Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit

Yang Tang  
Department of Computer Science  
Columbia University  
New York, NY, USA  
Email: ty@cs.columbia.edu

Junfeng Yang  
Department of Computer Science  
Columbia University  
New York, NY, USA  
Email: junfeng@cs.columbia.edu

Abstract—Serverless computing emerges as a new paradigm to build cloud applications. Developers write small functions that react to cloud infrastructure events, while the cloud provider maintains all resources and schedules the functions in containers. Thus, developers can focus on their core business logic, and leave server management and scaling to the cloud providers.

Unfortunately, existing serverless computing systems suffer from a key limitation that deprives them of enjoying significant speedups. Specifically, they treat each cloud function as a black box and are blind to which data the function reads or writes, therefore missing potentially huge optimization opportunities.

We present LAMBDATA, a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. Once data intents are made explicit, LAMBDATA performs a variety of optimizations to improve speed, including caching data locally and scheduling functions based on code and data locality.

Our evaluation of LAMBDATA shows that it achieves an average speedup of 1.51x on the turnaround time of practical workloads and reduces monetary cost by 16.5%.

Keywords—serverless computing; cloud function; cloud storage.

Serverless computing emerges as a new paradigm to build cloud applications. Developers write small functions, called cloud functions, that react to cloud infrastructure events, while the cloud provider maintains all resources and schedules the functions in containers. Thus, developers can focus on their core business logic, and leave server management and scaling to the cloud providers.

Besides ease of programming, serverless computing provides more efficient and fine-grained scaling than traditional clouds, because containers are more lightweight than virtual machines. It adjusts to dynamic workload at a per-function level and scales up or down in a second.

As a result, many companies, including Netflix, Coca-Cola, and the New York Times, are adopting serverless computing [1]. A 2018 survey of 600 IT decision-makers shows that 61% of respondents are already using or plan to use serverless computing by 2020 [2]. Practically, all major cloud providers provide serverless computing services, namely AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions.

Unfortunately, existing serverless computing systems suffer from a key limitation that deprives them of enjoying significant speedups. Specifically, they treat each cloud function as a black box and are blind to which data the function reads or writes, therefore missing potentially huge optimization opportunities. For instance, they schedule multiple functions working on the same data to run on different machines, neglecting data locality. Thus, each machine has to fetch a copy of the data, resulting in a 42% slowdown and 19.2% more monetary cost (see §II-C).

We present LAMBDATA, a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. Figure 1 shows an example that a thumbnail function intends to read pic/1.jpg and write thumb/1.jpg. Once data intents are made explicit, LAMBDATA performs a variety of optimizations to improve speed, such as colocalizing functions working on the same data. These intents are hints only: if a developer misses an intent or specifies an incorrect one, performance may be affected but not correctness.

Our design of LAMBDATA is strongly motivated by two key insights in serverless computing. First, a cloud function is almost always small doing a single task; therefore, developers can easily specify its inputs and outputs before executing it, as illustrated in Figure 1. In other words, the paths to input and output data are typically not calculated on the fly amid the execution of a cloud function. While current serverless clouds allow a developer to write a large monolithic function that dynamically computes data locations and accesses the data, such use would defeat the main benefit of serverless computing.

Second, based on our study of open-source serverless applicants and our own experience building such applications, a cloud function tends to be functional in the sense that it outputs an immutable object: once the object is written, the application does not mutate it. If an update is needed, the

Figure 1. Example of specifying data intent for a cloud function that generates the thumbnail of an image. It reads input from pic/1.jpg and writes output to thumb/1.jpg.
application simply writes a new object under a new path. This approach avoids complicating cloud functions with tricky concurrent and partial update handling logic, and it is a natural fallout from the idempotency requirement of the underlying serverless cloud (the cloud may kill and restart a cloud function without notification due to resource constraints or tail latencies). This insight enables LAMBDA to aggressively cache data objects throughout the system to improve locality without concerning consistency issues.

Operationally, LAMBDA works as follows. It leverages existing cloud object storage (e.g., AWS S3) to store data. LAMBDA adds a caching layer, where each computing node has its own object cache. LAMBDA schedules cloud functions based on both code and data locality. It tends to schedule multiple function invocations working on the same data on the same computing node so that they can reuse cached data.

