Cypress : Input size–Sensitive Container Provisioning and Request Scheduling for Serverless Platforms

Vivek M. Bhasi
The Pennsylvania State University
vmb5204@psu.edu

Jashwant Raj Gunasekaran
Adobe Research
jgunasekaran@adobe.com

Aakash Sharma
The Pennsylvania State University
abs5688@psu.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
mtk2@psu.edu

Chita Das
The Pennsylvania State University
cxd12@psu.edu

Abstract

The growing popularity of the serverless platform has seen an increase in the number and variety of applications (apps) being deployed on it. The majority of these apps process user-provided input to produce the desired results. Existing work in the area of input-sensitive profiling has empirically shown that many such apps have input size–dependent execution times which can be determined through modelling techniques. Nevertheless, existing serverless resource management frameworks are agnostic to the input size–sensitive nature of these apps. We demonstrate in this paper that this can potentially lead to container over-provisioning and/or end-to-end Service Level Objective (SLO) violations. To address this, we propose *Cypress*, an input size–sensitive resource management framework, that minimizes the containers provisioned for apps, while ensuring a high degree of SLO compliance. We perform an extensive evaluation of *Cypress* on top of a *Kubernetes*-managed cluster using 5 apps from the AWS Serverless Application Repository and/or OpenFaaS Function Store with real-world traces and varied input size distributions. Our experimental results show that *Cypress* spawns up to 66% fewer containers, thereby, improving container utilization and saving cluster-wide energy by up to 2.95× and 23%, respectively, versus state-of-the-art frameworks, while remaining highly SLO-compliant (up to 99.99%).

CCS Concepts

• **Computer systems organization → Cloud Computing**; *Resource-Management*; *Scheduling*.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563464

Keywords

serverless, input size, resource-management, scheduling

ACM Reference Format:
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 *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563464

1 Introduction

The growth in popularity of serverless platforms is being accompanied by a rapid increase in the variety of applications (apps) being deployed on them [22, 36, 37, 50]. The vast majority (~96%) of these apps are short-running, with execution times lesser than 30s [65]. This includes apps such as *Sentiment Analysis* and *Audio Translation*, which are being deployed in user-facing settings like automated chat moderation for real-time events [8, 10, 20] and advanced multimedia messaging [13, 21, 23], respectively. Being user-facing, such apps have been envisioned being administered under strict SLOs in terms of response time requirements [2, 27, 39, 43, 55, 58, 59]. Achieving a low tail latency is also critical for these apps as it is an integral part of the Quality of Service (QoS) delivered to the end-user [26, 27, 33, 34, 52, 55, 56, 62].

A significant portion of these apps are composed of one or more functions that process user-provided inputs to produce the desired output [11, 12, 16, 18, 35]. Existing work in the area of input-sensitive profiling has shown that many such apps, which we will call Input size–Sensitive apps (IS apps), have execution times that depend heavily on the provided input size [25, 31, 35, 38, 41]. This introduces challenges with respect to container provisioning and maintaining a high degree of SLO compliance, that are not currently addressed by state-of-the-art serverless Resource Management frameworks (RM frameworks) in both industry and academia [9, 12, 16, 27, 40, 43, 55]. The two main concerns that arise from this are:

• Existing RM frameworks, being agnostic to the input size–sensitive characteristics of the above apps, resort to using

the average execution times of their composite functions to determine the number of containers needed during autoscaling decisions. As we will discuss later, this leads to spawning an inappropriate number of containers.

• Incoming requests to the IS app will inevitably have starkly different execution times depending on their corresponding input sizes. Due to having an agreed-upon SLO for the app [27, 43], requests with a higher execution time will have a lower 'buffer time' before violating the SLO. Existing RM frameworks do not account for this.

To address these concerns, we propose *Cypress*[1], an input size–sensitive serverless RM framework that minimizes resource consumption, while maintaining a high degree of SLO compliance by virtue of its policies that adapt to the distribution of the input sizes associated with the request trace (we will, henceforth, refer to this simply as input size distribution). Two of *Cypress*'s chief policies are *Input size–Sensitive Request Batching (IS Batching)* and *Input size–Sensitive Request Reordering (IS Reordering)*. *IS Batching* refers to batching multiple requests onto fewer containers by taking their input size–dependent execution times and respective SLOs into account. *IS Reordering* reorders requests in the incoming request queue such that requests with higher potential execution times (and typically, higher input sizes) are executed first so as to meet the SLOs of as many requests as possible. These policies are utilized by the scaling services of *Cypress*, which are: (i) the Proactive Scaler, that deploys containers for functions in advance through prediction of future request loads and input size distributions, (ii) the Reactive Scaler, that scales containers appropriately to recover from potential resource mismanagement resulting from possible load/input size mis-predictions of the other scaling services, and (iii) the Look-Ahead Scaler, that allocates containers for downstream functions in advance as requests arrive at the initial functions in multi-function apps. Apart from *IS Batching* and *IS Reordering*, *Cypress* leverages *Chained Prediction*, wherein the input size distribution of downstream functions are also predicted using a variation of input-sensitive profiling, so as to cater specifically to the idiosyncrasies of multi-function apps. Note that we focus on input-sensitive profiling specific to input sizes, which we will henceforth refer to as Input size–Sensitive Profiling (IS Profiling). This is leveraged in all relevant aspects of *Cypress*'s architecture to facilitate the above policies/services.

We implement *Cypress* using *OpenFaaS*, an open source serverless framework [17], and extensively evaluate it using real-world datacenter traces subject to various input size distributions on a 288 core *Kubernetes* cluster. Our results show that *Cypress* spawns up to 66% fewer containers, thereby, improving container utilization and cluster-wide energy savings by up to 2.95× and 23%, respectively, versus state-of-the art serverless RM frameworks. Furthermore, *Cypress* guarantees the SLO requirements for up to 99.99% of requests.

[1]Our scheme adapts to practically all input size distributions of the request trace, mimicking the climate resilience of the Cypress evergreen tree.

Features	Atoll [55]	Serverless Cost Prediction [35]	Power-chief [63]	Fifer [40]	Xanadu [32]	GrandSLAm [43]	Sequoia [57]	Hybrid Histogram [51]	Cirrus [30]	*Q-Zilla* [46]	*Kraken* [27]	*Cypress*
SLO Guarantees	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Input size–Sensitive Profiling	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Future Input Size Prediction for function chains	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Input size–Sensitive Batching	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Input size–Sensitive Request Prioritization	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Energy Efficiency	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓
Request Arrival Prediction	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	✓	✓
Satisfactory Tail Latency	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of *Cypress* against related work pertaining to serverless and/or microservice-based applications.

2 Background and Motivation

This section provides a brief background of serverless computing and input size–sensitive functions, followed by the motivation for incorporating input size–sensitive policies into RM frameworks.

2.1 Serverless Computing

In serverless computing, the user writes code for a function, uploads it to the serverless platform and registers an event (such as an HTTP request or a file upload) to invoke the function (henceforth, we will use 'event' and 'request' interchangeably). Depending on the app, these functions may process input provided as part of an incoming request and/or input files from a storage service to produce the desired output (which may also be a file that gets stored). Such functions are supported in commercial serverless platforms including AWS Lambda [9], Microsoft Azure Functions [16] and Google Cloud Functions [12]. The registered event triggers function execution, possibly accompanied by a "cold start" latency, which is associated with launching a new container, setting up the runtime environment and deploying the function by downloading its code.

Cold starts can take up a significant proportion of a function's response time (up to tens of seconds [5, 6]) and can, thus, lead to SLO violations. That being the case, a wealth of existing research [24, 28, 29, 47, 48, 54, 64] focuses on mitigating cold start overheads. Another approach involves hiding the effects of cold starts via proactive container provisioning [19, 32, 55, 57]. Additionally, some of these provisioning policies minimize containers by batching multiple requests onto fewer containers, as opposed to spawning one for each request (as is the norm) [27, 40, 43]. *Kraken* [27] and *Fifer* [40] are two such serverless RM frameworks that accomplish batching using the average execution time and function's SLO. For a comparison of *Cypress* against related work, we refer the reader to Table 1.

2.2 Input size–Sensitive Functions

Interestingly, works such as [25, 31, 35, 38, 41] have demonstrated that the execution time of some functions can vary depending on various input parameters. Input-Sensitive profiling encompasses a set of techniques which can be used to empirically determine the execution time of individual routines as a function of such parameters. In particular, this can be done using the input size as the input parameter. We refer to this technique as Input size–Sensitive Profiling (IS Profiling). One key aspect in which this differs from theoretical computational complexity (that uses notations such as Big-O bounds to describe the scalability of algorithms) is that it can actually estimate the *value of running time* of the function *in practice*, as opposed to identifying the bounding factors of the growth rates of algorithms. IS Profiling typically achieves this by (i) performing multiple profiling runs of the function/routine with workloads spanning several magnitudes of input size, (ii) observing the performance and (iii) fitting these observations to a statistical model that predicts metrics (such as execution time) as a mathematical function of workload size. Note that this is a generalized approach that produces a model applicable to any input size. As we will demonstrate, the above techniques also apply to certain serverless functions, which we refer to as *Input size–Sensitive functions* (IS functions). Table 2 presents some IS apps from the AWS Serverless Application Repository [11] and/or the OpenFaaS Function Store [18]. Below, we investigate the implications of IS functions on state-of-the-art serverless resource provisioning policies.

IS App	Composite Functions
Sentiment Analysis	*Sentiment Analysis*
QR Code	*QR Code*
Image Compression	*Image Compression*
Email Categorization	*Text Summary → Text Classify*
Audio Translation	*Audio Transcribe → Text Translate → Send Message*

Table 2: Examples of IS apps from the AWS serverless app repository and OpenFaaS function store. The last two apps consist of multiple functions.

