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, in which developers write small functions that react to cloud infrastructure events, and cloud providers maintain all resources and schedule the functions in containers. Serverless computing thus enables developers to focus on their core business logic and leave server management and scaling to 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, such as caching data and colocating functions.

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, where 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 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 critical limitation that deprives them of enjoying significant speedups. Specifically, they treat each cloud function as a black box, and they are blind to which data the function reads or writes. Therefore, they miss 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).

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

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`.

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 intents 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 colocating 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 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 LAMBDATA to aggressively cache data objects throughout the system to improve locality without concerning consistency issues.

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

Compared with Pocket [3], we choose to build LAMBDATA 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 should be on ephemeral storage.

Our evaluation of LAMBDATA on an Instagram-like application and an online classroom application shows that on average, LAMBDATA achieves 1.51× speedup on the turnaround time of practical workloads and reduces monetary cost by 16.5%. All source code of LAMBDATA and benchmarking applications we wrote 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.

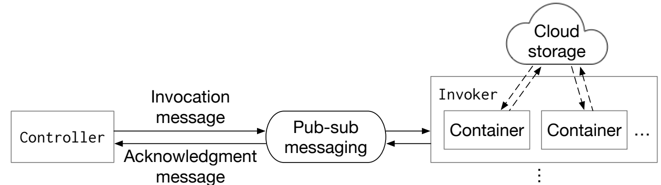

Figure 2. Overview of serverless architecture.

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 typically a tiny 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 LaTeX) 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 one second, 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. If a

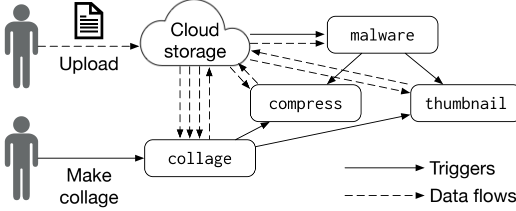

Figure 3. A photo-sharing application example. Solid arrows indicate triggers. Dashed arrows represent data flows.

function's output is small (*e.g.*, malware detection or image recognition), it may leverage the function's return value or use 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 an Instagram-like photo-sharing application with two workflows, *handling user upload* and *making collage*, using four cloud functions (`malware`, `compress`, `thumbnail`, and `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 the top 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.

[1]Because 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 at the time of writing.

Table I
INPUTS AND OUTPUTS OF EACH FUNCTION.

	Input	Output
Workflow: handle user upload		
`malware`	pic/1.jpg	none (return value only)
`compress`	pic/1.jpg	small/1.jpg
`thumbnail`	pic/1.jpg	thumb/1.jpg
Workflow: make collage		
`collage`	pic/1.jpg—pic/5.jpg	collage/col1.jpg
`compress`	collage/col1.jpg	small/col1.jpg
`thumbnail`	collage/col1.jpg	thumb/col1.jpg

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 with regard to the data objects, 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-

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 consider only 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 spends on getting data is significant. Table II 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 enable 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:

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

Table II
TIME SPENT ON EACH PHASE OF THE FUNCTIONS, IN MILLISECONDS.

Function	start	get	compute	put	%(get)
`malware`	402	208	69	n/a	44%
`compress`	132	216	117	83	39%
`thumbnail`	137	201	79	48	40%

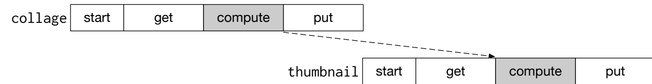

Figure 4. Dependency between `collage` and `thumbnail`.

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 costs.

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 `collage` finishes computing, and it gets the collage 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

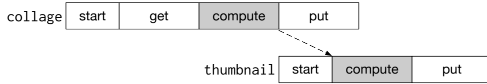

Figure 5. Overlapping functions in a pipeline.

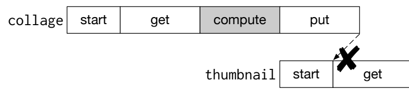

Figure 6. A rare case of overlapping functions in a pipeline.

computing providers may kill and restart a function without notice, and thus require all cloud functions to be idempotent, LAMBDATA's behavior does not impose any new limitations.