Compared with Pocket [3], we choose to build LAMBDA on top of existing cloud object storage. The benefits are two folds. First, using cloud storage is the best practice recommended by Amazon [4] and Google [5], since data objects enjoy the high durability they offer. Second, developers are familiar with this programming model, since they do not need to decide what data should be durable and what data can be put on ephemeral storage.

Our evaluation of LAMBDA on an Instagram-like application and an online classroom application shows that on average, LAMBDA achieves $1.51 \times$ speedup on the turnaround time of practical workloads and reduces monetary cost by 16.5%. The source code of LAMBDA and the applications we wrote for the evaluations are at https://columbia.github.io/lambdata.

I. BACKGROUND: SERVERLESS COMPUTING

In serverless computing, the basic building block is a function. A cloud function is similar to a function in a computer program, in that it takes some parameters, performs a task, and returns a result. A cloud function can be triggered by another function, by a RESTful API, or by a cloud event, such as when the cloud storage receives a new file or a database gets a new entry.

With serverless computing, developers need not manage any infrastructure. The cloud service provider handles all resource management. It runs a function in a container, and each container is isolated from one another. When a function ends, its container is paused for a few minutes before being terminated. If the same function gets invoked again while the container is paused, the same container will be resumed, which we call a warm start. Otherwise, it is a cold start.

Cloud functions cannot rely on containers to persist any state, because containers are ephemeral. Cloud service providers also limit the size of function parameters and return values to a few hundred kilobytes, making it impossible to pass large data this way. As a result, cloud functions have to leverage cloud storage services (e.g., AWS S3) to store or pass any non-trivial data.

The price for using serverless computing services typically consists of two parts: a flat-rate cost per function invocation ("request cost"), plus a cost proportional to the function's run time ("duration cost"). The request cost is only a fraction of the duration cost. Hence, it is desirable to minimize the function run time.

A. Overview of serverless architecture

A typical serverless cloud (e.g., Apache OPENWHISK [6], [7]) consists of two fundamental entities: one Controller and multiple Invokers (Figure 2). The Controller is the orchestrator of the system, and the Invokers are the executors. They communicate through a publish-subscribe messaging system (e.g., Apache KAFKA [8]).

To invoke a function, the Controller schedules the invocation to run on an Invoker and publishes an invocation message. When the Invoker receives an invocation message, it publishes an acknowledgment message, and starts or resumes a container to run the function.

B. Life of a cloud function

A typical lifecycle of a cloud function consists of four phases: start, get, compute, and put.

The start phase is starting up the function. For a cold start, the cloud starts a new container, downloads the function code to the container, and invokes the function. The function may then install additional packages (e.g., OpenCV, FFmpeg, or lsqlX) or make one-time network connections (e.g., to a database). For a warm start, the cloud resumes an existing container and invokes the function. A warm start takes less than 20ms, while a cold start usually takes more than 1s, depending on the function.

The get phase is getting the input data from the cloud storage service. The typical time spent on getting the data is between 100ms and 5s, depending on the data size.

The compute phase is performing the actual computation on the data. Although different functions have vastly different computations, most functions are quick tasks that finish within 3 seconds, the default time limit on AWS Lambda.

The put phase is putting the output data back to the cloud storage. Different functions generate different sizes of data, which usually takes between 100ms and 5s to upload. A special case is if a function performs a classification task,
such as malware detection or image recognition, it usually leverages the function’s return value or uses a database service (e.g., DYNAMODB) instead, without putting data back to the cloud storage.

II. A Motivating Example

A. Example and insights

We motivate the design of LAMBDATA through an example of a photo-sharing application with four cloud functions: malware, compress, thumbnail, and collage. We compose these functions into two workflows: handling user upload and making collage. Figure 3 shows the triggering of functions and how the data flows in and out of the cloud storage.

Handling user upload. The user uploads an image using a front-end app (e.g., a smartphone app), which puts the image on the cloud storage. As the cloud storage receives the data, it automatically triggers malware for next-stage processing.

Function malware is triggered by a file-upload event of the cloud storage service. When triggered, this function fetches the data from the cloud storage and runs a malware-detection program. If the file is clean, then it triggers both compress and thumbnail simultaneously for next-stage processing. Otherwise, it discards the file.

