Resource Allocation in Serverless Query Processing

Simon Kassing, Ingo Müller, Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich
Zürich, Switzerland
{simon.kassing, ingo.mueller, alonso}@inf.ethz.ch

ABSTRACT
Data lakes hold a growing amount of cold data that is infrequently accessed, yet require interactive response times. Serverless functions are seen as a way to address this use case since they offer an appealing alternative to maintaining (and paying for) a fixed infrastructure. Recent research has analyzed the potential of serverless for data processing. In this paper, we expand on such work by looking into the question of serverless resource allocation to data processing tasks (number and size of the functions). We formulate a general model to roughly estimate completion time and financial cost, which we apply to augment an existing serverless data processing system with an advisory tool that automatically identifies configurations striking a good balance—which we define as being close to the “knee” of their Pareto frontier. The model takes into account key aspects of serverless: start-up, computation, network transfers, and overhead as a function of the input sizes and intermediate result exchanges. Using (micro)benchmarks and parts of TPC-H, we show that this advisor is capable of pinpointing configurations desirable to the user. Moreover, we identify and discuss several aspects of data processing on serverless affecting efficiency. By using an automated tool to configure the resources, the barrier to using serverless for data processing is lowered and the narrow window where it is cost effective can be expanded by using a more optimal allocation instead of having to over-provision the design.

1 INTRODUCTION
Serverless computing, now a feature offered by all major cloud providers [8, 20, 36], has been touted as a major step forward in terms of more efficient computing [13, 43]. In spite of the limitations of serverless platforms [25, 26, 50], recent results indicate that analytical query processing over serverless is feasible but that it has a very narrow window where it remains cost effective. The work on Starling [40] and Lambada [38] show that, in current commercial offerings, the maximum throughput is a few tens of queries per hour. Beyond that, it is better to use a fixed infrastructure as it costs less per computation time unit. Nevertheless, these systems show that query processing over serverless can be competitive for interactive, occasional use. For instance, it can be an useful extension to data lakes such as Databrick’s Lakehouse [16], Microsoft Azure’s Data Lake [35], or Amazon’s Redshift [3]. Over large swaths of the data in these large repositories, analytical queries are run only infrequently resulting into what is referred to as “cold” data. Loading such data into traditional databases for processing is costly and impractical. Using a data lake infrastructure is more economical but still incurs too high costs for only occasional use. This is where serverless query processing can be very useful. Compared with using virtual machines, serverless functions have much lower latency to start, are more scalable, and are charged at a smaller time granularity (e.g., 1 ms [7]).

Provisioning resources in the cloud for query processing is not easy [30]. For serverless, the narrow effective window and the limited choices given to the user make the task even more difficult [50]. However, for occasional use over the cold data, such provisioning is critical to keep the system cost-effective. The two obvious extremes yield little practical value to the user: the cheapest choice (small number of the smallest functions) finishes too slow; the fastest (a large number of the largest functions) is too expensive. Our goal in this paper is to develop an advisory tool that can recommend suitable configurations for serverless query processing striking a good balance between completion time and financial cost.

To do this, we propose a model similar to [30]’s approach for VMs, that considers the major factors affecting completion time and cost in serverless query data processing. It includes processing and network overheads (as [30], except using data and rate rather than CPU hours for processing), and as well accounts for startup time, exchange overhead, and request cost. The latter three factors are relevant for serverless data processing due to (a) the large number of workers involved, (b) the brevity of interactive queries, and (c) communication through storage.

We use this model to augment an existing query data processing system, Lambada [38], with an advisory tool to provision serverless functions: given a query workload, the tool approximates the number of functions and their size to achieve a good balance between query completion time and the cost incurred. It does so by choosing configurations close to the “knee” of the Pareto frontier of these two dimensions. We show that obtaining accurate predictions of the run time or cost on serverless is very difficult due to a variety of system aspects that cannot be controlled by the user [26, 50]. However, we also show that it is still possible to recommend a reasonable configuration, thereby significantly simplifying the job of the user when using serverless for query processing over data lakes and expanding the narrow cost effectiveness of serverless query processing.

2 WHY SERVERLESS DATA PROCESSING?
Serverless has been proposed as an alternative to using virtual machines to process interactive queries on cold data [38, 40]. Cold data is so infrequently accessed that it is not cost-effective or practical to keep infrastructure on hot stand-by for processing it. Yet, users still expect interactive response times which requires sufficient computing power. For other query processing tasks, serverless might be too expensive. There are two reasons for this:

Billing method. Virtual machine instances are billed both at coarser time granularity and with a minimum time across all cloud providers.
4 ESTIMATION CHALLENGES

To recommend a suitable configuration, we build a model (§5) encompassing key characteristics of a serverless query execution system. The model does not try to make an accurate prediction of running time or cost but to differentiate between configurations. Estimating running time and cost in serverless is actually quite difficult (§8, §9) due to the many uncertainties associated with serverless execution [45, 50]. In this section, we highlight both the fundamental prediction limitations and practical challenges inherent to serverless data processing to define the problem space.

4.1 Data-processing-related limitations

At the core of query optimization is the need to predict the cost of an operation, for which often metadata and statistics on the data (selectivity, distribution, etc.) are used. It is an open question how to do this over the vast amounts of data stored in a data lake. There have been considerable efforts, using zone maps [15] or gradual transformations of the data [31], as well as table formats to include cost-based optimization metrics [12]. Lack of such data processing knowledge results in the first source of potential inaccuracies when trying to guess the running time of a query. In our case, these are the two factors that introduce the largest distortions:

- **Query selectivity.** The selectivity of an operator determines the amount of data the following operators process. As a result, the cost of intermediate operators in the query plan is difficult to gauge and can only be done on average or as a worst case.
- **Data distribution.** The performance of an operator (i.e., process rate) is affected by the data distribution of the input. For instance, the population of internal data structures can result in longer processing time in aggregation operators relying on hash tables. If many input tuple keys are unique, the hash table will become full, causing collisions, resizing overhead, no longer fitting in caches, etc. This makes it difficult to estimate the cost of such operators even when they are on the leaves of the query plan. Moreover, the size of each individual tuple can vary; e.g., 100 MiB of 10 B tuples compared to 100 MiB of 100 B tuples would result in 10× more operations.

We can account for these variances in two ways: (a) bounding the data size and effect on complexity, or (b) with additional knowledge about the data. In this project we opt for the former to avoid relying on functionality that is typically not available in data lakes. Note that adding such information (e.g., as suggested by [31]) to the model will only make it more accurate.