We ran the same workload from §II-B on LAMBDATA, 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, LAMBDATA reduced the overall turnaround time by 29.6%, a 1.42× speedup, and reduced the monetary cost by 19.2%. We show more case studies in §VI.

III. LAMBDATA API

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

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

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

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

Data intents. LAMBDATA 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. LAMBDATA inserts three optional 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 to `get_data` and `put_data`, and enjoy the benefits of LAMBDATA.

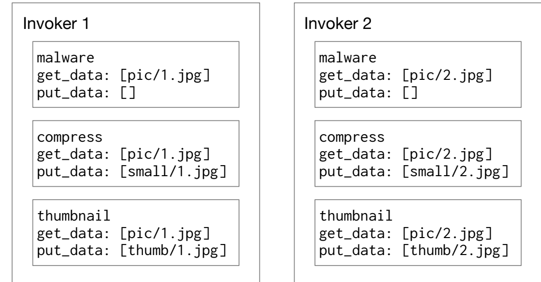

Figure 7. Optimized scheduling with LAMBDATA.

```
"get_data": [{"bucket": "pic", "key": "1.jpg"},
             {"bucket": "pic", "key": "2.jpg"}],
"put_data": [{"bucket": "collage", "key": "col1.jpg"}],
"num_threads": 5
```

Figure 8. Example annotations for an invocation of `collage`.

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, LAMBDATA 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), LAMBDATA can provide all benefits except that it would not overlap functions in a pipeline. If `num_threads` is missing, LAMBDATA assumes the function gets data sequentially when estimating the lead time for scheduling (see §V-A).

IV. LAMBDATA'S ARCHITECTURE

We modified OPENWHISK and added LAMBDATA 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

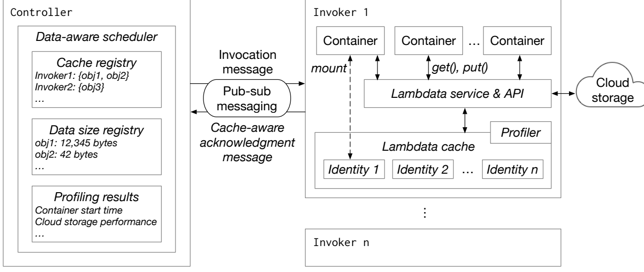

Figure 9. LAMBDATA's architecture. Components with italic font are LAMBDATA-specific.

message, too.

Each `Invoker` exports the LAMBDATA API via the 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, LAMBDATA 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, LAMBDATA'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_{start} and T_{get}, respectively.

A. Estimating the lead time

The start phase. Let us name the function f. We denote by T_{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_{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_{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, LAMBDATA monitors $T_{\text{cold}}(f)$ from the initialization to f's first LAMBDATA API call of the `get` method. If LAMBDATA 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 LAMBDATA monitors the time spent on recent cloud storage requests to track the relationship between T_{data} and object size dynamically. Let $D_c \subseteq D$ be the subset of cached data and $D_a = D \setminus D_c$ the subset of data absent. If $d_i \in D_c$, then the function can read it immediately. Otherwise, if $d_i \in D_a$, 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 $|D_a| \leq n$, 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 $|D_a| > n$, then the time is approximately the total time for getting all data over n threads. Therefore, the time spent in the `get` phase is:

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

Total lead time. As a result, the total lead time is $T = T_{\text{start}} + T_{\text{get}}$. LAMBDATA 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. LAMBDATA 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`. LAMBDATA piggybacks with this message a list of currently cached data, as an array of `(bucket, key, size)` tuples. 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 Invokers 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` $\rightarrow$ `Set[(bucket, key)]`. It also maintains the size of all data as a dictionary of `(bucket, key)` $\rightarrow$ `size`. Both dictionaries may be stale or incomplete, which affects only 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_{warm}, T_{cold}, and T_{data}, the `Invoker` also monitors the time whenever it starts a container or handles a cloud storage operation, and sends them to

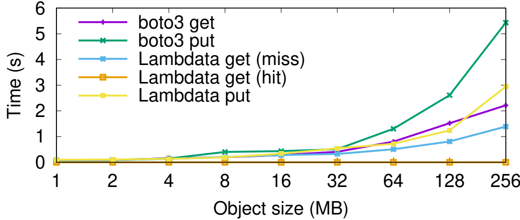

Figure 10. Microbenchmark: median time to get and put objects of various sizes to Amazon S3. Lower is better.

the `Controller` via the acknowledgment message. The `Controller` bookkeeps recent profiling results and interpolates them for estimation.