Function compress gets an image file from the cloud storage, compresses it, and puts it back to the cloud storage. Similarly, function thumbnail gets an image from the cloud storage, generates a thumbnail, and puts it back to the cloud.

Making collage. The user can also make a collage out of several existing images. The front-end app sends a REST request to the cloud gateway, which triggers collage with a list of image keys. Function collage gets each image file from the cloud storage, generates a collage image, and puts it back to the cloud. It also triggers compress and thumbnail to compress the collage and generate a thumbnail of it.

Insights. From the example, we observe two key insights in serverless computing. We have also studied 12 practical, real-world open-source serverless applications on Github and built two serverless applications ourselves.¹ We find that our insights are valid on all these applications.

Our first insight is that a cloud function is almost always small doing a single task; therefore, developers can easily determine its inputs and outputs before executing it, rather than calculate the object names on the fly. Table I shows an example of inputs and outputs of each function.

If a function is triggered by a cloud storage event, then the input is just the object that emits the event. For example, in the Upload workflow, malware is triggered by the cloud storage when it receives a new image (e.g., pic/1.jpg), so the input is just pic/1.jpg. The developer can easily calculate the output objects deterministically before executing the function. Function malware only appends an entry in the database and does not generate new data objects, so the output is empty. If the function wrote to the cloud storage instead of the database, then it would specify something like result/1.txt as the output.

If a function is triggered by another function or a REST request, then the developer can specify the inputs and outputs based on her intent of invoking the function. For example, in the Collage-making workflow, the front-end application invokes collage via a REST request to combine a list of images into a collage. So the inputs are the list of image objects (pic/1.jpg...pic/5.jpg), and the output is the intended filename of the collage object (collage/collage1.jpg). Function collage further invokes compress and thumbnail to compress the image and generate a thumbnail, so it specifies the collage file as the input, and the outputs are just the same filename prepended with buckets small/ and thumb/, respectively.

Our second insight is that a cloud function tends to be functional in the sense that it outputs an immutable object; once the object is written, the application does not mutate it. Because the serverless computing providers may kill a function or run a function more than once without any notice, they require that all functions be idempotent. Therefore, most developers write functions in a purely functional way, so that it is much easier to reason about the behaviors and handle failures.

For example, all of our four functions are purely functional, in that they never mutate data, and always generate the same output for the same input. Specifically, the image-compression function does not modify the input object in-

¹Because serverless computing is a relatively new paradigm, we could only find a few real-world open-source serverless projects beyond proof-of-concept demos and tutorials.
place but rather writes the result as a new object. Otherwise, if the function is invoked twice, it would end up with double-compressing the image.

These two insights enable LAMBDATA to cache data aggressively without worrying about data inconsistency, and schedule function invocations by considering both code and data locality.

B. Inefficiencies with existing serverless clouds

Existing serverless clouds regard a function as a black box, treating all invocations of a function in the same way. When scheduling functions, they only consider the function code, but not the data that the function computes. As a result, they tend to schedule multiple invocations of the same function on the same Invoker. For example, we ran the workload of handling user uploads on two images, using OPENWHISK with two Invokers. We found that Invoker1 handles all invocations of malware, and Invoker2 handles all invocations of compress and thumbnail, because this schedule is optimized for reusing warm containers.

Unfortunately, this scheduling is inefficient, because functions in both Invokers need to get both images from the cloud storage. As cloud functions are typically small, the time a function spent on getting data is significant. Table I shows that the functions in our example spend 40% of the time getting duplicate data from the cloud storage. Besides wasting time, it also costs more money because the cloud storage charges for each request.

The fundamental cause of this inefficiency is that existing serverless clouds have no way of knowing a function’s data intents (i.e., what data the function reads and writes); therefore, they cannot leverage such information for scheduling.

C. LAMBDATA’s optimizations

LAMBDATA optimizes for this inefficiency by making a cloud function’s data intents explicit. Developers or cloud events can easily annotate the input and output data when invoking a function, using two special fields in the function’s parameter list: get_data for input and put_data for output. These annotations enables the serverless cloud’s Controller to see through the black box when scheduling a function invocation. For example, the previous two invocations of thumbnail now look different with LAMBDATA:

```javascript
thumbnail({
  "get_data": ["pic/1.jpg"],
  "put_data": ["thumb/1.jpg"]
})

thumbnail({
  "get_data": ["pic/2.jpg"],
  "put_data": ["thumb/2.jpg"]
})
```

Under the hood, LAMBDATA securely caches data locally on each Invoker, and the Controller takes into account both code and data locality when scheduling function invocations. We noticed three common data usage patterns of cloud functions while studying open-source serverless applications and writing our own applications, and designed LAMBDATA accordingly.

