of 1.81-9.96× compared to the state-of-the-art serverless optimization tools. The recommended configurations also had 25.74% lower cost on average. It is publicly available at https://github.com/ubc-cirrus-lab/parrotfish.

ACKNOWLEDGMENTS

We thank Sathish Gopalakrishnan for constructive early-stage discussions and members of the UBC CIRRUS Lab for their feedback on this work. We also thank anonymous reviewers and our shepherd, Kostis Kaffes, for helping us improve the paper. We thank Efe Evci, Erik Langille, Jacob Grossbard, Skylar Liang, and Yaman Malkoc for conducting early explorations of this project. This work was supported in part by NSERC grants RGPIN-2021-03714 and DGECR-2021-00462 and the Mitacs GRI program. This work was enabled by the Digital Research Alliance of Canada, Google Cloud Research Credits, and AWS Cloud Credits for Research.

REFERENCES

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 469–482.

[3] Artem Arkhipov. 2022. How to Get Headless Chrome Running via Puppeteer on AWS Lambda. Retrieved 2023-09-25 from https://www.techmagic.co/blog/running-headless-chrome-with-aws-lambda-layers/

[4] AWS. 2023. AWS Compute Optimizer. Retrieved 2023-09-22 from https://aws.amazon.com/compute-optimizer/

[5] AWS. 2023. AWS Lambda Pricing. Retrieved 2023-09-25 from https://aws.amazon.com/lambda/pricing/

[6] AWS. 2023. Configuring Lambda Function Options. Retrieved 2023-09-25 from https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[7] AWS. 2023. Profiling functions with AWS Lambda Power Tuning. Retrieved 2023-09-25 from https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html

[8] AWS. 2023. S3 image resizer (Java). Retrieved 2023-09-18 from https://github.com/awsdocs/aws-lambda-developer-guide/tree/main/sample-apps/s3-java

[9] AWS. 2023. Troubleshooting Lambda configurations. Retrieved 2023-09-25 from https://docs.aws.amazon.com/lambda/latest/operatorguide/configurations.html

[10] Azure. 2023. Azure Functions pricing. Retrieved 2023-09-25 from https://azure.microsoft.com/en-us/pricing/details/functions/

[11] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. 2021. On Merits and Viability of Multi-Cloud Serverless. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. ACM, 600–608.

[12] Mike Bailey. 2019. Right-sizing your AWS Lambdas. Retrieved 2023-09-25 from https://mike.bailey.net.au/2019/05/right-sizing-your-aws-lambdas/

[13] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input Size-Sensitive Container Provisioning and Request Scheduling for Serverless Platforms. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*. ACM, 257–272.

[14] Alex Casalboni. 2019. AWS Lambda Power Tuning. Retrieved 2023-06-06 from https://github.com/alexcasalboni/aws-lambda-power-tuning

[15] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. ACM, 64–78.

[16] Robert Cordingly, Wen Shu, and Wes J. Lloyd. 2020. Predicting Performance and Cost of Serverless Computing Functions with SAAF. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 640–649. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00111

[17] Robert Cordingly, Sonia Xu, and Wes Lloyd. 2022. Function Memory Optimization for Heterogeneous Serverless Platforms with CPU Time Accounting. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 104–115.

[18] Datadog. 2021. The state of serverless. Retrieved 2023-09-25 from https://www.datadoghq.com/state-of-serverless-2021/

[19] Datadog. 2022. The state of serverless. Retrieved 2023-09-25 from https://www.datadoghq.com/state-of-serverless-2022/

[20] Arnaud de Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. 2016. Mean Absolute Percentage Error for regression models. *Neurocomputing* 192 (jun 2016), 38–48. https://doi.org/10.1016/j.neucom.2015.12.114

[21] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Data and evaluation scripts for the manuscript "Sizeless: Predicting the optimal size of serverless functions". Retrieved 2023-05-07 from https://codeocean.com/capsule/6066333

[22] 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*. ACM, 248–259.

[23] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*. 265–276.

[24] Mohamed Elsakhawy and Michael Bauer. 2021. Performance Analysis of Serverless Execution Environments. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. 1–6.

[25] Alexander Fuerst and Prateek Sharma. 2022. Locality-Aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*. ACM, 227–239.

[26] Samuel Ginzburg and Michael J. Freedman. 2020. Serverless isn't server-less: Measuring and exploiting resource variability on cloud FaaS platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 43–48.