2.3 Motivation

Challenge 1: Input size–Sensitive Container allocation As mentioned earlier, the container provisioning policies of some state-of-the-art works (such as *Kraken* [27] and *Fifer* [40]) batch requests using the average execution times and SLOs of functions (relative to the app's SLO) to determine the function's batch size (defined as the number of requests that can be served by a container without violating the SLO). This, in turn, is used to calculate the required number of containers. However, as we will demonstrate shortly, when the execution time is input size–dependent, request batching using average execution times proves to be inaccurate, as there is considerable variation in request execution times. This can lead to inappropriate container provisioning. Additionally, the input size distribution itself has to be predicted to facilitate proactive container provisioning.

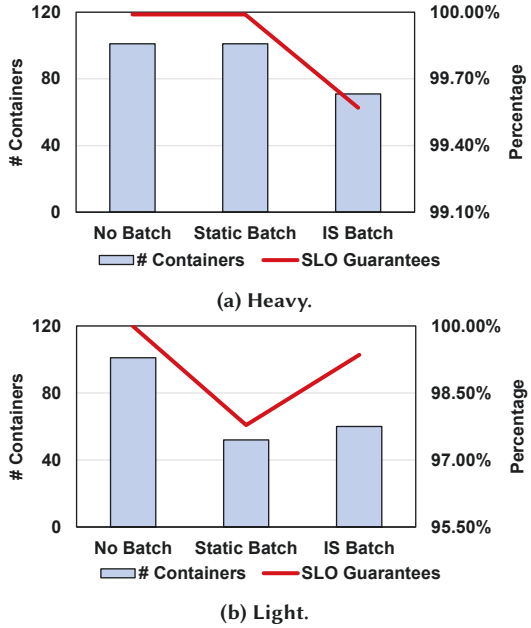


Figure 1: Number of containers spawned vs. SLOs satisfied for the *Sentiment Analysis* app under the heavy and light distributions for a Poisson trace (mean = 100 rps).

Opportunity 1: *In order to accurately provision the required number of containers, it is essential to incorporate an Input size–Sensitive container provisioning policy in the RM framework that can both predict the input size distribution and load of future requests as well as batch requests according to their input size–dependent execution times.*

To this end, we propose *Input size–Sensitive Batching* (*IS Batching*), which is a request batching policy that takes into account the input size–dependent execution time of each request and each function's SLO while performing request batching akin to the First Fit Bin Packing algorithm [45]. Here, the requests are 'packed' into containers depending on their potential execution times (estimated using IS profiling), and a new container is spawned when none of the existing containers can take an additional request without violating the function's SLO. Note that *IS Batching* needs to be performed together with input size distribution prediction to proactively spawn the requisite containers.

To explore the benefits of the above policies, we conduct an experiment on our multi-node cluster that pits a scheme, *IS Batch*, that has both input size distribution prediction and *IS Batching* against two others that use state-of-the-art policies, namely, *No Batch* (which spawns a container per incoming request, as in *Atoll* [55]) and *Static Batch* (which uses a static batch size, based on average execution time, to batch requests, as in *Kraken/Fifer* [27, 40]) (Figures 1a, 1b). Here, 'heavy' and 'light' are with regards to the relative input size of the majority of requests of the distribution. For the

heavy distribution (Figure 1a), *IS Batch* spawns 30% fewer containers compared to *No Batch* and *Static Batch*. This is because *IS Batch* spawns the requisite containers, whereas the other schemes over-provision containers, as *No Batch* does not perform batching and *Static Batch* inaccurately calculates the batch size using the function's average execution time (which skews towards the higher side here). However, partly due to spawning fewer containers, and, thereby, increasing the queuing at containers, *IS Batch* has lower SLO compliance compared to other schemes (99.57% vs. 99.99%).

In case of the light distribution (Figure 1b), *IS Batch* spawns 41% fewer containers than *No Batch*, which spawns the same number of containers as it does for the heavy distribution (being agnostic to input size distributions). Similarly, the SLO compliance of *IS Batch* is lower than that of *No Batch* (99.35% vs. 99.99%). However, since, here, the average function execution time skews towards the lower side, *Static Batch* under-provisions containers (13% fewer containers than *IS Batch*). Consequently, *IS Batch* has higher SLO compliance compared to it (99.35% vs. 97.78%).

Challenge 2: Input size–Sensitive Request Scheduling Although *IS Batch* seems to provision the requisite containers, its SLO compliance needs to be improved. Apart from allocating fewer containers, the other reason for this is that *IS Batch* is intended to spawn containers so as to meet each individual request's SLO by only considering its potential execution time and the First-Fit Bin Packing algorithm. It does not account for the queuing delays each request would face due to its relative position in the request queue. As a result, heavier requests queued behind lighter ones may potentially cause SLO violations, as they have a shorter 'buffer' time to execute (Section 1). To address this, we incorporate *Input size–Sensitive Request Reordering* (*IS Reordering*) on top of the *IS Batch* scheme to reorder the request queue to prioritize heavier requests over lighter ones (similar to the First-Fit Descending Bin Packing Algorithm [42]).

Opportunity 2: *Although the IS Batch scheme is useful for spawning the requisite containers to meet the SLOs of a large portion of requests, it is vital to perform Input size–Sensitive Request Reordering to maximize SLO compliance, while maintaining the same, minimal number of containers.*

Adding the *IS Reordering* policy to the previous *IS Batch* scheme substantially improves the SLO compliance of the resultant scheme (which we call *IS (Batch+RR)*), while spawning the same number of containers. For instance, for the previous experiment, under the light distribution, *IS (Batch+RR)* has improved SLO compliance compared to *IS Batch* (99.98% vs. 99.35%) (with equal containers) and nearly matches *No Batch*'s SLO compliance (99.99%) (using 41% fewer containers). While *Static Batch* provisions 13% fewer containers than

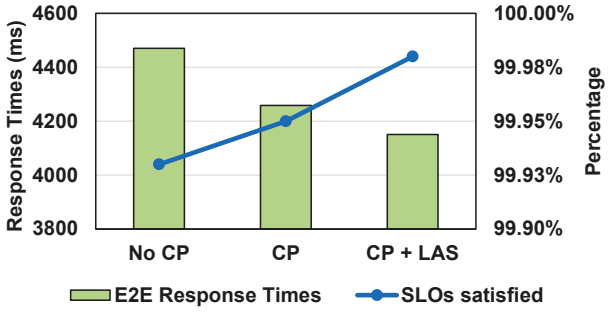


Figure 2: End-to-end response time vs. SLOs satisfied for the *Audio Translation* app for a Poisson trace (mean = 100 rps) under a uniform input size distribution.

IS (Batch+RR), *IS (Batch+RR)* greatly surpasses it in terms of SLO compliance (99.98% vs. 97.78%). [2]

Challenge 3: Multi-Function Applications While the above policies can help single-function apps remain highly SLO-compliant with the requisite containers, they may not suffice when the serverless app is composed of multiple IS functions (forming a 'function chain'). One reason is that the input size distribution for each IS function will be different since each function processes its input and produces an output, which then becomes the input for the subsequent function(s). Therefore, predicting only a particular IS function's input size distribution in a multi-function IS app is insufficient to perform effective *IS Batching* for all its functions.

A possible solution to this is, what we refer to as, *Chained Prediction*. This predicts the input size distribution of both the first IS function as well as those of subsequent IS functions. This is done by inferring the intermediate input sizes between each IS function in the function chain by using a variation of IS Profiling. Although *Chained Prediction* can help estimate the number of containers required for all functions, errors in prediction may arise, especially given that multiple predictions are done at once (for future request load as well as initial and intermediate input size distributions).

Normally, reactive scaling is used to appropriately scale containers to compensate for prediction errors in proactive scaling. However, this can potentially lead to cold starts if containers have to be scaled up. One way to mitigate this (and potentially improve SLO compliance) is to perform, what we call, *Look-Ahead Scaling* (*LA Scaling*), wherein containers are appropriately scaled in advance for all IS functions that appear later in the function chain (descendent functions) when requests arrive at the initial IS function(s). Note that *LA Scaling*, like Proactive Scaling, also uses *Chained Prediction* with *IS batching*.

To demonstrate the benefits of *Chained Prediction* and *LA Scaling*, we compare three schemes, namely, *No CP*, *CP* and

[2]The numbers for *IS (Batch+RR)* are not depicted anywhere

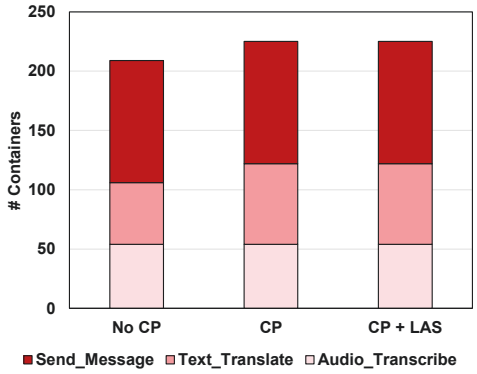


Figure 3: Breakdown of containers spawned for the *Audio Translation* app for a Poisson trace (mean = 100 rps) under the uniform input size distribution.

CP+LAS. *No CP* (No *Chained Prediction*) predicts the input size distribution for only the first IS function in the app and defaults to using static batch sizes to proactively spawn containers for subsequent IS functions. *CP* uses *Chained Prediction* to proactively spawn containers for all IS functions in the app according to their predicted input size distributions, whereas *CP+LAS* adds *LA Scaling* on top of this to provision containers for descendent functions early while requests arrive. Note that all three schemes leverage *IS Batching*.

CP improves upon both the response time (5%) and SLO compliance (99.95% vs. 99.93%) compared to *No CP* (Figure 2). This is because *CP* accurately provisions the requisite containers for all IS functions, including *Text_Translate* (which appears later in the function chain), as evidenced by Figure 3, which depicts the breakdown of containers spawned by each scheme. From Figure 2, it can also be seen that *CP+LAS* betters *CP* in terms of response time (~3%) and SLO compliance (99.98% vs. 99.95%) as it can alleviate potential cold start effects for descendent functions by accurately spawning containers for them in advance. Note that the above improvements in SLO compliance translate to a 4% and 6% improvement in tail latency for *CP* versus *No CP* and *CP+LAS* versus *CP*, respectively[3]. These improvements, as we will later see, are magnified under larger-scale request traces.