Temporal locality of data. A data is often reused by multiple functions within a short period of time. For example, in the workflow of handling user upload, immediately after malware finishes computation on a file, both compress and thumbnail perform computations on the same file concurrently. LAMBDATA securely caches the file locally on the Invoker, and schedules all three function invocations on the same Invoker according to their data intents (if it deems worthy, see §V). Therefore, only malware needs to get the data from the cloud, while compress and thumbnail read the cached data, reducing both time and monetary cost.

Spatial locality of data. Many tasks or workflows involve multiple functions computing on a small set of closely-related data. For example, a user often uploads several images in a row and then creates a collage from these images. Therefore, even a small cache provides many benefits.

Data pipelining. Multiple functions often process data as a pipeline. For example, in the collage-making workflow, functions collage and thumbnail form a pipeline, i.e., collage triggers thumbnail, and the output of collage directly becomes the input of thumbnail. With existing serverless clouds, thumbnail cannot start until collage finishes putting the data to the cloud storage. However, the real dependency between the two functions only lies in the actual computation of the data (the compute phase), as shown in Figure 4. Because LAMBDATA caches both input and output data, it schedules the next-stage function on the same Invoker as soon as the previous function enters the put phase (if worthy, see §V), effectively overlapping functions in a pipeline. Figure 5 shows LAMBDATA’s scheduling for this example: thumbnail starts when compute finishes, and it gets the cleanup output from the cache rather than going through the cloud storage.

In the rare case, if LAMBDATA schedules thumbnail on a different Invoker but thumbnail enters the get phase before collage finishes the put (Figure 6), then LAMBDATA would fail this invocation and retry it. Because serverless

---

**Table II**

<table>
<thead>
<tr>
<th>Function</th>
<th>start</th>
<th>get</th>
<th>compute</th>
<th>put</th>
<th>% (get)</th>
</tr>
</thead>
<tbody>
<tr>
<td>malware</td>
<td>402</td>
<td>208</td>
<td>69</td>
<td>n/a</td>
<td>44%</td>
</tr>
<tr>
<td>compress</td>
<td>132</td>
<td>216</td>
<td>117</td>
<td>83</td>
<td>39%</td>
</tr>
<tr>
<td>thumbnail</td>
<td>137</td>
<td>201</td>
<td>79</td>
<td>48</td>
<td>40%</td>
</tr>
</tbody>
</table>

---

**Figure 4.** Dependency between collage and thumbnail.
computing providers may kill and restart a function without notice, and thus require all cloud functions to be idempotent. LAMBDA's behavior does not impose any new limitations.

We ran the same workload from §II-B on LAMBDA, and got the scheduling shown in Figure 7: the first Invoker handles all invocations related to 1.jpg, and the second Invoker handles those related to 2.jpg. As a result, LAMBDA reduced the overall turnaround time by 29.6%, a 1.42x speedup, and reduced the monetary cost by 19.2%. We show more case studies in §VI.

III. LAMBDA API

LAMBDA exports a simple yet effective API for serverless functions to deal with cloud storage systems.

**Basic cloud storage methods.** LAMBDA's API for basic cloud storage methods resembles the cloud storage's original API, such as get and put.

```python
object = get(bucket, key)
put(object, bucket, key)
```

Developers simply import the LAMBDA library and use these methods in the same familiar way.

**Data intents.** LAMBDA lets a function specify in its parameters what data it needs to get and put. A function in current serverless computing services represents all parameters as a single JSON string. LAMBDA inserts three new fields get_data, put_data, and num_threads into the JSON. The developer specifies by get_data and put_data all data it needs to get and put, both as an array of (bucket, key) pairs, and by num_threads the number of threads it uses to connect to the cloud storage for parallel downloads. Figure 8 shows an example annotation of invoking the collage function to make a collage from two images, using up to 5 threads for getting the data. Therefore, the scheduler knows each function’s data intents by peeking at the JSON string, before running the function.