[27] Wenqi Glantz. 2023. Automating performance optimization with AWS Lambda Power Tuning. Retrieved 2023-06-06 from https://betterprogramming.pub/automating-performance-optimization-with-aws-lambda-power-tuning-d295e7141ecc

[28] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. 2022. Resource-Centric Serverless Computing. *arXiv preprint arXiv:2206.13444* (2022).

[29] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[30] IBM. 2023. IBM Cloud Functions Pricing. Retrieved 2023-09-25 from https://cloud.ibm.com/functions/learn/pricing/

[31] Daniel Ireson. 2021. Formplug. Retrieved 2023-09-25 from https://github.com/danielireson/formplug/

[32] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

[33] Mohamed Labouardy. 2019. How We Reduced Lambda Functions Costs by Thousands of Dollars. Retrieved 2023-09-25 from https://labouardy.com/how-we-reduced-lambda-functions-costs-by-thousands-of-dollars/

[34] Danielle Lambion, Robert Schmitz, Robert Cordingly, Navid Heydari, and Wes Lloyd. 2022. Characterizing X86 and ARM Serverless Performance Variation: A Natural Language Processing Case Study *(ICPE '22)*. ACM, 69–75.

[35] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 22). USENIX Association, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[36] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. ACM, 228–244.

[37] Sebastien Napoleon. 2023. How to optimize your lambda functions with Aws Lambda Power Tuning. Retrieved 2023-09-25 from https://dev.to/aws-builders/how-to-optimize-your-lambda-functions-with-aws-lambda-power-tuning-10h5

[38] OriginLab. 2023. OriginPro, Version 2023b. Retrieved 2023-09-25 from https://www.originlab.com/origin OriginLab Corporation, Northampton, MA, USA..

[39] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, Article 10, 16 pages.

[40] Anish Pimpley, Shuo Li, Anubha Srivastava, Vishal Rohra, Yi Zhu, Soundararajan Srinivasan, Alekh Jindal, Hiren Patel, Shi Qiao, and Rathijit Sen. 2021. Optimal resource allocation for serverless queries. *arXiv preprint arXiv:2107.08594* (2021).

[41] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, 753–767.

[42] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. 2023. UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 879–896.

[43] Gor Safaryan, Anshul Jindal, Mohak Chadha, and Michael Gerndt. 2022. SLAM: SLO-Aware Memory Optimization for Serverless Applications. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, 30–39.

[44] Joel Scheuner, Rui Deng, Jan-Philipp Steghöfer, and Philipp Leitner. 2022. CrossFit: Fine-grained Benchmarking of Serverless Application Performance across Cloud Providers. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 51–60.

[45] Trever Schirmer, Nils Japke, Sofia Greten, Tobias Pfandzelter, and David Bermbach. 2023. The Night Shift: Understanding Performance Variability of Cloud Serverless Platforms. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME '23)*. ACM, 27–33.

[46] Gideon Schwarz. 1978. Estimating the Dimension of a Model. *The Annals of Statistics* 6, 2 (1978), 461 – 464. https://doi.org/10.1214/aos/1176344136

[47] 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)*. ACM, 83–85.

[48] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 1063–1075.

[49] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. 2021. Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. ACM, 319–334.

[50] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Article 14, 14 pages.

[51] Prakash P Shenoy and Glenn Shafer. 1990. Axioms for probability and belief-function propagation. In *Machine intelligence and pattern recognition*. Vol. 9. Elsevier, 169–198.

[52] Prasoon Sinha, Kostis Kaffes, and Neeraja J. Yadwadkar. 2023. Online Learning for Right-Sizing Serverless Functions. In *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*. https://openreview.net/forum?id=4zdPNY3SDQk

[53] Brett Uglow. 2022. Easy lambda optimization. Retrieved 2023-09-25 from https://medium.com/digio-australia/easy-lambda-optimization-c2f2e6e49515

[54] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, 559–572.

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

[56] Howard Dean Watts. 2020. lambda-OCRmyPDF. Retrieved 2023-09-22 from https://github.com/chronograph-pe/lambda-OCRmyPDF

[57] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1868–1877. https://doi.org/10.1109/INFOCOM48880.2022.9796962

[58] 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)*. ACM, 452–465.

[59] 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)*. ACM, 724–739.

[60] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-Stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2023)*. ACM, 1–14.