Opportunity 3: *Multi-function apps require additional policies that are cognizant of some aspects of their function chains (such as Chained Prediction and Look-Ahead Scaling) to be incorporated into the RM framework to effectively provision the requisite containers and maximize SLO compliance.*

As we will see in Section 6, all these policies synergize to help the resultant scheme outperform state-of-the-art RM frameworks in terms of resource consumption and/or SLO compliance.

[3]Not shown in any figure

3 Input size–Sensitive Profiling of Real-World Serverless Applications

As elucidated in the previous section, it is essential to incorporate IS Profiling into the RM framework as this grants it visibility into the input size–dependent function execution times, thereby, enabling intelligent container scaling and request scheduling decisions to be made. In this section, we describe the variations of IS Profiling that are employed by the framework on real-world serverless apps. These will be instrumental in the policies that will be discussed later.

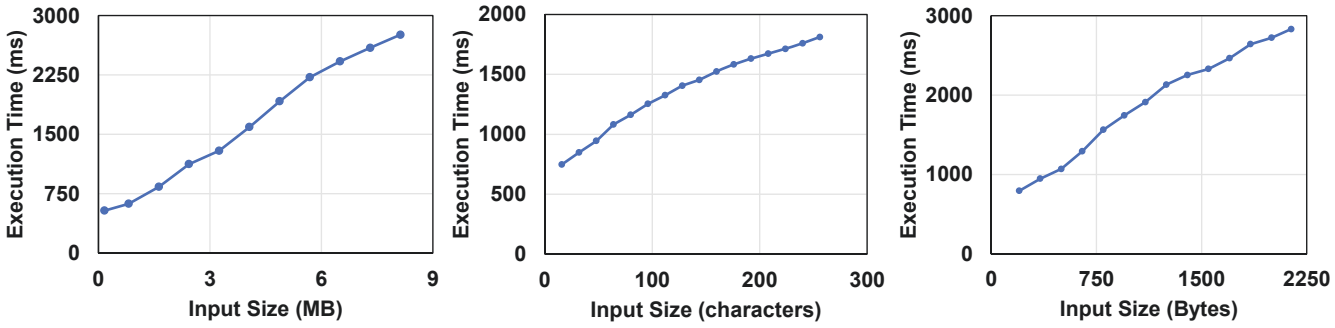
3.1 Mapping Input Size to Execution Time

One of the primary goals of integrating IS Profiling into the system is to determine a function's execution time as a mathematical function of its input size. To achieve this, online model-fitting is performed using function execution times that are sampled across time for various runs for a multitude of input sizes [1, 3, 7, 49, 53]. This yields a statistical model that estimates the function's execution time, given an input size. Figures 4, 5 depict various plots showing the relationship between input size and execution time for the IS functions for different ranges of input sizes that are typically seen for the corresponding app [1, 3, 7, 49, 53]. Note that *Send Message* is treated as a non-IS function here since its execution time is practically the same for the input sizes it receives as part of *Audio Translation*'s function chain.

Once profiling causes the model to attain sufficient accuracy, the framework automatically infers *input size classes* for each function to aid in input size distribution prediction. An input size class is a range of input sizes that results in a distinct batch size for the function, given its SLO (the class boundaries are determined using the latest IS Profiling model). Input size classes are introduced to simplify prediction and facilitate the use of a light-weight, history-based prediction model (EWMA) (overhead of $\sim 10^{-3}$ ms). Input size distribution prediction is carried out by predicting the fraction of future requests that will fall under each of the constructed input size classes.

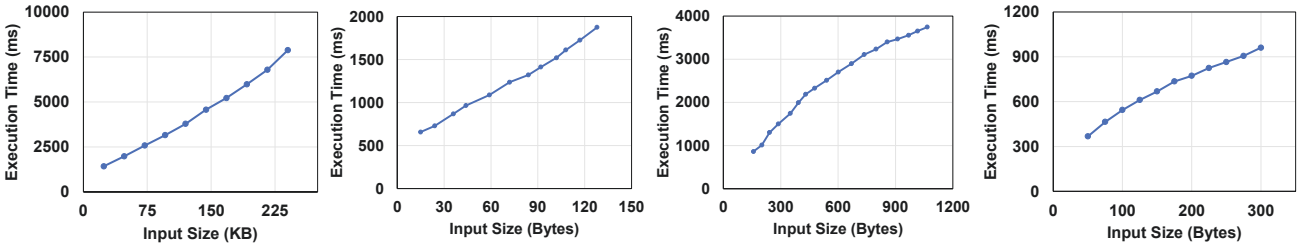
For our purposes, model-fitting for IS profiling is performed with least-squares linear regression and powerlaw regression (as in [31, 38]). Regression selects model parameters (a and b below) so as to minimize some measure of error.

Linear Models: For a set of points (x_i, y_i), least-squares linear regression constructs a model that can predict y as $\hat{y}(x) \stackrel{\text{def}}{=} a + bx$. Therefore, for a particular data point (x_i, y_i), the predicted y value, $\hat{y}_i$, is $\hat{y}_i \stackrel{\text{def}}{=} \hat{y}(x_i) = a + bx_i$, whereas the actual y value is denoted simply as y_i. The quantity $r_i \stackrel{\text{def}}{=} y_i - \hat{y}_i$ is called the *residual* of the fit at (x_i, y_i). Linear regression chooses a and b to minimize the sum of squared



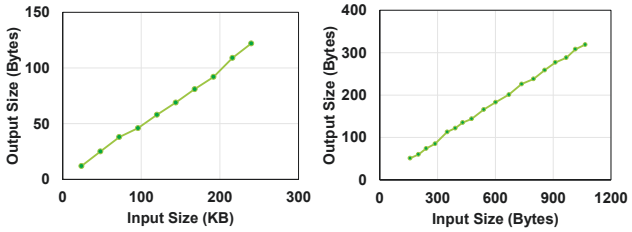
(a) Image Compression. (b) QR Code. (c) Sentiment Analysis.

Figure 4: Input size vs. execution time profiles of the single-function apps.



(a) Audio Transcribe. (b) Text Translate. (c) Text Summary. (d) Text Classifier.

Figure 5: Input size vs. execution time profiles of functions composing the multi-function apps.



(a) Audio Transcribe. (b) Text Summary.

Figure 6: Input size vs. output size profiles of the IS functions that appear earlier in the multi-function apps.

residuals:

$$\sum_{i=1}^{k} r_i^2 = \sum_{i=1}^{k} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{k} (y_i - (a + bx_i))^2$$

Powerlaw Models: A powerlaw model, on the other hand, predicts y as $\hat{y}(x) \stackrel{\text{def}}{=} ax^b$. Since the plot of a powerlaw model is a straight line on log-log axes, linear regression can be used on $(logx_i, logy_i)$ to fit observations to the model. Thus, we find a and b that minimizes the following quantity:

$$\sum_{i=1}^{k} (\log y_i - (\log a + b \log x_i))^2 = \sum_{i=1}^{k} \left(\log \frac{y_i}{ax_i^b} \right)^2$$

Since regression, by itself, does not evaluate the suitability of the model, *Cypress* utilizes the R^2 statistic to select the best-fitting model from the two. R^2 is a measure of the model's goodness-of-fit that quantifies the fraction of the variance in y accounted for by a least-squares linear regression on x [38]:

$$R^2 \stackrel{\text{def}}{=} \frac{\sum_{i=1}^{k} (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{k} (y_i - \bar{y})^2} = \frac{\left(\sum_{i=1}^{k} (x_i - \bar{x})(y_i - \bar{y}) \right)^2}{\left(\sum_{i=1}^{k} (x_i - \bar{x})^2 \right) \left(\sum_{i=1}^{k} (y_i - \bar{y})^2 \right)}$$

Note that $\bar{y}$ and $\bar{x}$ denote the sample means of k-vectors y and x, respectively. This formula also applies to powerlaw fits, but with x and y replaced by $logx$ and $logy$, respectively. R^2 can take values ranging from 0 (bad) to 1 (excellent). *Cypress* chooses the fit with the highest R^2 value as the best-fitting model for a function.

3.2 Mapping Input Size to Output Size

So far, we have discussed IS profiling in the context of predicting execution times of functions as a mathematical function of their input sizes. However, achieving this for multi-function apps is challenging as descendant IS functions will receive requests with different input sizes from those that the initial one(s) received (Challenge 3, Section 2.3). *Chained Prediction* is used to address this. It uses a variation of IS profiling wherein the input size of a function is used to predict its resultant output size. This can be used to predict the intermediate input sizes of each function in the function chain, given an initial input size. This is achieved using techniques similar to those seen earlier: the input-to-output size profiles collected for IS functions that appear earlier in the function chain (Figures 6a, 6b) are used to construct models online from which the one with the highest R^2 value is chosen to be the best fitting. In addition to Proactive Scaling,

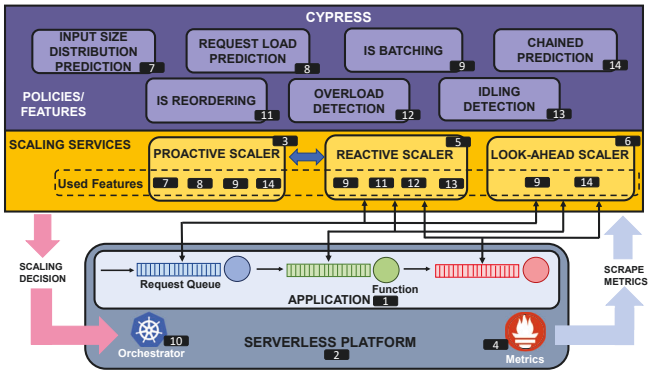


Figure 7: High-level view of *Cypress*'s design.

LA Scaling also leverages the above techniques to estimate the requisite containers for descendent functions. Note that for both variants of IS Profiling seen thus far, the best-fitting model's predicted values are within ~7% of the actual values, on average (which is sufficient for our purposes). As we will now see, incorporating IS profiling into all scaling services enables them to make intelligent scaling/scheduling decisions.