4.2 Serverless platforms related limitations

Serverless query processing systems make use of the many services a cloud provider offers, in particular, storage and serverless functions. The experienced quality of service, however, can vary: (a) the way the implementation and resource allocation of cloud services are opaque to their users, and (b) the resources behind these services typically are not reserved: they are shared among users and, thus, service quality can vary. The following are side effects observed in Amazon AWS influencing run time and cost:

- **Start-up time variance (starting stragglers).** The time from invocation until the worker is ready to start the computation is
sometimes longer than usual, already creating starting stragglers. We hypothesize this can be an effect of the caching strategy, as well as temporarily contention due to high demand of the shared infrastructure.

- **Storage read/write variance (storage stragglers).** Serverless query processing systems make use of long-term storage (e.g., S3 [9]) to read input and potentially write output, as well as to exchange data in-between computational stages. Infrequently, the read or write requests to S3 take considerably longer than expected (by as much as one order of magnitude). This can be due to a variety of reasons: e.g., S3 replication, S3 caching, network packets being lost, or high demand (on either the function host or S3). Even if it occurs only 1% of the time, if there are 100 workers, these tail failure probabilities become significant [17, 40] and create large variations on the running time.

- **Queueing service send/receive variance.** Queuing services (e.g., SQS) can be used to exchange small messages between workers, or to the driver, for example if the query result is small enough. Infrequently, but especially with many simultaneous finishes, the workers will simultaneously send their result message to the SQS queue. This can lead to significant delay if the underlying application/transport stack starts timing out. This is a known problem [2] in parallel processing when multiple parallel workers finish at the same time and all try to communicate back with a centralized server.

- **Computational contention.** A worker is allocated computational power proportional to its memory allocation. With limited computational power, in particular, less than one vCPU, even a single-threaded system combined with background computation can exhibit unpredictable computation patterns that can affect the observed network bandwidth and, thus, the time needed to read/write data.

- **Non-linear network performance.** In line with previous work [51], we observe that lambda functions in AWS can exhibit an increased network rate for a very short period of time at the beginning of their execution. Whether this behavior is seen or not depends on whether the storage (S3) can supply data at this increased rate and whether the function has sufficient parallelism available to deal with the higher incoming data rate (both in terms of vCPU and the need to download multiples files). In practice, it greatly affects the ability to estimate how long a query will take.

- **Polling requests.** When awaiting for a file from another worker to become available, a worker will poll to check when it is available. The more frequently the worker polls, the faster it is made aware and can start reading; however, this affects the price since, the more read requests, the higher the cost of the computation. The duration of waiting (and thus polling) is influenced by stragglers.

Because of these intrinsic difficulties, in this paper we do not aim to build a query cost predictor. Our model does accurately estimate running time and cost in many situations but not always. However, the estimations are good enough to be able to differentiate between configurations in terms of picking suitable ones in terms of balancing running time and cost. This is the basis for the advisory tool we propose.

## 5 GENERAL ESTIMATION MODEL

Rough estimations of cost/execution time on virtual machines have been recently proposed [30]. However, such an approach does not directly translate to serverless functions due to: (a) the difference in the number of workers involved (10s for virtual machines, 100s for serverless functions), (b) the overhead incurred for data exchanges (typically through storage), and (c) the difference in task duration. Serverless operates in the few to tens of seconds range, whereas virtual machines in minutes to hours – factors insignificant at a one hour scale become so at a one second scale [45, 50]. In this section, we formulate an analytical model which incorporates the most important factors in our understanding that affect the completion time and financial cost of query processing using serverless functions. The model includes processing and network overhead similar to [30] (using data and rate instead of CPU hours for processing), but also incorporates the factors unique to serverless query processing systems, namely startup time, exchange overhead and request cost. The model we present is directly applicable to Lambda [38], and we posit models in a similar vein can be applied to other systems to roughly estimate performance albeit with changes, for instance to account for different number of workers per stage (e.g., for [29, 40]).

### 5.1 Start-up

The largest serverless function is equivalent to at most a medium-tiered virtual machine instance (10 GiB of memory with 5.8 vCPUs for AWS). Thus, many serverless functions must be spawned to be able to process large amounts of data. A single driver to start the workers can only start workers at a limited rate. To speed up the start up phase, it has been proposed to use a two-level broadcast tree where the first workers start other workers [38]. We model the starting of serverless workers by an invocation rate $R_{1-inv}$ and $R_{2-inv}$ (i.e., how many workers can be started per second) for the main driver and a worker respectively. We define the invocation delay as $T_{inv}$ delay (i.e., how long from invocation till the worker is started). For each worker $i \in 1..W$, we estimate its startup and ready time.

**One-level invocation:**

$$T_{\text{startup}}(i) = \frac{i}{R_{1-inv}} + T_{\text{inv. delay}}$$

$$T_{\text{ready}}(i) = T_{\text{startup}}(i)$$

**Two-level invocation:**

$$T_{\text{startup}}(i) = \begin{cases} \frac{i}{R_{1-inv}} + T_{\text{inv. delay}} & i \leq \sqrt{W} \\ \frac{i}{g} \frac{1}{g} + T_{\text{inv. delay}} + \frac{h}{R_{2-inv}} + T_{\text{inv. delay}} & \text{else} \end{cases}$$

$$T_{\text{ready}}(i) = \begin{cases} T_{\text{startup}}(i) + \frac{\sqrt{W}}{R_{2-inv}} & i \leq \sqrt{W} \\ T_{\text{startup}}(i) & \text{else} \end{cases}$$

$g$ is the index of the worker that starts $i$ and $h$ the index within worker $g$’s start list (first $\sqrt{W}$ workers invoke $\frac{W-\sqrt{W}}{\sqrt{W}}$ others each).

The startup time indicates when the worker has started, and ready time when the worker can start its data processing tasks (both since query epoch). This is different for some workers in the two-level
5.2 Base overhead

Every worker performs a certain amount of base tasks that are independent of the workload, including initializing its variables, extracting input, allocating memory, and packaging its output. We model this (billed) overhead time as a single constant $T_{base}$.

5.3 Process, compress, and network rates

We model the process, compress, and network rate as three constants (dependent on the memory size chosen): $R_{network}$, $R_{compress}$, and $R_{process}$. We consider network and compress rate as independent of the workload, whereas the process rate as dependent. These constants can be attained through measurement (as is done in §7.3), by applying some model (e.g., using Amdahl’s law) to account for the effect of more vCPU, or a combination of the two. Network rate is the rate at which both transfer of read and written data takes place.