We note that many existing serverless functions already include an equivalent of get_data and put_data in their parameters, but there is no standard on how they would name these fields. For those functions, developers simply need to change the names of these fields into get_data and put_data, and enjoy the benefits of LAMBDA.

All annotations are hints only. If a developer misses an intent or specifies an incorrect one, performance may be affected but not correctness. For example, if get_data is missing, LAMBDA may not schedule the function on the best Invoker, but the function can still opportunistically benefit from cached data. If put_data is missing (or if the developer cannot determine put_data ahead of time), LAMBDA can provide all benefits except that it would not overlap functions in a pipeline. If num_threads is missing, LAMBDA assumes the function gets data sequentially.

IV. LAMBDA’S ARCHITECTURE

We modified OPENWHISK and added LAMBDA to several components. Figure 9 shows the architecture.

**Controller.** We implement a data-aware scheduler in the Controller, which consists of a cache registry, a data size registry, and some profiling results. The cache registry keeps a list of cached data keys for each Invoker, possibly stale. The data size registry maintains the size of all data objects. The profiling results contain the recent performance of the cloud, including the time to start a container, the time to get and put a file from the cloud storage. The Controller does not actively probe any information, but only bookkeeps information sent by Invokers.

**Invoker.** Each Invoker independently manages its own cache and sends a list of all currently cached data keys and the size of each data object to the Controller by piggybacking it with the acknowledgment message of each function invocation. It also monitors the time to resume a warm container or start a cold container for each function. Whenever it gets or puts a data object to the cloud storage, it monitors the time it takes, in order to estimate the time of cloud storage operation for various data sizes. It sends these profiling results to the Controller via the acknowledgment message, too.
Each Invoker exports the LAMBDA API via Unix domain socket. Each container on the Invoker has access to the API socket, and securely mounts a portion of the cache that the function’s identity has access to.

V. DATA-AWARE SCHEDULING

In existing serverless cloud architectures such as OPENWHISK, each Invoker individually manages containers, and the Controller does not know where the warm containers are. Thus, the Controller only uses a deterministic hash of the function to pick an Invoker. Although this method has a good chance of picking a warm container, it fails to consider data locality.

By contrast, LAMBDA employs a data-aware scheduling algorithm. Among the four phases of a serverless function, the compute and put phases are essential computations, unaffected by scheduling. Therefore, LAMBDA’s Controller schedules an Invoker in order to minimize the lead time, defined as the time spent in the start and the get phases, based on the bookkeeping information and profiling results. We denote them by $T_{\text{start}}$ and $T_{\text{get}}$, respectively.

A. Estimating the lead time

The start phase. Let us name the function $f$. We denote by $T_{\text{warm}}$ the time to resume a warm container, and notice that it is fast regardless of what function is in it. We denote by $T_{\text{cold}}$ the time in the start phase if we have to start a new container, and find that it is relatively stable for the same function but varies greatly across functions, so we model it as a function of $f$. We notice that $T_{\text{cold}}$ is not determined by the code size of $f$ because some functions install additional packages or make one-time network connections after the initialization of the container. To deal with this issue, LAMBDA monitors $T_{\text{cold}}(f)$ from the initialization to $f$’s first LAMBDA API call of the get method. If LAMBDA has never seen $f$ before, it estimates $T_{\text{cold}}(f)$ by other functions with similar code size as $f$. Therefore, the time spent in the start phase is:

$$T_{\text{start}} = \begin{cases} T_{\text{warm}} & \text{if a warm container is available} \\ T_{\text{cold}}(f) & \text{otherwise} \end{cases}$$

The get phase. Let $D = \{d_1, d_2, \ldots, d_n\}$ be the set of data that $f$ depends on. We denote by $T_{\text{data}}(d)$ the time to get data $d$ from the cloud storage. We notice that $T_{\text{data}}(d)$ is mainly determined by the size of $d$ and the region of the cloud storage, so LAMBDA monitors the time spent on recent cloud storage requests to track the relationship between $T_{\text{data}}$ and object size dynamically. Let $D_c \subseteq D$ be the subset of cached data and $D_n = D \setminus D_c$ the subset of data absent. If $d_i \in D_c$, then the function can read it immediately. Otherwise, if $d_i \in D_n$, the function needs to get it from the cloud storage, using $n$ concurrent threads ($n$ is an optional annotation provided by the developer, see §III). In order to calculate the total time to get all the data, we consider two cases. If $|D_n| \leq n$, meaning that there are enough threads to download all data in parallel, then the time is dominated by the slowest thread (i.e., getting the largest data). If $|D_n| > n$, then the time is approximately the total time for getting all data over $n$ threads. Therefore, the time spent in the get phase is:

$$T_{\text{get}} = \begin{cases} \max_{d \in D_n} T_{\text{data}}(d) & \text{if } |D_n| \leq n \\ \frac{\sum_{d \in D_n} T_{\text{data}}(d)}{n} & \text{otherwise} \end{cases}$$

Total lead time. As a result, the total lead time is $T = T_{\text{start}} + T_{\text{get}}$. LAMBDA tries to schedule the function on the Invoker with the smallest $T$. If the best Invoker is offline or overloaded, then it picks the next one, and so forth. LAMBDA uses the same load-detection mechanism as existing serverless computing services.

B. Collecting bookkeeping data

When an Invoker receives a function invocation, it sends an acknowledgment message to the Controller. LAMBDA piggybacks with this message a list of currently cached data, as an array of (bucket, key, size) pairs. The reason to include data size is to help the scheduling algorithm determine how long it would take to get the data from the cloud storage, on other Invoices that has not cached the data or on the same Invoker if the data is evicted from the cache.

The Controller has a cache registry that maintains a global view of all data currently cached at each Invoker as a dictionary of Invoker → Set[(bucket, key)]. It also maintains the size of all data as a dictionary of (bucket, key) → size. Both dictionaries may be stale or incomplete, which only affects scheduling efficiency but not the correctness of the system. For example, the Controller may think a data is cached on an Invoker, but it has actually been evicted since the last acknowledgment message. In this case, the function simply gets the data from the cloud storage again. If any dictionary grows too large, the Controller just purges old entries.

In order to estimate $T_{\text{warm}}$, $T_{\text{cold}}$, and $T_{\text{data}}$, the Invoker also monitors the time whenever it starts a container or handles a cloud storage operation, and sends them to

![Figure 9. LAMBDA’s architecture. Components with italic font are LAMBDA-specific.](image-url)
the Controller via the acknowledgment message. The Controller bookkeeps recent profiling results and interpo-
lates them for estimation.

VI. EVALUATION

We deployed LAMBDA on Amazon EC2 in the us-east–1 region, with an m5a.large instance as the Controller and five m5a.2xlarge instances as Invokers. All instances are running Ubuntu 18.04 and Docker CE 17.03.3. We used Amazon S3 as the cloud storage. We implemented LAMBDA in Scala 2.12 atop OPENWHISK and wrote all cloud functions in Python 3.6. We limit each function’s memory usage and cache size to 512MB.

Our experiments aim to answer four research questions:

A. Is LAMBDA fast handling cloud storage requests?
B. Does LAMBDA speed up function invocations?
C. Does LAMBDA speed up multi-function workflows?
D. Does LAMBDA reduce monetary cost?

A. Microbenchmark

We first evaluate LAMBDA’s performance of basic cloud storage operations. We get and put data objects from 1MB to 256MB, to Amazon S3, using both BOTO3, the official AWS Python library, and LAMBDA. Figure 10 shows the median time for each operation.

For the get operation, if the data is cached, LAMBDA takes less than 1ms, because it is simply accessing files on the local disk, and LAMBDA does not need to do anything. If the data is missing, LAMBDA’s performance is comparable to BOTO3’s up to 16MB, and it shows a speedup of up to 1.6× with larger data. For the put operation, LAMBDA’s performance is comparable to BOTO3’s for small data, and it shows a 1.85× speedup for 256MB data. These speedups are because BOTO3 connects to the cloud storage in the Python runtime inside a container, whereas LAMBDA connects to the cloud storage in the Invoker, using a Java runtime outside of containers. Since the LAMBDA is just a wrapper over the underlying AWS SDK, we do not claim any contribution on the speedups. Nevertheless, the results show that LAMBDA performs well for basic cloud storage operations.

B. Function performance

To evaluate LAMBDA’s performance on running functions, we wrote two serverless applications modeled from real-world applications, each with 10 functions. Table III lists all the functions and their parameters. We now briefly describe these two applications.