4 Overall Design of Cypress

Figure 7 outlines the overall design of *Cypress*. Users trigger the invocation of apps ❶ hosted on a Serverless platform ❷ by submitting requests to them (either in the form of HTTP requests or through actions like uploading input files to a storage service). The function execution times and corresponding input sizes are sampled to perform IS profiling and thereby, construct a statistical model online (and corresponding input size classes) to use in the various policies that will be described shortly. In *Cypress*, containers are provisioned in advance for the app's functions by the Proactive Scaler (PS) ❸ to serve the incoming requests by avoiding cold starts. The PS accomplishes this by making use of relevant metrics obtained from the serverless platform's monitoring tools/logs ❹ and *Cypress*'s other scaling service(s) ❺ / ❻. These metrics, in addition to a developer-provided app descriptor, are then used by the PS along with the input size classes inferred from IS profiling to perform Input Size Distribution Prediction ❼. This is, in turn, coupled with Request Load Prediction ❽ to estimate the number of future requests of each input size class. With this information, the PS uses *IS Batching* ❾ to calculate the required number of containers and notifies the underlying resource orchestrator ❿ to proactively spawn them.

Cypress also employs a Reactive Scaler (RS) ❺ that makes use of two key features to further reduce the app's SLO violations. Firstly, the RS performs *IS Reordering* ⓫ to prioritize heavier requests over lighter ones to improve SLO compliance. Secondly, it also uses Overload Detection ⓬ to keep track of request overloading at functions by monitoring queuing delays at containers. In case of an overload, it triggers container scaling after calculating the additional containers needed to mitigate the delay. The RS also utilizes Idling Detection ⓭ to scale down the number of containers when an excess is detected. Both Overload and Idling Detection aid in coping with the PS's potential mis-prediction of load and/or input size distribution(s).

For multi-function apps, *Cypress* employs an additional scaling service called the Look-Ahead Scaler (LAS) ❻ that uses *Chained Prediction* ⓮ to scale containers appropriately for the app's descendent functions as incoming requests arrive at the initial function(s). This performs a more accurate allocation of containers compared to proactive scaling, while also reducing cold start effects that could arise from reactive scaling. *Chained Prediction* is also leveraged by the PS to effectively spawn containers for all functions in the function chain in advance. Note that *IS Batching* permeates the container scaling process employed in the PS, RS and LAS. Below, we discuss these aspects of *Cypress*'s design in more detail.

4.1 Proactive Scaler (PS)

The PS is an integral component of *Cypress* designed to accurately provision the requisite containers in advance. As shown in Algorithm 1, this is done for each function such that enough containers will be provisioned for them at the end of fixed time windows (ⓐ in Algorithm 1). This requires not only the prediction of future request load (as in [27, 40, 55]), but also that of future input size distributions (a feature unique to *Cypress*). For this, the PS makes use of input size classes (derived from the latest IS profiling model) and a pluggable statistical model (EWMA, in our case) to predict the fraction of future requests that will belong to different input size classes based on the corresponding fractions seen in previous prediction windows. Thus, PS, in effect, performs input size distribution prediction (ⓒ in Algorithm 1). This is used in conjunction with the prediction for the total number of future requests (also using a statistical model) (ⓑ in Algorithm 1) to estimate how many of them will fall under each input

Algorithm 1 Proactive Scaling

```
1: from REACTIVE_SCALER get current_queue_details
2: for Every Monitor_Interval= PW do                    (a)
3:     Proactive_Scaler(∀func ∈ functions)
4: procedure PROACTIVE_SCALER(FUNC)
5:     cl ← Current_Load(func)
6:     pl_{t+PW} ← Load_Predictor(cl, pl_t)             (b)
7:     if multifunc == False then
8:         cisd ← Curr_Input_Size_Distr(func)
9:         pisd_{t+PW} ← Input_Size_Distr_Prediction(cisd, pisd_t)   (c)
10:    else
11:        pisd_{t+PW} ← Chained_Prediction(func)       (d)
12:    req_queue_{t+PW} ← Predict_Request_Queue(pl_{t+PW}, pisd_{t+PW})   (e)
13:    reqd_con ← Calc_Reqd_Containers(req_queue_{t+PW})   (f)
14:    Scale_Containers(func, reqd_con)
```

size class. Note that a bit of history information is allowed to accumulate (typically for a few prediction windows) before the aforementioned statistical models are used to make predictions for subsequent windows. All the above information serves as an approximation for the future request queue (e in Algorithm 1), which is used together with *IS Batching* to calculate the required number of containers to proactively spawn (f in Algorithm 1).

For multi-function apps, the PS is augmented with *Chained Prediction* (d in Algorithm 1) to help predict the input size distribution for all IS functions so as to spawn the requisite containers for them. This is achieved by additionally inferring all the intermediate input sizes between each function in the app's function chain by performing *Input-to-Output size mapping* (Section 3).

Once the required number of containers is estimated, the PS instructs the underlying orchestrator to provision them. Note that, for non-IS functions, the PS defaults to simply predicting the future number of requests alone and considers the average execution time of each of them for calculating the required containers (as in [27, 40, 55]).

4.2 Input size–Sensitive Request Batching (IS Batching)

Many serverless frameworks [9, 12, 16, 32, 55, 57, 61] spawn one container to serve each incoming request to a function. While this can help minimize SLO violations, comparable performance can be achieved with fewer containers by exploiting the notion of slack. Slack refers to the difference in expected response time (based on the function's SLO, calculated relative to the app's SLO) and actual execution time of functions within an app. This allows requests to meet their SLO deadlines, even if they incur a queuing delay. Note that this is equally applicable to single and multi-function apps. Slack is leveraged by works such as *Kraken* [27] and *Fifer* [40] to batch multiple requests to fewer containers by queuing the requests at them, thus, reducing the number of containers spawned. For this, the batch size of a function, f, is calculated as $\textit{Batch Size}\ (f) = \left\lfloor \frac{\textit{Function SLO}\ (f)}{\textit{Average Execution Time}\ (f)} \right\rfloor$.

However, as execution time is input size–dependent for IS functions, request batching using average execution times proves to be inaccurate (Section 2), as it is unaware of the considerable variation in execution times that, in turn, leads to variable slack. To account for these facts, we introduce *IS Batching*, which is a request batching policy that is cognizant of the input size–dependent execution times of IS functions. To this end, it first uses IS Profiling to map each request's associated input size to the corresponding execution time (Section 3). With this knowledge, request batching is performed as in the First-Fit Bin Packing algorithm, as alluded to in Section 2. Note that for non-IS functions, the average execution time is used to perform batching as it is independent of the input size in such cases. As mentioned earlier, *IS Batching* is used throughout all of *Cypress*'s scaling services (Figure 7) to calculate the number of required containers to serve a given request queue to an IS app.

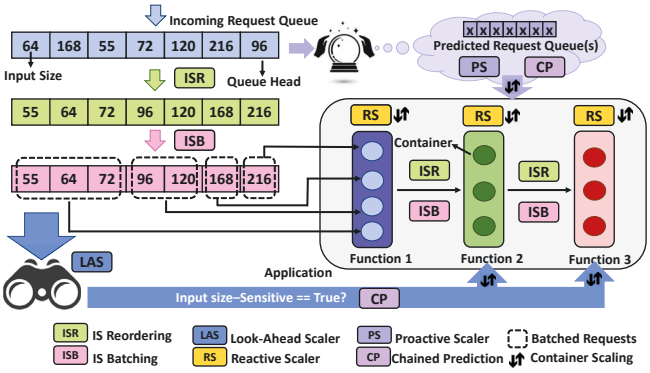


Figure 8: A schematic diagram of *Cypress*'s key components.

4.3 Reactive Scaler (RS)

Although proactive scaling and *IS Batching* can ensure that the requisite containers are spawned, further optimization is required to maximize SLO compliance (Challenge 2, Section 2.3). This is because these policies aid in spawning containers to meet each individual request's SLO by considering only its potential execution time and the First-Fit bin packing policy and not the queuing delay caused by the relative position of the request in the request queue. Furthermore, the PS may mis-predict future request loads and/or input size distributions, potentially leading to container mismanagement, thereby, adversely affecting SLO compliance.

To address these concerns, *Cypress* employs the RS to perform request queue management in addition to reactive container scaling. The RS performs *IS Reordering* to prioritize heavier requests over lighter ones in the incoming request queue (*Heaviest Job First*) (a in Algorithm 2). This ensures that the heavier requests (that have lower slack) are served earlier so as to minimize SLO violations. This resultant container allocation policy resembles the First-Fit Descending

Algorithm 2 Reactive Scaling

```
1: for Every Monitor_Interval= RW do
2:     Reactive_Scaler(∀func ∈ functions)
3: for Every Monitor_Interval= RW* do
4:     IS_Reorder(Current_Queue(func)) (a)
5: procedure REACTIVE_SCALER(FUNC)
6:     existing_con ← Current_Replicas(func)
7:     req_queue ← Current_Queue(func)
8:     if Calc_Reqd_Containers(req_queue)<existing_con then (b)
9:         reqd_con ← Calc_Required_Containers(req_queue)
10:    else
11:        delayed_req_queue ← Delayed_Requests(func) (c)
12:        extra_con ← Calc_Required_Con(delayed_req_queue)
13:        reqd_con ← existing_con + extra_con
14:    Scale_Containers(func, reqd_con)
```

Bin Packing algorithm. To deal with potential container under-provisioning, the RS leverages Overload Detection (ⓒ in Algorithm 2) to, firstly, calculate the estimated wait times of queued requests at existing containers. Then, if it detects requests whose wait times exceed the cost of spawning a new container (the cold start of the function), overloading is said to have occurred at the function. If so, *Cypress* spawns new containers to serve the delayed requests as they would get served faster this way, as opposed to waiting at an overloaded container.