5.4 Input

The input is solely concerned with the reading and processing of the input:

$$T_{input} = \max\left(\frac{D_{input}}{R_{network}}, \frac{D_{input}}{R_{process}}\right)$$

5.5 Exchange(s)

The purpose of an exchange operator is to partition data such that each function receives all tuples with the same key. An exchange can be done in one or more levels, which for two or more levels means that first functions exchange in a group, and then having the groups exchange (and so forth depending on number of levels). An exchange operator is used to accomplish join and reduce operations, among others. Its operation consists of compressing and writing the data, and subsequently reading and processing the data destined to the function. The number of the group size in an exchange level is defined as $G_{ex[j]}$, a value we calculate based on the total number of workers and whether an exchange is one or two levels. We define an additional penalty factor for each worker participating in the exchange level, $T_{overhead[j]}$: For each exchange level $j$:

$$T_{ex[j]} = \frac{D_{ex[j]}}{R_{compress}} + \frac{D_{ex[j]}}{R_{network}} + \max(\frac{D_{ex[j]}}{R_{network}}, \frac{D_{ex[j]}}{R_{process}}) + G_{ex[j]} \times T_{overhead[j]}$$

5.6 Output

The output is solely concerned with the amount of data to compress and subsequently written to storage:

$$T_{output} = \frac{D_{output}}{R_{compress}} + \frac{D_{output}}{R_{network}}$$

5.7 Postprocessing

The driver must receive the final results from the workers and process them to return a final result to the user. We model this as a constant: $T_{postprocess}$.

5.8 Requests costs

Unlike virtual machines, which almost exclusively communicate directly between each other, most serverless data processing systems communicate through storage services (e.g., S3 [38, 40]). In general, cloud providers charge for the access to the storage service. In exchanges between many participants, the number of requests can become a significant cost factor. When applying the advisory tool to a system, we assume we can give a reasonable estimate of the request cost $C_{requests}$.

5.9 Overall Completion Time and Cost

The completion time is defined as the last worker completing and its result having been collected by the driver:

$$T_{completion} = T_{ready(W)} + T_{base} + T_{input} + \sum_j T_{ex[j]} + T_{output} + T_{postprocess}$$

The billable time of each worker does not directly equal the completion time. Instead, it equals the sum of time the worker was alive:

$$T_{billable} = \sum_i T_{alive}(i)$$

The alive (i.e., billable) time of a worker is only independent from other workers if there no exchanges as else it has to pace with the slowest worker. We define the worker alive time of a simple scan (without exchanges) as:

$$T_{alive}(i) = T_{ready}(i) - T_{startup(i)} + T_{base} + T_{input} + T_{output}$$

Similarly, if there are exchanges, the workers have to wait until the last worker is ready. Hence, we define the worker alive time for a query with exchanges as:

$$T_{alive}(i) = T_{ready}(W) - T_{startup(i)} + T_{base} + T_{input} + \sum_j T_{ex[j]} + T_{output}$$

The final cost is the sum of the worker runtime cost and the sum of the requests cost:

$$C_{total} = T_{billable} \times price/ms + C_{requests}$$

5.10 Model analysis

The general model has several parameters, each of which effects increase in completion time and/or financial cost, the latter either through increased billed serverless time or requests cost. The general tendencies are as follows. Firstly, increasing the number of workers ($W$) can have the following effects:

- increased completion time as more workers must be invoked and exchange overhead is larger;
- decreased completion time as the amount of data each worker reads, writes and exchange reduces;
- increased cost through billed time as there is base overhead and invocation;
- and increased cost through request cost as there are more exchange members.

The above effects are under the assumption of invocation and exchange levels remaining the same. Increasing the worker memory size ($M$) can have the following effects:
The choice of configuration is a multi-objective optimization of completion time and cost, and as such can be visualized as a two-dimensional Pareto plot (Fig. 1). We consider as suitable configurations those closest to the "knee" of the Pareto frontier. We automate the picking of this "knee" using the following method.

Let \( F \) be the set of candidate configurations. For each candidate configuration \( i \in F \), we define its completion time as \( t_i \) and its cost as \( c_i \). We select as optimum the configuration with the shortest distance to the origin, with its completion time and cost normalized by the fastest and cheapest possible outcomes across all configurations:

\[
\begin{align*}
    t_{\text{fastest}} &= \min_{i \in F} t_i \\
    c_{\text{cheapest}} &= \min_{i \in F} c_i \\
    d(i) &= \sqrt{\alpha \left( \frac{t_i}{t_{\text{fastest}}} \right)^2 + \beta \left( \frac{c_i}{c_{\text{cheapest}}} \right)^2} \\
    \text{knee-opt} &= \arg \min_{i \in F} d_i
\end{align*}
\]

The selection of this point is depicted visually in Fig. 1. In our work, we set the weights \( \alpha \) and \( \beta \) to one, which means that we value completion time and cost equally. For example, a configuration with \( 2 \times t_{\text{fastest}} \) and \( 1 \times c_{\text{cheapest}} \) has the same score as a configuration with \( 1.6 \times t_{\text{fastest}} \) and \( 1.6 \times c_{\text{cheapest}} \) approximately.\(^1\) A related work on multi-objective optimization for cloud data analytics proposed a similar method, but instead uses as metric the Euclidian distance directly to the idealized fastest and cheapest point (the so-called "Utopia point") [46] (further discussed in §10). We chose to use the normalized distance to the origin such that the resulting distance metric is relatively interpretable and comparable.

### 6.2 Advisory tool and evaluation

The advisory tool uses the procedure above to select a configuration. As it does not know the actual completion times and costs, it instead uses the values supplied by the estimation model.

As an example, consider the following example. Suppose the best configuration has \( t = 1.1 \times t_{\text{fastest}} \) and \( c = 1.3 \times c_{\text{cheapest}} \), whereas the advised configuration outcome has \( t = 1.4 \times t_{\text{fastest}} \) and \( c = 1.2 \times c_{\text{cheapest}} \). The distances of the best and the advised configurations 1.70 and 1.84, respectively. The aforementioned metrics are in this case the following: the advised configuration is overall \( \frac{1.70}{1.70} = 100\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice and it runs \( \frac{1.84 - 1.70}{1.70} = 8\% \) worse than the experimentally determined best choice.