VI. EVALUATION

We deployed LAMBDATA 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 LAMBDATA 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 LAMBDATA fast handling cloud storage requests?
B. Does LAMBDATA speed up function invocations?
C. Does LAMBDATA speed up multi-function workflows?
D. Does LAMBDATA reduce monetary cost?

A. Microbenchmark

We first evaluate LAMBDATA's performance of basic cloud storage operations. We `get` and `put` data objects from 1MB to 256MB, to Amazon S3, using both BOTO3 [9], the official AWS Python library, and LAMBDATA. Figure 10 shows the median time for each operation.

For the `get` operation, if the data is cached, LAMBDATA takes less than 1ms, because it is simply accessing files on the local disk, and LAMBDATA does not need to do anything. If the data is missing, LAMBDATA'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, LAMBDATA'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 LAMBDATA connects to the cloud storage in the `Invoker`, using a Java runtime outside of containers. Since the LAMBDATA is just a wrapper over the underlying AWS SDK, we do not claim any contribution on the speedups. Nevertheless, the results show that LAMBDATA performs well for basic cloud storage operations.

B. Function performance

To evaluate LAMBDATA'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.*, applying filters) locally on a mobile phone, we implement everything as cloud functions to demonstrate the feasibility. We used images from the Div2K dataset [10], [11] 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 the authors' 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 LAMBDATA 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 LAMBDATA with OPENWHISK and found no statistically significant difference if the data is not in the cache. If the data is cached, LAMBDATA 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 LAMBDATA offers significant speedup over existing serverless clouds.

2) Case study: In order to understand how data matters to LAMBDATA'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

Table III
LIST OF ALL FUNCTIONS AND THE PARAMETERS USED IN THE EXPERIMENT.

Function	Descsription	Parameters used in this experiment
App 1: Photo sharing		
`malware`	Scan a file for malware, using `yextend` [12].	Used malware rules form `BinaryAlert` [13].
`compress`	Compress an image, using `Pillow` [14].	Output JPEG quality = 75%.
`thumbnail`	Generate a thumbnail of an image, using `Pillow`.	Thumbnail size = 320 × 320.
`image_filter`	Apply a filter on an image, using `Pillow` and `numpy` [15].	Applied an Instagram "Amaro"-like filter.
`create_story`	Generate a video from a list of images, using `OpenCV` [16].	1920 × 1080 M-JPEG, 5 seconds per imaage.
`add_text`	Add a text label to a video, using `OpenCV`.	Added a label with random text.
`add_audio`	Add an audio track to a video, using `FFmpeg` [17].	Used a pop music track in MP3 format.
`transcoding`	Convert a video to another codec, using `FFmpeg`.	Transcoded into the `msmpeg4v2` format.
`video_filter`	Apply a filter on a video, using `OpenCV`.	Applied a cartoon-like filter.
`publish`	Publish an image or a video into a dedicated bucket.	Bookkeeping only, no computation on data.
App 2: Online classroom		
`lecture_note`	Compile a lecture note in LaTeX `beamer` to PDF.	The lecture note had 10 slides.
`merge_notes`	Merge a list of PDF files into one PDF.	Merged 10 lecture notes.
`watermark`	Add a watermark to all pages of a PDF file.	Used "lecture notes" as the watermark.
`split_note`	Split a PDF file into two files (slides and speaker notes).	Splitted a 10-page PDF into two 5-page PDFs.
`write_homework`	Compile a LaTeX homework document to PDF.	The PDF had three questions, one per page.
`grade_homework`	Generate per-question PDF files for all submissions.	Used three questions, 20 student submissions.
`question_pool`	Create a question pool for an exam from a PDF repository.	Randomly chose 5 out of 15 questions.
`make_exam`	Generate per-student problem set from the question pool.	Randomly chose 3 out of 5 questions.
`answer_exam`	Compile a LaTeX document and attach it to the exam PDF.	The document had 10 pages.
`grade_exam`	Attach a grade on each page of a PDF file.	The document had 10 pages.