Similarly, the RS uses Idling Detection (ⓑ in Algorithm 2) for functions where container over-provisioning has occurred. The RS gradually scales down its allocated containers to the appropriate number, if excess containers are detected for serving the current request load. Note that the RS incorporates *IS Batching* into all of its container scaling decisions.

4.4 Look-Ahead Scaler (LAS):

For multi-function apps, as mentioned earlier, multiple predictions have to be made to provision containers proactively: request load prediction, input size distribution prediction for the initial IS function and *Chained Prediction* for the descendent IS functions. This may lead to inappropriate container allocation, which can be rectified using the RS. However, the reactive scaling entailed may lead to SLO violations when new containers are spawned. To mitigate this, *Cypress* leverages the LAS to appropriately provision containers in advance for all descendent IS functions when requests arrive at the initial function(s). It accomplishes this by utilizing *Chained Prediction* and *IS Batching* to predict the number of containers needed for the descendent functions based on the initial request queue (ⓐ and ⓑ in Algorithm 3). Since the container scaling here is done based on the actual request queue at the initial IS function(s), as opposed to their predicted counterparts, it is more accurate. Additionally, it takes advantage of the buffer time between the requests arriving at the initial function(s) to finally reaching the concerned functions to spawn containers in advance, thereby, alleviating the potential cold start effects of reactive scaling.

Thus, *Cypress*, by leveraging its scaling services and resource management/scheduling policies (Figure 8), remains highly SLO compliant with a minimal resource footprint.

Algorithm 3 Look-Ahead Scaling

```
1: for Every Monitor_Interval= LW do
2:     LookAhead_Scaler(∀func ∈ functions)
3: procedure LOOK-AHEAD_SCALER(FUNC)
4:     if has_descendant(func) == True then
5:         cl ← Current_Load(func)
6:         for desc_func in descendants(func) do
7:             desc_isd ← Chained_Prediction(desc_func) (a)
8:             desc_req_queue ← Predict_Request_Queue(cl, desc_isd)
9:             reqd_con ← Calc_Reqd_Containers(desc_req_queue) (b)
10:            Scale_Containers(desc_func, reqd_con)
```

Policy	Implemented using/as
IS Batching	• Collected Metrics • Persisting Containers in Memory • First-Fit Descending Bin Packing Algorithm • IS Profiling (input size to execution time)
IS Reordering	Daemons to intercept and reorder incoming requests to each function
Chained Prediction	• Collected Metrics • IS Profiling (input size to output size)

Table 3: Implementation details of *Cypress*'s key policies.

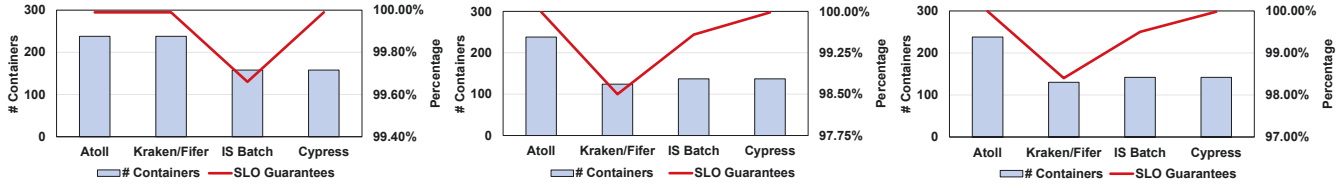
5 Implementation and Experimental Setup

Cypress is implemented using Python and Go on top of *OpenFaaS* [17], an open-source serverless platform. *OpenFaaS* is deployed with *Kubernetes* [15] as the container orchestrator. We disable *OpenFaaS*'s default Alert Manager module (that detects load surges) to deploy our scaling services (Figure 7) that leverage *Cypress*'s policies (Table 3). *Cypress* uses dynamically populated hash tables (that are looked up on demand) to map input file names to their corresponding input sizes. Note that performing the lookup and *IS Reordering* (Table 3) together incur minimal overhead (e.g. ∼1 ms per request queue of 200 requests).

Evaluation Methodology: We evaluate the *Cypress* prototype on a 6 node *Kubernetes* cluster with a dedicated manager node. Each node is equipped with 48 cores (Intel Cascade Lake), 192 GB of RAM, 1 TB of storage and a 10 Gigabit Ethernet connection [44]. For energy measurements, we use the Intel CPU Energy Meter [14] that measures the energy consumed by all sockets in a node.

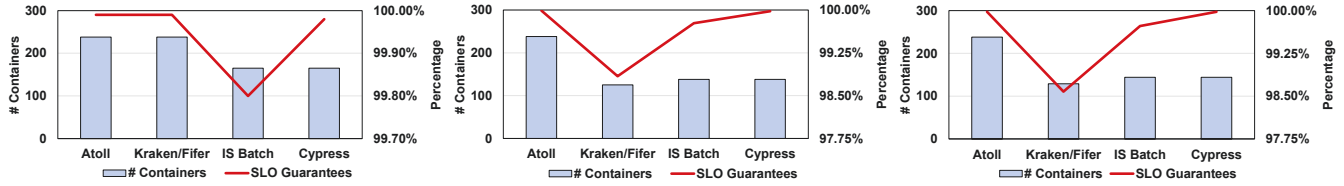
Request Traces: We use a synthetic Poisson arrival trace with an average rate $\mu = 250$ rps and real-world traces from Wiki [60] and Twitter [4]. The Twitter trace is erratic and has a large peak-to-mean ratio (5450:3139) compared to that of the Wiki trace (331:302). We choose these traces and scale their request loads based on the trend of increasing request rates evidenced by a recent study on Azure traces [65] and also envisioned in other related work [27, 55]. For most experiments, we use 3 input size distributions, namely, Heavy, Medial and Light, where input sizes are randomly generated such that the majority (50-55%) will be in the first, second or third terciles (∼33%, descending), respectively.

Applications: We implement 5 apps in *OpenFaaS* by composing one or more IS functions based on those from the AWS Serverless Application Repository [11] and/or OpenFaaS Function Store [18]. To model the characteristics of the original functions, we invoke sleep timers within our functions (as in [27]), with their durations determined by adding a *salt* term (based on the model's error) to the value given by the fitted model obtained from IS Profiling. This includes the time for reading input and state recovery (if any), as they are also billable [11]. Note that we set all app SLOs to be ∼20% higher than the app's maximum possible execution time [27].



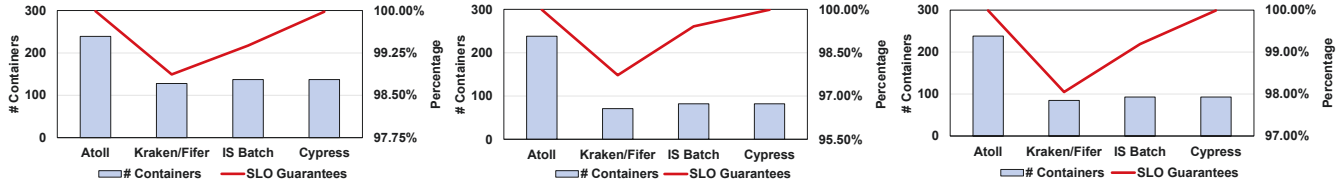
(a) Heavy. (b) Light. (c) Medial.

Figure 9: Real System: number of containers spawned vs. SLOs satisfied by each scheme for the *Sentiment Analysis* app under various input size distributions averaged across the stable traces (Wiki and Poisson).



(a) Heavy. (b) Light. (c) Medial.

Figure 10: Real System: number of containers vs. SLOs satisfied by each scheme for the *QR Code* app for the same cases as above.



(a) Heavy. (b) Light. (c) Medial.

Figure 11: Real System: number of containers vs. SLOs satisfied by each scheme for the *Image Compression* app for the same cases as above.

Resource Management schemes: We compare *Cypress* against the container provisioning schemes of *Atoll* [55], *Kraken* [27] and *Fifer* [40], which we will, henceforth, refer to as *Atoll*, *Kraken* and *Fifer*, respectively. Note that commercial providers spawn the same containers as *Atoll* [9, 12, 16], but have worse SLO compliance [27, 55] and, hence, are not presented in our experiments. Additionally, we compare *Cypress* against the scheme, *IS Batch*, which uses all policies/features of *Cypress* except *IS Reordering* and *LA Scaling*.

Large Scale Simulation: To evaluate the effectiveness of *Cypress* in large-scale systems, we built a high fidelity, multi-threaded simulator in Python using container cold start latencies and function execution times profiled from our real-system counterpart. We have validated its correctness for scaled-up versions of all traces by correlating various metrics of interest generated from experiments run on the real system (e.g. the Twitter trace was scaled up from having a peak of 250 rps in the real system to 5450 rps here). The simulator allows us to evaluate our model for a larger setup with a ~6.4k core cluster which can handle up to ~5500 requests (22× more than the real system). Additionally, it aids in comparing the resource footprint and SLO compliance of *Cypress* against that of a clairvoyant scheme (*Oracle*) that has 100% load and input prediction accuracy.

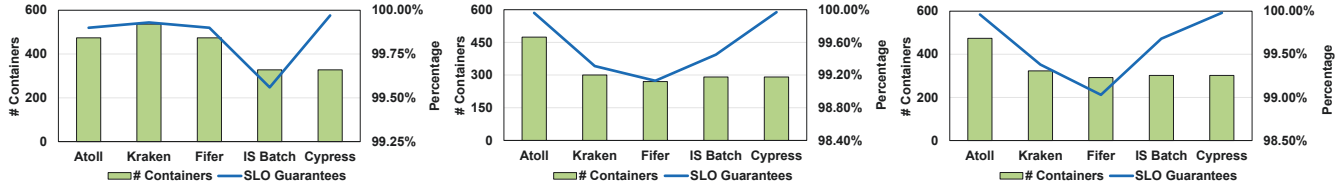
6 Results and Analysis

This section presents a thorough evaluation of *Cypress* using the real system and our simulator. Unless mentioned otherwise, most of the plots presented here pertain to either averaged or specific values for the Wiki and/or Poisson traces (which we will, henceforth, refer to as stable traces, owing to their relatively low peak-to-mean ratios) for various input size distributions. Note that, for isolated examples, similar results are seen for other apps and workload mixes, wherever applicable. Furthermore, the brick-by-brick analyses shown in Section 2.3 extend to the results presented here as well.