### 7 APPLYING THE MODEL: LAMBADA

To test the tool, we use it on Lambada [38]. We first describe Lambada (§7.1), and then explain the experiments conducted (§7.2). We apply the general model by filling in the Lambada/AWS-specific parameters (§7.3) and finally analyze the effect of the filled values in terms of overarching configuration preference (§7.4).

#### 7.1 Description

Lambada [38] takes as user input a computational DAG of operators, from which it devises a distributed query execution plan. The driver starts the selected amount and type of serverless workers (i.e., AWS Lambda) and passes on the (distributed) plan to each worker. If many workers have to be started (if \( W > 100 \)), the driver makes use of "two-level invocation": workers started first are also
in charge of starting other workers. Once a worker is ready, it starts executing the plan: each worker reads its data partitions from cloud storage (i.e., AWS S3). The partitioned tables are stored in Parquet format, to take advantage of compression and the ability to only download the columns relevant to the query. Each worker then goes through the same computations (albeit on different data), and amongst them perform exchanges when data should be shared (e.g., join, aggregation). Exchanges take place through S3: each worker writes to a file the data to send to other workers. The exchanges can either be one-level (“all-to-all”) or two-level (if number of workers \( W > 32 \)). Two-level exchanges double the amount to read and write, while quadratically reducing the number of read and write requests (which is what the cloud provider charges for). Once done, a workers returns its result by putting them in a shared result queue which the driver monitors. The driver collects all results, performs some last operations, and returns the result to the user.

7.2 Instrumentation

To capture runtime and cost information, we have augmented Lambada with the following instrumentation:

- **Query completion time.** The completion time is measured by the driver from the moment it starts the workers until it has returned the final result to the user.
- **Query cost.** Each worker keeps track of the time from when it started until it returns the final result to the SQS queue, which constitutes the amount of billable time. Each worker counts the number of HEAD, GET, and PUT requests and logs them.

We use worker logs to keep track of request counts but also for post-mortem analysis: to know how long certain operations and functions within the workers take. This is vital to identify bottlenecks and the occurrence of stragglers. The logs are not used by the system itself, and as such the upload (by the workers) and download (by the driver) of worker logs are excluded from both by the system itself, and as such the upload (by the workers) and download (by the driver). We perform a simple benchmark in which each worker retrieves a single small file. We subtract the scan duration from the total time the worker is deducted from its alive time. The result, its mean across three repetitions, is made monotonic, and is shown together with the measurements in Fig. 5.

7.3 Applied model

In this section, we fill in the parameters of the general model (§5) except the data sizes, which we fill in on a per-query basis in the experiments (§8, §9).

**Invocation.** The rate and delay at which a system can invoke workers depends on the invocation implementation and the cloud. We use 128 concurrent threads for invocation on the driver, and 32 on a worker. We perform a microbenchmark in which we start 512 workers using only the driver or only the worker, under either COLD start\(^2\) and HOT start. Tab. 2 shows the mean invocation rates and the mean 99th percentile invocation delay across three repetitions. The later experiments of this paper all are HOT started, in order to have the many runs finish in feasible time (as re-deployment to achieve coldness takes long and is rate-limited by AWS).

**Base overhead.** We perform a simple benchmark in which each worker retrieves a single small file. We subtract the scan duration from the total time the worker is deducted from its alive time. The result, its mean across three repetitions, is made monotonic, and is shown together with the measurements in Fig. 5.

**Compress rate.** Before writing data to storage, it must first be compressed into Parquet format. We assume the compression rate is independent of the type of underlying data. We perform an exchange microbenchmark with each of 32 workers reading and compressing 32 MiB with two columns. The experiment outcome is shown in Fig. 4(top). The mean compression rate (for all 96 samples across three repetitions) made monotonic is used in the model, depicted in Fig. 4(bot). It scales proportionally with vCPU allocation (Fig. 3) (single-core bound).

**Network rate.** Unlike VMs [30], there is no network bandwidth info available for serverless functions (see Tab. 1). Although it is observed that there is an initial higher network burst rate [38, 51], we model network rate at its sustained state. In the microbenchmark, each worker reads in a 128 MiB file with eight columns. The mean network (for all 96 samples across three repetitions) made monotonic is used in the model, depicted in Fig. 4(bot). Note that

---

2By re-deploying the function beforehand to ensure it is not cached.
3By performing another run with at least as many workers beforehand, which warms up the AWS lambda caches.
especially network rates can have outliers, we have observed very low rates every now and then (see minima in Fig. 4(top)). Lambada has a small multi-threaded behavior in which, if it has two partitions to read in, it does so simultaneously. If it has sufficient compute available to it (at least more than 768 MiB), it can achieve (temporarily) a network rate higher than the model predicts. We consider these conditions too system-specific and as such are not captured in the general model (in this microbenchmark, there is only a single input partition per worker).

Requests and exchange overhead. Lambada fetches the exact columns it needs from each Parquet file. With the knowledge of tables/columns involved for each input scan and exchange, we can calculate the exact number of GET requests. Due to availability polling during exchanges, the number of HEAD requests can vary between runs. An exchange microbenchmark yielded approximately 3 HEAD requests for each exchange file. We use the number of requests and their duration to determine the exchange overhead.

Microbenchmark measurements indicate a HEAD and a GET request take approximately 13.1 ms and 18.4 ms on average, though there were values of up to 215 ms (HEAD) and 530 ms (GET). The exchange overhead per group member $T_{\text{overhead}[j]}$ is set to the multiplication of the requests to read its exchange file and the request type duration. From the AWS documentation, we directly retrieve pricing for serverless function time (0.0000166667 $ per GiB-second) [7], and read and write requests (respectively 0.0004 $ and 0.005 $ per 1000 requests) [10].

Post-processing. We have 128 workers each fetch a file of 32 MiB, and measure the duration between the last worker having returned its result and the driver finishing. Tab. 2 shows the mean duration across 10 repetitions.

Processing rate. The process rate of a query is the most difficult parameter to estimate, as it is query dependent. As an admittedly imperfect solution, we use the reduction to 1 million unique values as the process operation; its internal data structure, the hash table, is used across many common operations such as joins, reductions, and group-by’s. The results are shown in Fig. 4(top). Note that Lambada’s operator execution model is single-threaded and only marginally benefits from multiple vCPUs: it roughly scales proportionally with vCPU allocation (Fig. 3) up until single core.