Table IV
BREAKDOWN OF THE PHASES IN EACH FUNCTION (IN MILLISECONDS).

Function	get	compute	put	speedup
App 1: Photo sharing				
`malware`	208	69	0	3.99×
`compress`	216	117	83	2.08×
`thumbnail`	201	79	48	2.58×
`image_filter`	130	8129	111	1.02×
`create_story`	651	865	418	1.51×
`add_text`	353	938	392	1.27×
`add_audio`	419	70	249	2.31×
`transcoding`	414	591	154	1.56×
`video_filter`	398	7600	461	1.05×
`publish`	299	0	423	1.71×
App 2: Online classroom				
`lecture_note`	116	1345	122	1.08×
`merge_notes`	615	1436	145	1.39×
`watermark`	129	513	116	1.20×
`split_note`	65	89	157	1.27×
`write_homework`	98	1470	123	1.06×
`grade_homework`	711	904	311	1.58×
`question_pool`	101	83	102	1.54×
`make_exam`	129	61	114	1.74×
`answer_exam`	115	1340	143	1.08×
`grade_exam`	133	856	139	1.13×
Geometric mean				1.50×

OPENWHISK and LAMBDATA's run time in each phase.

We pre-warmed the container and ran `thumbnail` with input images of four popular dimensions: 1024×768 (web quality, 460KB), 1920×1080 (full HD, 1.2MB), 3840×2160 (4K UHD, 4.7MB), and 4032×3024 (12 megapixel iPhone photo, 7MB). Figure 11 shows the timeline of each function invocation. Each cluster represents an image dimension, of which the top bar is the timeline of OPENWHISK, and the bottom bar is the timeline of LAMBDATA.

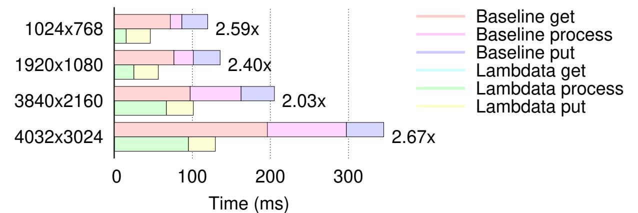

Figure 11. Time breakdown and speedup of the thumbnail function for various image sizes.

We observe that all phases' run time increase with the image dimension, but the ratio of these increases is non-linear. 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 non-linear 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 to 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

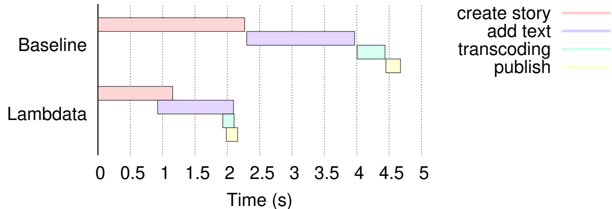

Figure 12. Timeline for the slideshow workflow.

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.

[2]We did not use AWS's billing statement because it was too coarse-grained.

Table V
MONETARY COST. NUMBERS ARE IN $\times 10^{-6}$ DOLLARS.

	Baseline	LAMBDATA	Savings
Serverless cost	40.8	24.1	
Storage cost	21.6	20.4	
Total	62.4	44.5	28.6%

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 [18], [19], [20] is a technique that reuses prior computation results of pure functions. Nectar [21] manages data and computation in the traditional data center setting. It memoizes intermediate computation results of Dryad programs. Incoop [22] 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 [23], [24], [25]. For example, Oozie [26] manages workflows for Hadoop systems, and Yu *et al.* [27] schedules workflows for grid computing. Unfortunately, their models do not fit in the scope of serverless programming. Cloud bursting schedulers, such as [28], only consider occasional workload offloading rather than general cloud computing use cases. DEWE v3 [29] employs a hybrid execution model on serverless computing, but it focuses on scientific workflows and resource underutilization, different from LAMBDATA's goal of data-awareness.