6.1 Real System Results

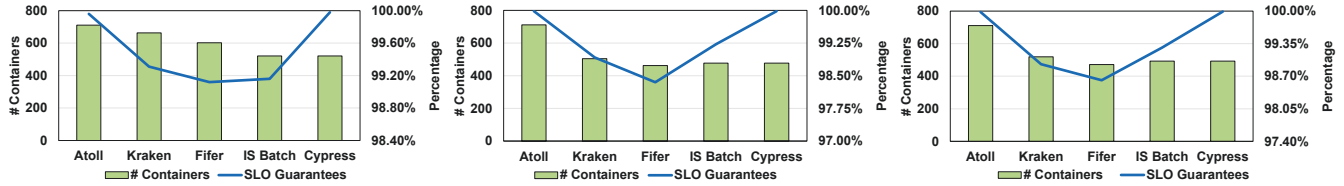
6.1.1 ***Containers Spawned vs. SLOs Satisfied*** Figures 9, 10, and 11 show the number of containers versus SLOs satisfied by all schemes for the single-function apps whereas Figures 12 and 13 represent the same for the multi-function apps.

Single-function apps: From Figures 9, 10, and 11, we observe that *Atoll* spawns the most containers in all scenarios because (i) it is agnostic to the input size variations of requests and (ii) it spawns a container per request, without performing request batching. *Cypress* spawns much fewer containers than *Atoll* while ensuring near-identical SLO compliance (within 0.01%). *Cypress* accomplishes this by using



(a) Heavy. **(b) Light.** **(c) Medial.**

Figure 12: Real System: number of containers spawned vs. SLOs satisfied by each scheme for the *Email Categorization* app for various input size distributions averaged across the stable traces (Wiki and Poisson).



(a) Heavy. **(b) Light.** **(c) Medial.**

Figure 13: Real System: number of containers vs. SLOs satisfied by each scheme for the *Audio Translation* app for the same cases as above.

IS Batching to only spawn the requisite containers while also performing *IS Reordering* to maximize SLO compliance. Heavy distributions tend to decrease the extent of request batching in *Cypress* (compared to the medial and light distributions), thereby increasing containers spawned, since the higher number of heavier requests in the trace would likely need dedicated containers to reduce app SLO violations. For instance, for *Image Compression*, *Cypress* spawns 43% fewer containers versus *Atoll* for the heavy distribution (Figure 11a), but for the light distribution, it spawns 66% fewer containers, due to batching more (Figure 11b).

Kraken and *Fifer* have the same container scaling policies for single-function apps, where they use the average function execution time to statically calculate batch sizes for request batching. Owing to this, they inappropriately allocate containers. For the heavy distributions, since the average execution time would skew towards the higher side (due to the majority of heavy requests), the calculated batch size generally becomes lower (Section 4.2). This leads to container over-provisioning since there will be lighter requests that do not need as many containers as the heavier ones. On the other hand, *Cypress*, by leveraging *IS Batching*, only spawns the requisite containers. For example, *Cypress* spawns 34% fewer containers than *Kraken*/*Fifer* for *Sentiment Analysis* for the heavy distributions, while also matching its SLO compliance using *IS Reordering* (Figure 9a).

For the light and medial distributions, *Kraken*/*Fifer* tend to under-provision containers (leading to SLO violations) due to the calculated batch size being too high because of a low average execution time. For example, although *Cypress* spawns 9% more containers than *Kraken*/*Fifer* for *Image Compression* for the medial distribution, it has 99.99% SLO compliance compared to *Kraken*/*Fifer*'s 98.05% (Figure 11c). *Cypress* spawns the same number of containers as *IS Batch*, since both schemes use *IS Batching*. Despite this, *Cypress* outperforms *IS Batch* in terms of SLO compliance in all cases as it uses *IS Reordering*.

Multi-function apps: For the same reasons as with the single-function apps, *Atoll* is seen to spawn the most containers in almost all scenarios, with *Cypress* provisioning much fewer containers than it (Figures 12, 13). For example, *Cypress* spawns 33% fewer containers than *Atoll* for the *Audio Translation* app for the light distribution (Figure 13b). Moreover, *Cypress* has even better (if not, as good) SLO compliance compared to not only *Atoll*, but all other schemes, for multi-function apps, with a minimum of 99.97%. This is because the other 'input size–agnostic' RM frameworks are adversely affected by the presence of multiple IS functions in the app, as this exacerbates the effects of queued heavy requests on SLO violations (Challenge 2, Section 2.3).

Fifer is seen to suffer from the same effects of using static batch sizes as in single-function apps, leading to inappropriate container provisioning. For example, it over-provisions containers for *Email Categorization* for the heavy distribution, with *Cypress* spawning 30% fewer containers than it (Figure 12a). For the light distribution, although *Cypress* spawns 8% more containers than *Fifer*, it has an SLO compliance of 99.97% versus *Fifer*'s 99.13% (Figure 12b). *Kraken* has a similar container provisioning strategy to *Fifer*, with the exception that it provisions extra containers for functions considered important, depending on their position in the function chain. As a result, *Cypress* is observed to spawn fewer containers than it, while also having higher SLO compliance in all cases. For instance, *Cypress* allocates 21% fewer containers than *Kraken* for *Audio Translation* for the heavy distribution while having an SLO compliance of 99.98% versus *Kraken*'s 99.31% (Figure 13a). Compared to *IS Batch*, similar trends to those of single-function apps can be observed

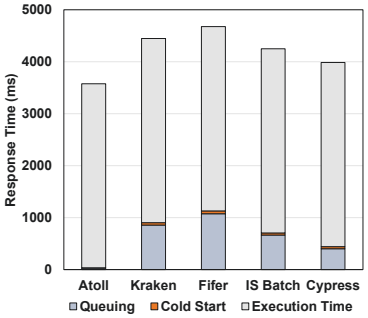


Figure 14: Real System: end-to-end response time breakdown for the *Audio Translation* app for the Poisson trace under the light distribution.

here as well. Note that *Cypress* leverages *IS Reordering* and *LA Scaling* to remain highly SLO compliant.

6.1.2 ***End-to-End Response Times and Latency Distribution*** While we observed that *Cypress* has near-identical SLO compliance to *Atoll*, it also has slightly higher execution times (4% on average) than the latter for both single-function and multi-function apps. This is primarily due to *Cypress* spawning much fewer containers than *Atoll* (40% on average), thereby, causing queuing, in turn, increasing the queuing delay of requests. For example, the relatively higher queuing time of *Cypress* versus *Atoll* leads to an 11% increase in its response time for *Audio Translation* (Figure 14). However, in this case, *Cypress* spawns 32% fewer containers than *Atoll*. Moreover, as we will shortly see, *Cypress* remains within the SLO for all workload mixes for all functions at the tail (P99). These benefits more than compensate for the slightly higher execution times versus *Atoll*.

It can be observed from Figure 14 that *Cypress* achieves 15%, 10% and 6% less execution time than *Fifer*, *Kraken*, and *IS Batch*, respectively. Although the difference in the number of containers is just within 6% compared to these schemes, *Cypress* achieves its lower execution time primarily by virtue of its *IS Reordering*, *LA Scaling* and *IS Batching*. From Figure 15, we observe that *Atoll*, like *Cypress*, meets the SLO at P99. However, it does this using 70% more containers than *Cypress*. Compared to *Kraken*/*Fifer*, although *Cypress* spawns 11% more containers, it remains SLO-compliant as opposed to those schemes, that exceed the SLO by 50% at P99. This is reflected in the SLO compliance of *Kraken*/*Fifer*, which is only 98.19% in this case [4]. Note that, since *IS Reordering* prioritizes heavier requests, the fastest executing requests of *Cypress* are slowed down slightly in exchange for a much lower tail latency. This ensures that no isolated request suffers from abysmal response times (unlike in *Kraken*/*Fifer* and *IS Batch*).

6.1.3 ***Analysis of other Key Benefits*** This subsection discusses a few other key benefits offered by *Cypress*.

[4]Note that not all numbers specified here have been shown in the graphs.

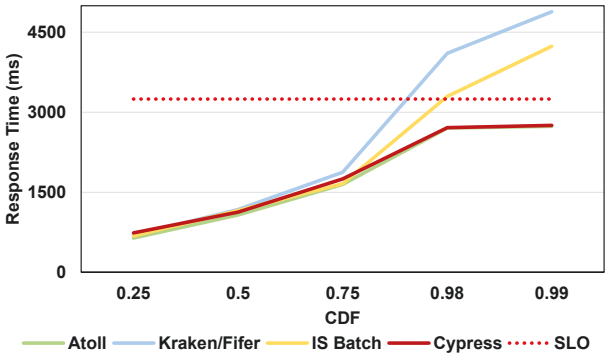
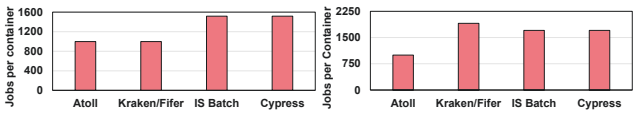


Figure 15: Real System: response time distribution for the *Image Compression* app for the Poisson trace under the heavy distribution.



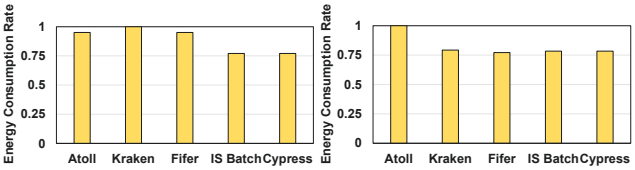
(a) Heavy. **(b) Light.**

Figure 16: Real System: container utilization of the *Sentiment Analysis* app for the Wiki trace under the heavy and light distributions.