7.4 Applied model analysis

The general model applied to Lambada ($\S$7.3) preserves the general model influences ($\S5.10$) by monotonically modeling the rates and overhead. Its modeled sustained network rate is capped out at 82.1 MiB/s, with 78.7 MiB/s being already reached at $M = 1024$
8 INTO PRACTICE

In this section, we describe the manner how to apply the model and the practical guidelines in terms of choosing the correct configuration that it implies. We first describe in §8.1 how we apply the evaluation methodology of the advisor (§6) in our benchmarks. In the context of serverless data processing tasks, we characterize workloads into two categories: (1) workloads that solely scan data (§8.2), and (2) workloads that involve one or more exchanges (§8.3). Either category has its own trend in the sweet spot in its choice of number of workers and memory size. This section lays the groundwork for the large scale benchmark in §9.

8.1 Methodology

The goal of our work is to characterize serverless data processing performance in general, which implies some version of ideal circumstances. As such, we want to evaluate the model’s rough estimation and advice using an ideal system. In particular, this means we do not wish to compare to runs that are unfortunate enough to encounter stragglers (§4), which are in particular present for network rates and requests (§7.3, §7.4). We similarly can only perform a limited number of repeated runs with a Cartesian product of memory sizes and number of workers. In order to eliminate outliers, we opt to choose the run best suited to be a kneepoint. We choose this by first performing the distance metric using all data points, and for each configuration choose the run which had the smallest distance metric. The chosen configuration is then the experimental best and used as a reference for error calculation. As such, each point in any plotted actual Pareto frontier (e.g., Fig. 6b) is not the mean across all repetitions, but the best run among them. Note that although we do choose the representative data point for each configuration to be the best, the normalization is done with the cheapest and fastest outcome across all repetitions. This is done because the representative data point selection is to account for outliers, whereas normalization should be done with the true lowest values. The theory as presented in §6.2 is upheld, as the fastest and cheapest value can only be lower by considering all repetitions.

It is possible that the advisor recommends a configuration which does not have an actual experimental outcome (i.e., due to running out of memory or hitting the timeout limit), as it is not always aware of this. In this case, which is exceedingly unlikely occur, the advisor error would be set to infinite – though, in none of our experiments this was the case as these are generally the extremes.

8.2 Scan workloads

In a scan workload, there is no communication between workers. Workers read in the input partitions assigned to them, scan through their assigned data, and return their result directly to storage or the driver in case of simple aggregation. The typical use case for this workload type is to select tuples that fit certain criteria, or very simple aggregations whose return value is a few tuples (e.g., counting, summing, reduction to very few tuples). The associated computational requirement is as such not much. The primary bottleneck is their reading of input over the network.

Based on the applied model analysis (§7.4), around a memory size of 768-1024 MiB there starts to be diminishing returns on network rate when increasing further. Because the workers do not exchange data in a scan, there is no strong penalty on the increase of workers to large numbers. Only the relatively minor base overhead and invocation increase the billed runtime, and only the latter to a small extent the completion time (as more workers must be started).

To study the scan workload, we perform the following experiment. We make use of a 32 GiB table of eight 64-bit columns filled with random values. It is partitioned into 256 Parquet files of 128 MiB each. The query counts the number of rows (which is $2^{29}$). As this operation is computationally simple, we set the model $R_{\text{process}}$ to a value larger than $R_{\text{network}}$; which means it is of no consequence. As only the large number of workers are of interest, we vary the number of workers $W \in \{86, 128, 256\}$ (corresponding to 3, 2, and 1 partitions/worker respectively). We vary the memory size at steps between 384 MiB and 4096 MiB. The configurations ($W=86/128, M=384$) did not make the 20 second start-up time limit

$M=384$  
$M=512$  
$M=768$  
$M=1024$  
$M=1280$  
$M=1536$  
$M=1769$  
$M=2048$  
$M=4096$  

$W=86$  
$W=128$  
$W=256$  

Figure 6: Scan benchmark: Pareto frontiers with the red lines pinpointing their kneepoint (estimated = actual: $W=256, M=768$).
and as such did not finish. Decreasing the number of workers further results in too long completion time, and decreasing the memory size further leads to out-of-memory (i.e., infeasible) and too slow completion. Both of which are not part of the regime of interest, which is the knee of the curve: achieving decent performance at a cost-efficient price point. Each configuration run is repeated five times.

The estimated Pareto frontier is depicted in Fig. 6a: the maximum number of workers \( W = 256 \) with a modest memory size of \( M = 768 \) is advised to achieve both good completion time and cost. This advice matches the experimentally determined best choice, as is seen in the actual Pareto frontier in Fig. 6b.

Note that the estimation does not always match the actual outcome. For \( W = 256 \) the completion time and cost estimate is slightly lower: this is due to regular variance of network rate. For \( W = 128 \) and \( W = 86 \) the estimates are consistently too high in completion time and cost for memory sizes over 768 MiB: this is because \( W = 128 \) and \( W = 86 \) have more than one input partition, such that Lambda can download two input partitions simultaneously. With its dual download and sufficient compute to it, it can make use of the temporary burst rate serverless functions on AWS have available to it [38]. Our model asserts only a sustained rate, which purposefully does not capture such system-specific intricacies.

These results show that scan workloads are generally network-intensive, not compute-intensive (unless the scan involves a computationally heavy operation e.g., a reduction-by-key), and do not include exchanges. Their optimal configuration is to make use of many workers with a memory size that provides high network bandwidth.

### 8.3 Exchange Workloads

In an exchange workload, workers read in their respective input data, and then proceed to perform one or more exchanges to facilitate certain global operations such as joining, grouping, or sorting. The associated computation can be considerably intensive, and its complexity can depend on the data (§4.1). An exchange workload has to not only read input data and write output data, but also read and write for each exchange. As such, it is network-intensive from a rate perspective. Besides rate limits, there is also a cost to an increase in exchange members: there is a limit to the amount of outstanding requests and ongoing connections. This is particularly important for Lambda, which communicates through the writing and reading of files to storage (a practice not unique in serverless data processing systems [40]) as inter-lambda communication is not (easily [51]) possible. For every exchange member, a worker has to read in all columns individually.

Following the applied model (§7.3), we expect the memory size with near-highest network rate, process rate, and compress rate to be the best pick: around \( M = 1769 \) appears to be the best. We expect a number of workers that balances for completion time the decreasing influence of more workers processing the amount of data (i.e., rate) and the increasing influence of the extra overhead caused by more exchange members. Another factor we must consider is out-of-memory: hash tables can grow to large sizes with many unique keys for instance.