Serverless function orchestration. AWS Step Functions [30] 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 [31] and Azure Durable Functions [32] 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 LAMBDATA, 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 LAMBDATA 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 LAMBDATA, 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 LAMBDATA achieves an average of 1.51× speedup on the turnaround time and reduces monetary cost by 16.5%. All source code of LAMBDATA and benchmarking applications we wrote are at https://columbia.github.io/lambdata.

REFERENCES

[1] K. Corless, M. Kavis, and K. Norton, *NoOps in a serverless world*, https://www2.deloitte.com/insights/us/en/focus/tech-trends/2019/noops-serverless-computing-transforming-it-operations.html.

[2] Cloud Foundry, *Where PaaS, Containers and Serverless Stand in a Multi-Platform World*, https://www.cloudfoundry.org/multi-platform-trend-report-2018/.

[3] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *OSDI*, 2018.

[4] Amazon Web Services, *AWS Lambda Developer Guide: Programming Model*, https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html.

[5] Google, *Cloud Functions Execution Environment*, https://cloud.google.com/functions/docs/concepts/exec.

[6] Apache OpenWhisk, *Open Source Serverless Cloud Platform*, https://openwhisk.apache.org.

[7] IBM Cloud, *How Cloud Functions works*, https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-about.

[8] Apache Kafka, *A distributed streaming platform*, https://kafka.apache.org/.

[9] Boto3, https://aws.amazon.com/sdk-for-python/.

[10] E. Agustsson and R. Timofte, "NTIRE 2017 challenge on single image super-resolution: Dataset and study," in *CVPR Workshops*, 2017.

[11] R. Timofte, E. Agustsson *et al.*, "NTIRE 2017 challenge on single image super-resolution: Methods and results," in *CVPR Workshops*, 2017.

[12] yextend, https://github.com/BayshoreNetworks/yextend.

[13] BinaryAlert, https://github.com/airbnb/binaryalert.

[14] Pillow, https://pillow.readthedocs.io/en/stable/.

[15] NumPy, https://www.numpy.org.

[16] OpenCV, https://opencv.org.

[17] FFmpeg, https://ffmpeg.org.

[18] Y. A. Liu, S. D. Stoller, and T. Teitelbaum, "Static caching for incremental computation," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, May 1998.

[19] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *POPL*, 1989.

[20] D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, pp. 19–22, Apr. 1968.

[21] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *OSDI*, 2010.

[22] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *SoCC*, 2011.

[23] A. K. Bardsiri and S. M. Hashemi, "A review of workflow scheduling in cloud computing environment," 2012.

[24] Z. hui Zhan, X. F. Liu, Y. jiao Gong, J. Zhang, H. S. hung Chung, and Y. Li, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Comput. Surv.*, vol. 47, pp. 63:1–63:33, 2015.

[25] O. Alqaryouti and N. Siyam, "Serverless computing and scheduling tasks on cloud: A review," *American Scientific Research Journal for Engineering, Technology, and Sciences*, vol. 40, pp. 235–247, 2018.

[26] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: Towards a scalable workflow management system for hadoop," in *SWEET*, 2012.

[27] J. Yu, R. Buyya, and K. Ramamohanarao, *Workflow Scheduling Algorithms for Grid Computing*, 2008, pp. 173–214.

[28] Y. C. Lee and B. Lian, "Cloud bursting scheduler for cost efficiency," *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 774–777, 2017.

[29] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *ICSOC*, 2017.

[30] Amazon Web Services, *AWS Step Functions*, https://aws.amazon.com/step-functions/.

[31] IBM, *Composer*, https://github.com/ibm-functions/composer.

[32] Microsoft, *About Durable Functions*, https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.