Photo sharing. We modeled this application according to Instagram, a popular photo-sharing application. Users can upload images, create short video stories from images, apply filters or add special effects, and publish them. The application also includes functions to scan for malware, compress images, and transcode videos. Although Instagram performs many computations (e.g., apply filters) locally on a mobile phone, we implement everything as cloud functions to demonstrate the feasibility. We used images from the Div2K dataset [9], [10] as the workload.

Online classroom. We modeled this application according to Canvas, a popular online learning management system. Teachers can manage lecture notes, assign homework, prepare exams, and grade them. We used documents at one author’s institution as the workload.

1) Speedup of function invocations: To evaluate the speedup of function invocations, we instrumented the time spent on each phase of a function invocation. Table IV shows the breakdown of each phase’s time, and the speedup of LAMBDA compared with OPENWHISK, where all numbers are the median of 100 invocations. In order to eliminate the variance in the start time and make a fair comparison, we pre-warmed the container before each invocation.

First, we find that cloud functions are small and short-running. With practical workloads, all function invocation times are shorter than 10 seconds, and the majority shorter than 2 seconds.

We further find that the time spent on each phase varies significantly across functions. Both get and put phases take hundreds of milliseconds for most functions. They are mainly determined by the number of data objects and each object’s size. The times of the compute phase are diverse, ranging from 0 to 8 seconds. They are mainly determined by the function’s business logic.

We compared the run time of LAMBDA with OPENWHISK and found no statistically significant difference if the data is not in the cache. If the data is cached, LAMBDA gives an average speedup of 1.50×. Of all functions, the speedup is higher (up to 3.99×) if the get phase dominates, and lower (1.02×) if the compute phase dominates.

Overall, our experiment shows that LAMBDA offers significant speedup over existing serverless clouds.

2) Case study: In order to understand how data matters to LAMBDA’s performance, we show a case study on a typical cloud function: thumbnail. This function gets an image file, generates its thumbnail, and puts it on the cloud storage. We study how various image size affects both

![Figure 10. Microbenchmark: median time to get and put objects of various sizes to Amazon S3. Lower is better.](image-url)
The top bar in Figure 11 is the timeline of OPENWHISK, and the bottom bar is the timeline of LAMBDATA. We observe that all phases’ run time increase with the image dimension, but the ratio of these increases is nonlinear. For example, the compute time on an iPhone image is 6.9 times as long as on a web image, while the get time is only 2.6 times as long. LAMBDATA shortens the get time to almost zero, but the compute time remains the same. Therefore, the speedup LAMBDATA provides is nonlinear with regard to the image dimension. We find that while LAMBDATA gives speedup for all images, it works best with iPhone photos.

The same observation applies for all functions. We conclude that LAMBDATA works well with all real-world data.

C. Workflow performance

To evaluate LAMBDATA’s performance on real-world usage involving multiple functions, we simulated 10 workflows that resemble practical scenarios and used synthetic workloads derived from real-world parameters to trigger these workflows.
functions. We ran each workload 15 times and chose the result with the median turnaround time. We now show the most typical workflow: slideshow in the photo-sharing app. In this workflow, the user creates a short 1080P M-JPEG video with five recently-uploaded images and adds some text to the video. The application then transcodes the video to the msmpeg4v2 format and publishes the converted video.

Figure 12 shows the timeline of this workflow. The top cluster is the baseline running on OpenWhisk, and the bottom cluster is of LAMBDATA. This workflow has four functions chaining into one pipeline: create story, add text, transcoding, and publish. Since the user has just uploaded the source images, the data cache is warm. For each individual function, LAMBDATA’s run time is shorter than the baseline, because it reuses data from the cache.

For the orchestration of the four functions, the baseline schedules each function invocations consecutively, whereas LAMBDATA overlaps the next-stage functions with put phase of the previous function, further reducing the turnaround time.

Overall, LAMBDATA achieves a 2.16× speedup to finish the workflow. Across all 10 workflows, LAMBDATA achieves an average of 1.51× speedup.