We examine the exchange workload through a benchmark. We use a 4 GiB table of one column filled with 64-bit random values, partitioned into 256 Parquet files of 16 MiB each. The query is to count the number of unique rows (which is in expectation \( 2^{25} \)) – which requires a reduction operation. This is a rather compute-intensive operation: every tuple processed will be inserted into the hash table that facilitates the reduction. We take the 1M-reduction (\( \approx 2^{20} \)) processing rate from Fig. 4, although the operation will likely be slower. This is one of the fundamental limitations of estimating arbitrary compute: it is as arbitrarily difficult. We vary the memory size between 1024 and 4096 MiB in several steps. We choose a number of workers in the mid-range, with 1 to 5 partitions/worker: \( W \in \{54, 64, 90, 132, 256\} \). Reducing memory size or number of workers further results in out-of-memory cases in the Cartesian product. The configurations \( W=54/64, M=1024 \) already ran out-of-memory: hash tables can grow to large sizes with many unique keys for instance.

Figure 7: Exchange benchmark: Pareto frontiers with the red lines pinpointing their knee points (estimated: \( W=64, M=1769 \), actual: \( W=90, M=1769 \)).
recommend the \((W = 64, M = 1769)\) configuration as best, whereas the experimentally determined best choice actually was the \((W = 90, M = 1769)\) configuration. If the user would have followed the advice (as in practice, the user would not explore the space), there would have been a 32% increase in completion time with a -9% cost reduction. With additional prior information (e.g., if the query is run repeatedly), this discrepancy could be adjusted and the estimation (and as such, the advice) improved. For example, by simply reducing the model process rate by one-third for all memory sizes in Fig. 4, the advisor would elongate along the x-axis (completion time) and correctly pick the best configuration.

Unlike for scanning, an increase in workers leads also to an increase in the number of requests \((O(W \sqrt{W}) [38])\) and exchange overhead. These two factors cause the curve to go up with the number of workers: for instance, at \((W = 64, M = 1769)\) a mere 15% of cost is requests, whereas at \((W = 256, M = 1769)\) it is 38%. Note that the benchmark exchange has a large amount of exchanged data (its entire input). In other exchanges which exchange little data, throughput is likely to be less of a bottleneck, whereas the exchange overhead can dominate the completion time.

In exchange workloads, the optimal configuration makes use of a number of workers that balances total rate and the overhead/cost of the enlarged exchange. Exchange workloads are generally both network- and compute-intensive, and as such favors a memory size which offers both at reasonable cost.

### 9 LARGE BENCHMARK

A version of the TPC-H benchmark [47] that runs on Lambada is used to explore how well the estimation performs for representative workloads [38]. Here we expect to see larger errors in estimating running time and cost but what matters is how close the suggested configuration is to the optimal.

#### 9.1 Dataset generation

For each scale factor (SF), we performed the following steps to prepare the dataset for consumption by lambda workers. (a) We generated the raw text CSV files using the Lambada [38] version of dbgen [48]. (b) We use LINEITEM as the basis to decide how to split all tables as it is the largest table. At SF=1, the size of LINEITEM is approximately 586.2 MiB uncompressed. After compression into a Parquet file, it is 193.3 MiB. We set the divisor to achieve a size of \pm 100 MiB per LINEITEM partition. We split each single large CSV file (of the eight tables) into the smaller files. (c) We convert each raw text CSV to Parquet format using Apache Arrow [11]. The final result for a subset of the scale factors is depicted in Tab. 3. (d) The files are stored on AWS S3. Note that in query execution, only the relevant columns are read: a worker retrieving a partition thus only retrieves a fraction of the total size of each file. We did not sort the CSV files before splitting, so the min/max column metadata in the Parquet files does not allow reads to be skipped.

#### 9.2 Query modelling

For each query, the model workload parameters must be determined to facilitate the rough estimation. The selectivity in the data flow we manually determine by loading the benchmark data (at SF=1) into a relational database, and manually executing selection queries corresponding to each phase (input, exchange(s), output). For example for Q4, approximately 63.2% of the LINEITEM tuples and 3.8% of ORDERS tuples pass the input selection criteria. Q19 of TPC-H was altered using the correct REG AIR instead of AIR REG. The compression ability of Parquet varies for each column, and the workers only read in the columns relevant to the query. To determine the tuple data size of a read, we determine the mean entry size of each column by inspecting the Parquet files. For each table, we inspect the first partition at SF=100, which yields us the total byte size of each column and the total number of records. We obtain the size of a column by simply dividing the two. Using this method, we determined column \_extendedprice of LINEITEM (size: SF \times 6M tuples) has on average 5.26 byte per tuple. This is of use during estimation: for example in a scenario of SF=200 with 384 partitions this means retrieving the column from a single partition will result in 200 \times 6M \times 5.26/384 = 15.7 MiB the model expects to be retrieved. We use the 1M-reduction process rate from §7.3 for all except Q6, which is the only query of the six which does not have a join or reduce-by-key operation. For Q6, we set the model process rate to a value higher than any possible network rate, which means it is of no consequence as the minimum of the two rates is taken to calculate input duration in the model (§5.4).

#### 9.3 Experimental setup

We evaluate across four scale factors \(SF \in \{20, 100, 200, 500\}\) (see Tab. 3) for six queries (Q1, Q4, Q6, Q12, Q14, Q19). For each \((SF, Q)\) combination, we perform runs across 49 configurations: the Cartesian product of seven number of workers and seven memory sizes (768, 1024, 1280, 1769, 2048, 2560, 4096). The seven numbers of workers are determined via the targeted number of input partitions \((F)\) per worker: \(F \in \{1, 2, 3, 4, 8, 16, 32\}\). For example, at \(SF = 100\) with 192 partitions, the number of workers is \(W \in \{192, 192, 192, 192, 192, 192, 192\}\) = \{6, 12, 24, 48, 64, 96, 192\}. There is one exception for \(SF = 500\), at which we do not run \(F = 1\) (W=960) as it is costly and clearly not in the regime of interest. Any configuration for which any of its repeated runs results in out-of-memory or reached time limit are considered infeasible (26 out of 1134, thus
we have 1108 total configurations). Each configuration is run three times, and the best data point is chosen in accordance with §8.1.