Container Utilization: Here, we define container utilization as the jobs (requests served) per container. Hence, a scheme should ideally have high container utilization (by packing more jobs onto fewer containers) with minimal SLO violations. As per Figure 16, *Cypress* has 52% and 71% more container utilization compared to *Atoll* for the heavy and light distributions, respectively. This is because *Cypress* performs *IS Batching*, thereby, provisioning only the requisite containers as opposed to *Atoll*, that does not batch requests (and thus, over-provisions containers). For the heavy distribution, *Kraken/Fifer* use a static batch size of 1, thereby, behaving similarly to *Atoll*. However, for the light distribution, they have 11% more container utilization than *Cypress* since they under-provision containers as a result of inaccurate batching. In this case, *Cypress* has much better SLO compliance than them (99.98% versus 99.32%) and meets the SLO at P99, which they do not. Although, *IS Batch* has the same utilization as *Cypress* (due to provisioning the same number of containers), it has worse SLO compliance for both the heavy (99.82% versus 99.99%) and light (99.80% versus 99.98%) distributions than *Cypress* as a result of not having *Cypress*'s additional policies. [5]

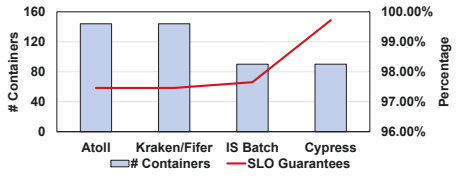
Energy Efficiency: We measure the energy consumption as the total energy consumed by a scheme divided by total time. *Cypress* consumes 19% and 22% less energy than *Atoll* for the heavy and light distributions, respectively (Figures 17a and 17b). The energy savings of *Cypress* are a direct consequence of the savings in compute and memory used by the fewer containers it spawns. For the same reason, *Cypress*

[5] Not all numbers have been shown in the Figures/Tables

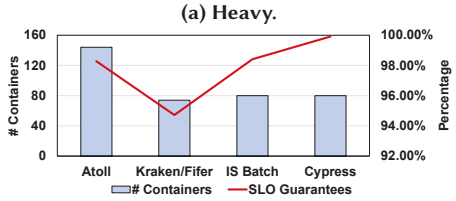


(a) Heavy. (b) Light.

Figure 17: Real System: normalized energy consumption rate of *Email Categorization* for the Poisson trace under the heavy and light distributions.



(a) Heavy.



(b) Light.

Figure 18: Real System: number of containers spawned vs. SLOs satisfied for the *Sentiment Analysis* app under the heavy and light distributions for the erratic trace (Twitter).

consumes 23% and 19% less energy than *Kraken* and *Fifer*, respectively, for the heavy distribution (Figure 17a). For the light distribution, *Cypress* consumes 1% less and 1% more energy than *Kraken* and *Fifer*, respectively (Figure 17b). This implies that the extra containers spawned by *Cypress* versus *Fifer* (9% in this case) translates to a much lower difference in energy consumption. Note that despite the relatively low difference in energy consumption between *Cypress*, *Kraken*, and *Fifer*, *Cypress* outperforms the others in terms of SLO compliance (99.96% versus 98.79% and 98.56%) and minimizing tail latency (the others exceed the SLO by 30% at the tail) [5]. Similarly, *IS Batch*, while consuming the same amount of energy, also has worse SLO compliance and tail latency, regardless of distribution.

Resilience to Erratic traces: As noted earlier (Section 5), the Twitter trace is highly erratic, with a high peak-to-mean ratio (~2:1) and therefore, can adversely affect the SLO compliance of various schemes. For the heavy distribution, it is observed that *Cypress* outperforms the other schemes, *both* in terms of containers spawned (34% lesser than *Atoll*, *Kraken/Fifer* and equal to *IS Batch*) and SLOs satisfied (99.72% versus 97.46% of *Atoll*, *Kraken/Fifer* and *IS Batch*'s 97.65%) (Figure 18a). This is because, despite the erratic request load, *Cypress*'s features such as input size distribution prediction, *IS Batching* and *IS Reordering* combined with its load prediction enable it to tolerate the influx of various input sizes that accompanies the variable load. For the light distribution (Figure 18b), almost all schemes improve their SLO compliance (including *Cypress*, with the highest, 99.91%), which may be due to the request trace inherently having a majority of lighter requests (which have more execution slack). To highlight the effects of the other schemes' low SLO compliance on their tail latencies, we can refer to Table 4 that shows all P99 values (in ms) corresponding to Figure 18. For both the heavy and light distributions, *Cypress* is the *only* scheme to remain SLO compliant at the tail, with *Atoll*, *Kraken/Fifer* and *IS Batch* violating the SLO at P99 by as much as 57%, 89% and 55%, respectively.

Distribution	Scheme				SLO
	Atoll	*Kraken/Fifer*	*IS Batch*	*Cypress*	
Heavy	5256	5260	5189	2989	3340
Light	5059	6262	5022	2889	

Table 4: Real System: P99 values (in ms) of all schemes for *Sentiment Analysis* under the heavy and light distributions for the erratic trace.

6.2 Simulator Results

Seeing that the real system implementation is limited to a 288 core cluster, we use our in-house simulator, that can simulate a 6.4k core cluster, to study the scalability of *Cypress*. To this end, we also use large-scale versions of the Poisson trace (μ = 1000 rps), Wiki (μ = 302 rps) and Twitter (μ = 3139 rps) traces. The simulator results are observed to closely correlate to those of the real system (Table 5). Generally, for the heavy distribution, *Cypress* spawns fewer containers than all other schemes (up to 43% fewer) while being highly SLO compliant (up to 99.99%). For the light and medial distributions, *Cypress* spawns even fewer containers, especially compared to *Atoll* (up to 65% fewer). While *Cypress* does spawn more containers than *Kraken/Fifer* in similar scenarios, it has superior SLO compliance and tail latency compared to them. For example, although *Cypress* spawns 15% more containers than *Kraken/Fifer* for *Image Compression* for the light distribution for the stable trace, it has an SLO compliance of 99.98% versus *Kraken/Fifer*'s 99.17%. Moreover, *Cypress* remains within the SLO at P99, whereas *Kraken/Fifer* violates it by 30% for the same case[6]. *IS Batch*, as expected, spawns the same number of containers as *Cypress* on average, but is outperformed by it in terms of SLO compliance and tail latency as well. For similar reasons as in the real system, *Cypress* outperforms the other schemes in both metrics, for multi-function apps, on many occasions. For instance, *Cypress* spawns 20% fewer containers than *Kraken* and has higher SLO compliance (99.99% vs. 99.81%) for *Audio Translation* for the stable trace under the heavy distribution. *Cypress* is also more resilient to the erratic trace than other schemes, as in the real system.

6.2.1 ***Sensitivity Study*** This subsection compares *Cypress* against *Oracle*, an ideal policy assumed to be able to predict

[6]Not shown in Table

App	Scheme	Heavy		Light		Medial	
		Stable	Erratic	Stable	Erratic	Stable	Erratic
Sentiment Analysis	*Atoll*	(304, 99.99%)	(3141, 97.40%)	(304, 99.99%)	(3141, 98.22%)	(304, 99.99%)	(3140, 97.55%)
	Kraken/Fifer	(304, 99.99%)	(3140, 97.40%)	(160, 99.28%)	(1617, 94.62%)	(169, 99.32%)	(1708, 95.40%)
	IS Batch	(211, 99.78%)	(2078, 97.61%)	(184, 99.75%)	(1847, 98.36%)	(189, 99.79%)	(1945, 97.40%)
	Cypress	**(211, 99.99%)**	**(2078, 99.68%)**	**(184, 99.98%)**	**(1847, 99.89%)**	**(189, 99.99%)**	**(1945, 99.77%)**
QR Code	*Atoll*	(304, 99.99%)	(3141, 99.43%)	(304, 99.99%)	(3141, 99.70%)	(304, 99.99%)	(3140, 99.58%)
	Kraken/Fifer	(304, 99.99%)	(3141, 99.43%)	(161, 99.24%)	(1691, 96.38%)	(171, 99.36%)	(1738, 95.83%)
	IS Batch	(223, 99.95%)	(2206, 98.50%)	(184, 99.88%)	(1960, 99.00%)	(192, 99.82%)	(1993, 98.36%)
	Cypress	**(223, 99.99%)**	**(2206, 99.71%)**	**(184, 99.95%)**	**(1960, 99.91%)**	**(192, 99.99%)**	**(1993, 99.90%)**
Image Compression	*Atoll*	(304, 99.99%)	(3141, 98.50%)	(304, 99.99%)	(3141, 99.20%)	(304, 99.99%)	(3140, 98.81%)
	Kraken/Fifer	(165, 99.45%)	(1673, 95.23%)	(96, 99.17%)	(966, 92.78%)	(112, 99.38%)	(1198, 95.35%)
	IS Batch	(173, 99.70%)	(1845, 97.98%)	(110, 99.70%)	(1130, 98.20%)	(129, 99.80%)	(1381, 98.01%)
	Cypress	**(173, 99.98%)**	**(1845, 99.90%)**	**(110, 99.98%)**	**(1130, 99.95%)**	**(129, 99.98%)**	**(1381, 99.93%)**
Email Categorization	*Atoll*	(608, 99.97%)	(6280, 99.12%)	(608, 99.99%)	(6280, 99.80%)	(608, 99.99%)	(6280, 99.62%)
	Kraken	(688, 99.98%)	(7095, 99.23%)	(419, 99.83%)	(4173, 98.47%)	(447, 99.86%)	(4767, 98.49%)
	Fifer	(608, 99.97%)	(6280, 99.12%)	(374, 99.71%)	(3861, 98.30%)	(406, 99.37%)	(4160, 98.29%)
	IS Batch	(455, 99.78%)	(4721, 98.65%)	(408, 99.88%)	(4082, 98.84%)	(430, 99.92%)	(4455, 98.88%)
	Cypress	**(455, 99.98%)**	**(4721, 99.89%)**	**(408, 99.99%)**	**(4082, 99.93%)**	**(430, 99.99%)**	**(4455, 99.95%)**
Audio Translation	*Atoll*	(912, 99.99%)	(9423, 99.83%)	(912, 99.98%)	(9423, 99.88%)	(912, 99.99%)	(9423, 99.84%)
	Kraken	(848, 99.81%)	(8668, 98.33%)	(652, 99.56%)	(6719, 97.80%)	(691, 99.59%)	(7030, 97.82%)
	Fifer	(777, 99.69%)	(7955, 97.96%)	(604, 98.84%)	(6116, 97.58%)	(627, 99.38%)	(6416, 97.57%)
	IS Batch	(678, 99.70%)	(6938, 98.11%)	(629, 99.69%)	(6412, 98.29%)	(653, 99.71%)	(6733, 98.47%)
	Cypress	**(678, 99.99%)**	**(6938, 99.93%)**	**(629, 99.99%)**	**(6412, 99.96%)**	**(653, 99.99%)**	**(6733, 99.96%)**

Table 5: Simulator: (# containers spawned, SLOs satisfied) for all schemes for all apps under three input size distributions for a stable (Wiki) and erratic (Twitter) trace.