D. Cost savings

LAMBDATA’s cost savings come from two factors. First, LAMBDATA shortens the run time of cloud functions, thus reducing cost on the serverless computing service. Second, LAMBDATA eliminates redundant requests to the cloud storage, thus reducing cost on the storage service. We applied the current pricing model of AWS Lambda and AWS S3 and calculated the cost savings for all workflows.²

Table V shows the cost savings for the slideshow workflow in §VI-C, where LAMBDATA reduces its cost by 28.6%.

Across all workflows, LAMBDATA’s achieves an average cost savings of 16.5%.

VII. DISCUSSION

Cache coherence. LAMBDATA requires that data are immutable, so the cache is never incoherent. This requirement follows our insights (§II-A) and is recommended by cloud vendors. If a function needs to mutate data, it should store the data with a new key and delete the old data.

Concurrent writes. Writing different data to the same object from multiple functions violates our requirement that data are immutable, and is a bad practice in any serverless computing. LAMBDATA does not prevent concurrent writes but leaves it to the developer to use distinct object keys.

Cache eviction policy. The cache eviction policy is orthogonal to LAMBDATA’s design. LAMBDATA can use any policy to evict cache. Because data in serverless computing are small and demonstrate good temporal and spatial locality, the cache need not be large. In practice, we find that a 512MB cache with a simple LRU algorithm works well, with no implications on the maximum throughput or the number of functions that can be run in parallel inside an Invoker.

Data prefetching. By making a cloud function’s data intents explicit, one further optimization is that the Invoker can prefetch data on behalf of a function, while the container is being initialized. LAMBDATA did not implement this optimization because cloud providers charge users by the duration a container runs. This prefetching that happens outside a container’s lifetime would violate the billing model.

Security. LAMBDATA maintains the same container isolations as in existing serverless clouds, except for the cache. It leverages existing cloud services’ identity and access management (IAM) policies to restrict what cache a function can access. Only functions under the same identity can see one another’s cached data. A malicious function could try to cache a lot of data in the hope of exhausting the cache space and evicting other identities’ cached data. LAMBDATA can mitigate this impact by imposing a limit on the maximum cache size per identity.

VIII. RELATED WORK

Memoization for dataflow programs. Memoization [17], [18], [19] is a technique that reuses prior computation results of pure functions. Nectar [20] manages data and computation in the traditional data center setting. It memoizes intermediate computation results of Dryad programs. Incoop [21] uses memoization on the MapReduce framework. However, these systems only work for specific programming models, and it is non-trivial to generalize their use cases to serverless computing where code and data are decentralized. LAMBDATA generalizes the idea of memoization to cloud functions.
Scheduling workflows. Numerous work studies the scheduling of cloud workflows [22], [23], [24]. For example, Oozie [25] manages workflows for Hadoop systems, and Yu et al. [26] schedules workflows for grid computing. Unfortunately, their models do not fit in the scope of serverless programming. Cloud bursting schedulers, such as [27], only consider occasional workload offloading rather than general cloud computing use cases. DEWE v3 [28] employs a hybrid execution model on serverless computing, but it focuses on scientific workflows and resource underutilization, different from LAMBDA’s goal of data-awareness.

Serverless function orchestration. AWS Step Functions [29] manages serverless computing workflows by describing functions as a state machine. Although it maintains states for serverless applications, the 32KB size limit is insufficient for large data objects. IBM Composer [30] and Azure Durable Functions [31] let developers write function compositions with special library functions, and allow larger state size. Nevertheless, these frameworks do not consider data locality, and their states are only for intermediate data, not persisted in the cloud storage. Besides, they all require new programming models unfamiliar to developers. By contrast, with LAMBDA, developers write functions and manipulate data objects in a familiar way.

Data systems for serverless computing. Pocket [3] introduces a multi-tier storage system for interactive serverless data-analytic applications. However, it focuses on intermediate data and does not provide high data durability. By contrast, we choose to build LAMBDA on top of existing cloud storage so that data are highly durable, and developers are familiar with this programming model.

IX. CONCLUSION

This paper presented LAMBDA, a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. It caches data locally, and its data-aware scheduling algorithm considers both code and data locality. Evaluation on two practical applications with 20 cloud functions shows that it achieves an average of 1.51× speedup on the turnaround time and reduces monetary cost by 16.5%. The source code of LAMBDA and the applications we wrote for the evaluations are at https://columbia.github.io/lambdata.

REFERENCES