### 9.4 Results

The main aspect to explore is how close the recommended configuration given by the advisor is to the best configuration determined experimentally. For this, we use the metric as defined in the evaluation methodology (§6.2): this distance metric combines the achieved completion time and financial cost, and produces a single distance value. We normalize the distance by that of the experimentally chosen best configuration distance (i.e., the best observed run). We show this value for all settings in Fig. 8. The configurations determined best experimentally and the advisor choices are shown in Fig. 9. The way to read this information is as follows:

For Q14 at SF=100, the advice is the same number of workers (Fig. 9a) but with more memory (Fig. 9b) than the actual best configuration. This results in an outcome 2% worse overall (Fig. 8) as it finishes 2% faster but at 9% higher cost.

Although the suggested configuration not always matches the best experimental run, the advisor chooses a configuration that is within 15% off the best one determined experimentally 21 out of 24 times. This is sufficiently accurate in practice given the large amount of uncertainty inherent to cloud execution and the problem at hand (see below for the noise inherent to the experimental runs). The larger differences observed for some queries and scale factors have two main reasons.

The first reason is the inherent service variance of the cloud. Especially the start-up time of workers and network rates when reading files exhibit large fluctuations and can occasionally experience very low performance. This is influential because of (a) the completion time being in the order of seconds where even a small delay immediately causes a large delay in percentage, and (b) the dependency on the tail: the completion time is determined by the last worker finishing and different forms of stragglers occur often, especially at larger scales. To illustrate this effect, we compare in Fig. 10 the distance metric for the best and the worst of the three runs of each of the 1108 configurations (contiguous line) and the subset of kneepoints (dotted line). For the kneepoints (the points we use as reference): half have a variance of more than 10% between best and worst and almost a quarter of them have a difference of over 20%. This is why we consider that an advice that is within 15% of the best observed run is a more than reasonable approximation
of a suitable configuration. In current systems, it is impossible to get more accuracy but the advice is still valuable in automating the process of deploying queries on serverless platforms.

The model limitations and assumptions are the second reason for differences between estimation and actual runs. The goal of the model is to capture the relative performance impact of the various components (i.e., start-up, input, exchanges, ...). Certain aspects of the Lambda and AWS are difficult to estimate and are often not needed to capture the overall trend. Yet, in some cases, these can have an impact in advice outcome. In particular, issues such as the temporary bursty higher network rate when having more than one vCPU, and the effect of concurrent processing and networking on their respective rates cause unusual experimental results that are infeasible to model.

Although the first reason above is always present, the latter reason is of most interest to explain the two cases in Fig. 8 which are more than 20% off.

In the first case, Q6 at SF=500 has a low completion time, with the best (kneepoint) choice finishing in 2.7 s. This amplifies any delay in terms of relative outcome: the advice outcome is 3.9 s (+43%) which is far from the best run but close to the other runs. Secondly, the model overestimates network and process performance of low memory workers, and as a result prefers the cheaper lower memory size. The preference for higher number of workers stems from modeling the worker start-up as a constant (it actually grows with the number of workers but it is difficult to predict by how much, introducing variances in the order of hundreds of milliseconds), and with increased probability of encountering stragglers at higher worker numbers (with stragglers being completely unpredictable). The second case, Q12 at SF=200, has an advice whose outcome is 28% off the best run with a completion time 12% slower and 41% more expensive. The model underestimates the increased processing and network rate at higher memory size, and is too optimistic regarding the additional overhead changing from a 1-level to a 2-level exchange (with 48 instead of 24 workers exceeding the threshold of 32), which experimentally does not yield a sufficiently cost-effective decrease in completion time.

Some of these aspects can be corrected an accounted for by modifying the constants or specializing the model even more for exchange heavy or scan heavy queries. We did not think it was effective to do so given the level of noise that is inherent to the system. In addition, if very many repetitions of the same query are run, a more realistic picture of its actual performance emerges but the number of runs is very high and the variance remains very high. For a system like Lambda, which targets occasional queries over cold data, the average over many runs is not relevant because queries will be run only once or twice and the probability that they run into the situations described is high as our own experiments with three runs for each configuration indicate (Figure 10).

```
Across the more representative workloads, we observe the following:
- There is significant variance in the outcome of configurations as serverless data processing systems depend on the
```

Figure 10: CDF comparing the performance of the worst and best repeated run of each configuration.

- The model is able to suggest a configuration with a good balance of completion time and financial cost: in the vast majority of cases it was less than 15% off the best choice a margin within the noise of the system.

10 RELATED WORK

There is a lot of work on resource allocation in the cloud for VMs [23] and containers [39]. Recent work from Microsoft has explored how to map serverless to the underlying VMs running functions from the perspective of the cloud provider [52] but that is a different problem from the one we address in this paper. There is also work in trying to explore the properties of serverless systems [45, 50]. Existing serverless data processing systems [38, 40, 51] show how to outperform alternatives in both completion and cost in select configurations, but use manually selected configurations.

Existing work on rough estimation of execution time and cost for VMs [30] does not directly translate to serverless functions due to: (a) the difference in the number of workers involved (10s for virtual machines, 100s for serverless functions), (b) the overhead incurred for data exchanges (typically through storage), and (c) the difference in task duration. Serverless operates in the few to tens of seconds range, whereas virtual machines in minutes to hours – factors insignificant at a one hour scale become so at a one second scale [45, 50]. Similar to [30], we account for processing (albeit using data and rate instead of CPU hours) and network overhead, but also incorporate the factors unique to serverless query processing systems, namely startup time, exchange overhead and request cost. Another related work is Astr(e)a [27, 28], which similarly proposes a parameterized model to estimate and advise serverless configurations. However, their model focuses on long running analytic queries and thus does not incorporate startup time and exchange overhead. Their configuration advice minimizes completion time or cost provided a user-defined budget or performance constraint respectively, rather than a mechanism as ours which determines a tradeoff between completion time and cost.
In the context of Apache Spark, SparkLens [42] is a tool which estimates completion time and utilization given a certain number of compute nodes after inspecting a single run – especially useful for repeated queries but not applicable in our use case. In the context of Azure Synapse, [44] proposes the AutoExecutor framework, built upon SCOPE [14], to automatically choose the number of executors for its Spark SQL queries. TASQ [41] which similarly is built on top of SCOPE [14], trains and makes use of (graph) neural networks to optimize the amount of tokens allocated to a job. Song et al. [46] proposes a multi-objective optimization method to tune several Spark parameters including number of executors and cores, as well as memory allocation. In the work, they ran a variety of run configurations and used its resulting traces to train for each workload neural network models for various objectives including latency and cost. Besides the different target platform, these works differ from ours in the design of the model. Our goal is to provide an advisory tool with a rough estimation model based on reasoning with parameters that can be adjusted for any serverless query processing system or particular query, rather than applying learning from traces with performance metrics to determine the best allocation. Our model incorporates the most important dimensions for serverless query processing in the cloud, namely start-up, compute, network and exchange overhead. [46] similarly made use of Pareto set to determine the best configuration, although the method at which the Pareto frontier is formed (via multi-objective optimization) differs from ours (which simply estimates the entire Cartesian product of possibilities), as well as the strategy of choosing the best configuration differs as they use direct distance to the idealized fastest and cheapest point rather than using that point as normalization for distance to the origin as we do.