App	Heavy		Light		Medial	
	Cypress	*Oracle*	*Cypress*	*Oracle*	*Cypress*	*Oracle*
IC	(875, 99.95%)	(831, *C*)	(531, 99.97%)	(515, *C*)	(637, 99.96%)	(617, *C*)
QR	(1015, 99.88%)	(923, *C*)	(925, 99.95%)	(869, *C*)	(940, 99.95%)	(893, *C*)
SA	(999, 99.88%)	(939, *C*)	(876, 99.95%)	(840, *C*)	(913, 99.93%)	(876, *C*)

IC: ***Image Compression*** *QR*: ***QR Code*** *SA*: ***Sentiment Analysis*** *C*: **100%**

Table 6: Simulator: comparison of (# containers spawned, SLO compliance) of *Cypress* and *Oracle* averaged across all traces.

future request load as well as the exact pattern of future input sizes to all functions with 100% accuracy. Note that apart from being clairvoyant, *Oracle* uses the same policies as *Cypress*. By comparing against *Oracle*, we intend to isolate the effects of *Cypress*'s load/input size mis-prediction(s) on its performance for individual functions and hence, perform the experiments for the single-function apps. Generally, *Cypress* is seen to be conservative in dealing with load/input pattern prediction errors by slightly over-provisioning containers, via reactive scaling, in response to them. From Table 6, we can see that *Cypress* over-provisions containers by 4%, 7%, and 5% on average for the *Image Compression*, *QR Code*, and *Sentiment Analysis* apps, respectively, in comparison to *Oracle*. Although *Oracle* outperforms *Cypress* in terms of SLOs satisfied owing to its clairvoyance, *Cypress* remains within 0.12% of its SLO compliance, even during its least performant scenarios. This can be attributed to the synergy of its various policies compensating for its prediction errors.

7 Concluding Remarks

Adopting the serverless platform for input size–sensitive apps introduces critical request scheduling and resource management challenges for the cloud provider. To address these, we design, implement, and evaluate *Cypress*, an input size–sensitive serverless resource management framework, for efficiently running such apps with minimal resources, while remaining highly SLO-compliant.

To this end, *Cypress* employs various scaling services that leverage its policies/features, that include *Input size–Sensitive Request Batching*, *Input size–Sensitive Request Reordering*, and *Chained Prediction*, among others. Our experimental evaluation on a 288 core cluster using 5 apps from the AWS Serverless App Repository and the OpenFaaS Function Store with real-world traces and various input size distributions demonstrates that *Cypress* spawns up to 66% fewer containers, thereby improving container utilization and cluster-wide energy savings by up to 2.95× and 23%, respectively, compared to state-of-the-art serverless resource management frameworks.

8 Acknowledgement

We are indebted to our anonymous reviewers and shepherd, Lei Zhang, for their insightful comments. This research was partially supported by NSF grants #1931531, #1955815, #1763681, #2116962, #2122155 and #2028929. We also thank the NSF Chameleon Cloud project CH-819640 for their generous compute grant. All product names used here are for identification purposes only and may be trademarks of their respective companies.

References

[1] 2020. Email Length Best Practices. https://www.campaignmonitor.com/blog/email-marketing/email-length-best-practices-for-email-marketers-and-email-newbies/.

[2] 2020. Establishing Effective SLOs. https://www.datadoghq.com/blog/establishing-service-level-objectives/.

[3] 2020. Findings about Search Engine Optimization. https://backlinko.com/search-engine-ranking.

[4] 2020. Twitter Stream traces. https://archive.org/details/twitterstream. Accessed: 2020-05-07.

[5] 2021. AWS Lambda Cold Starts. https://mikhail.io/serverless/coldstarts/aws/.

[6] 2021. Azure Functions Cold Starts. https://mikhail.io/serverless/coldstarts/azure/.

[7] 2021. Image Compression with Lambda. https://dev.to/aarongarvey/size-matters-image-compression-with-lambda-and-s3-40bf.

[8] 2022. Amazon Automated Chat Moderation. https://aws.amazon.com/blogs/business-productivity/automated-moderation-and-sentiment-analysis-with-amazon-chime-sdk-messaging/.

[9] 2022. AWS Lambda. https://aws.amazon.com/lambda/.

[10] 2022. AWS Lambda for Sentiment Analysis. https://aws.amazon.com/blogs/compute/using-aws-lambda-and-amazon-comprehend-for-sentiment-analysis/.

[11] 2022. AWS Serverless Application Repository. https://aws.amazon.com/serverless/serverlessrepo/.

[12] 2022. Google Cloud Functions. https://cloud.google.com/functions/docs/.

[13] 2022. HappyScribe: Online Audio Transaltion Service. https://www.happyscribe.com/.

[14] 2022. Intel CPU Energy Meter. https://github.com/sosy-lab/cpu-energy-meter.

[15] 2022. Kubernetes. https://kubernetes.io/.

[16] 2022. Microsoft Azure Serverless Functions. https://azure.microsoft.com/en-us/services/functions/.

[17] 2022. OpenFaaS. https://www.openfaas.com/.

[18] 2022. OpenFaaS Function Store. https://github.com/openfaas/store.

[19] 2022. Provisioned Concurrency. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[20] 2022. Sentiment Analysis of Social Media. https://docs.aws.amazon.com/whitepapers/latest/big-data-analytics-options/example-3-sentiment-analysis-of-social-media.html.

[21] 2022. Serverless Speech to Text. https://awscloudfeed.com/whats-new/videos/serverless-speech-to-text-using-s3-and-lambda.

[22] 2022. The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[23] 2022. Translating documents with Amazon Translate and AWS Lambda. https://aws.amazon.com/blogs/machine-learning/translating-documents-with-amazon-translate-aws-lambda-and-the-new-batch-translate-api/.

[24] Istemi Ekin Akkus et al. 2018. SAND: Towards High-Performance Serverless Computing. In *ATC*.

[25] A. Alourani, M.A.N. Bikas, and M. Grechanik. 2016. Input-Sensitive Profiling: A Survey. Advances in Computers, Vol. 103. Elsevier, 31–52. https://doi.org/10.1016/bs.adcom.2016.04.002

[26] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 195–212. https://www.usenix.org/conference/osdi18/presentation/berger

[27] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3472883.3486992

[28] Marc Brooker, Andreea Florescu, Diana-Maria Popa, Rolf Neugebauer, Alexandru Agache, Alexandra Iordache, Anthony Liguori, and Phil Piwonka. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.

[29] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[30] Joao 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 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[31] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-Sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/2254064.2254076

[32] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.

[33] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (feb 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[35] 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* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/3358960.3379133

[36] 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)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[37] 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[38] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 395–404. https://doi.org/10.1145/1287624.1287681

[39] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. 2019. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 199–208. https://doi.org/10.1109/CLOUD.2019.00043

[40] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Proceedings of the 21st International Middleware Conference*. 280–295.

[41] Ling Huang, Jinzhu Jia, Bin Yu, Byung-gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *Advances in Neural Information Processing Systems (NeurIPS)*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta (Eds.), Vol. 23. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2010/file/995665640dc319973d3173a74a03860c-Paper.pdf

[42] David S Johnson. 1973. *Near-Optimal Bin Packing Algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[43] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *EuroSys*.

[44] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[45] Bernhard Korte and Jens Vygen. 2018. Bin-Packing. In *Combinatorial Optimization*. Springer, 489–507.

[46] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2020. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 207–219. https://doi.org/10.1109/HPCA47549.2020.00026

[47] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[48] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*.

[49] Ruth Rettie and Lisa Chittenden. 2003. *Email marketing: Success factors*. Kingston Business School, Kingston University.

[50] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (apr 2021), 76–84. https://doi.org/10.1145/3406011

[51] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[52] Aakash Sharma, Saravanan Dhakshinamurthy, George Kesidis, and Chita R. Das. 2021. CASH: A Credit Aware Scheduling for Public Cloud Platforms. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 227–236. https://doi.org/10.1109/CCGrid51090.2021.00032

[53] Christy Sich. 2017. A Comparison of Traditional Book Reviews and Amazon.com Book Reviews of Fiction Using a Content Analysis Approach. *Evidence Based Library and Information Practice* 12, 1 (Mar. 2017), 85–96. https://doi.org/10.18438/B8CW4N

[54] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference*. 1–13.

[55] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[56] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 570–584. https://doi.org/10.1145/3472883.3486979

[57] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 311–327.

[58] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2017. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 977–987. https://doi.org/10.1109/ICDCS.2017.262

[59] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2019. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–13. https://doi.org/10.1109/CLUSTER.2019.8891040

[60] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2009. Wikipedia workload analysis for decentralized hosting. *Computer Networks* (2009).

[61] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *ATC*.

[62] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 329–341. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing

[63] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. 2017. PowerChief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained CMP. In *Computer Architecture News*.

[64] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference*. 173–186.

[65] 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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580