11 CONCLUSION

We have presented an advisory tool for serverless data processing designed to pick a configuration (i.e., number of workers and their memory size) striking a good balance between performance (completion time) and cost. The advisory tool makes use of a rough estimation model which incorporates processing and networking, start-up, exchange overhead and request cost. We evaluated the advisor’s performance using an existing serverless data processing system and show it is able to identify desirable configurations. Automated configuration facilitates the job of the user and broadens the scope at which serverless data processing is cost-effective.

REFERENCES

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarajeta Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC ’18).

[2] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Anklet Singla. 2020. Specializing the network for scatter-gather workloads. In Proceedings of the 11th ACM Symposium on Cloud Computing, 267–280.

[3] Amazon. 2021. Amazon Redshift. https://aws.amazon.com/redshift/. Accessed: 29 November 2021.

[4] Amazon. 2021. AWS EC2: pricing. https://aws.amazon.com/ec2/pricing/. Accessed: 13 December 2021.

[5] Amazon. 2021. AWS Lambda: configuring lambda function options. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html. Accessed: 13 December 2021.

[6] Amazon. 2021. AWS Lambda: FAQs. https://aws.amazon.com/lambda/faqs/. Accessed: 5 November 2021.

[7] Amazon. 2021. AWS Lambda pricing (eu-west-1, Europe, Ireland). https://aws.amazon.com/lambda/pricing/. Accessed: 21 October 2021.

[8] Amazon. 2021. AWS Lambda: Serverless Computing. https://aws.amazon.com/lambda/. Accessed: 26 November 2021.

[9] Amazon. 2021. AWS S3: Cloud Object Storage. https://aws.amazon.com/s3/. Accessed: 26 November 2021.

[10] Amazon. 2021. AWS S3 pricing (eu-west-1, Europe, Ireland). https://aws.amazon.com/s3/pricing/. Accessed: 21 October 2021.

[11] Apache Arrow contributors. 2021. Apache Arrow. https://arrow.apache.org/. Accessed: 29 October 2021.

[12] Apache Iceberg contributors. 2022. Apache Iceberg. https://iceberg.apache.org/. Accessed: 28 July 2022.

[13] Paul Castro, Vatche Ishakian, Vinod Mathusamy, and Aleksander Sloiminski. 2019. The Rise of Serverless Computing. Commun. ACM 62, 12 (Nov 2019), 44–54.

[14] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shilb, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. Proc. VLDB Endow. 1, 2 (aug 2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[15] Benoit Daveville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jianheng Huang, Allison W. Lee, Ashish Motivola, Abdul Q. Muntz, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In Proceedings of the 2016 International Conference on Management of Data.

[16] Databricks. 2021. Databricks: The Data and AI Company. https://databricks.com/. Accessed: 26 November 2021.

[17] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. Commun. ACM 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/3/160173-the-tail-at-scale/fulltext

[18] Dong Du, Tianyi Xu, Yubin Xia, Binya Zang, Guangyu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyst: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[19] Google. 2021. Cloud Functions Execution Environment: File System. https://cloud.google.com/functions/docs/concepts/exec#file_system. Accessed: 13 December 2021.

[20] Google. 2021. Google Cloud. Cloud Functions. https://cloud.google.com/functions. Accessed: 26 November 2021.

[21] Google. 2021. Google Cloud Functions pricing. https://cloud.google.com/functions/pricing. Accessed: 13 December 2021.

[22] Google. 2021. Google Cloud pricing. https://cloud.google.com/compute/vm-instance-pricing. Accessed: 13 December 2021.

[23] Ori Hadary, Luke Marshall, Ishai Menache, Abhasek Pan, Essaia E Greff, David Dion, Star Dorminey, Shaleish Joshi, Yang Chen, Mark Rusinovich, and Thomas Moscibroida. 2020. Protean; VM Allocation Service at Scale. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI ’20).

[24] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. 2021. An Empirical Analysis of VM Startup Times in Public IaaS Clouds. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). DOI:10.1109/CL OUD53861.2021.00053

[25] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In Conference on Innovative Data Systems Research, CIDR’19.

[26] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshvaran Venkataraman, and Joseph M. Hellerstein. 2013. The Tail at Scale. Commun. ACM 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/3/160173-the-tail-at-scale/fulltext

[27] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with cost-efficiency and QoS-awareness. In Proceedings of the 2018 International Conference on Innovative Data Systems Research, CIDR’22.

[28] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2022. Astrea: Auto-Serverless Analytics with Cost-efficiency and QoS-awareness. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 756–765.

[29] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2022. Astrea: Auto-Serverless Analytics towards Cost-Efficiency and QoS-Awareness. IEEE Transactions on Parallel and Distributed Systems (2022).

[30] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with flint. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 451–455.

[31] Viktor Leis and Maximilian Kuschewski. 2021. Towards cost-optimal query processing in the cloud. Proceedings of the VLDB Endowment 14, 9 (2021), 1606–1612.

[32] Samuel Madden, Jaclin Ding, Tim Kraska, Sivaprasad Sudhir, David E Cohen, Timothy Mattson, and NoemieTabut. 2022. Self-Organizing Data Containers. In Conference on Innovative Data Systems Research (CIDR’22).

[33] Microsoft. 2021. Azure Container Instances pricing. https://azure.microsoft.com/en-us/pricing/details/container-instances/. Accessed: 13 December 2021.

[34] Microsoft. 2021. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. Accessed: 13 December 2021.

[35] Microsoft. 2021. Azure Linux Virtual Machines pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Accessed: 13 December 2021.
