Tracing Function Dependencies Across Clouds

Wei-Tsung Lin, Chandra Krintz, Rich Wolski
Dept. of Computer Science, UC Santa Barbara

***Abstract*—In this paper, we present *Lowgo*, a cross-cloud tracing tool for capturing causal relationships in serverless applications. To do so, *Lowgo* records dependencies between functions, through cloud services, and across regions to facilitate debugging and reasoning about highly concurrent, multi-cloud applications. We empirically evaluate *Lowgo* using microbenchmarks and multi-function and multi-cloud applications. We find that *Lowgo* is able to capture causal dependencies with overhead that ranges from 2-12%, which is less than half that of the best-performing, cloud-specific approach.**

I. Introduction

To accelerate and simplify application development for the next generation of Internet-of-Things (IoT), mobile, and web applications, a new, event-based, cloud programming and execution model has emerged. Using this model, developers upload arbitrary computations as simple functions to a cloud-hosted service along with a specification of the inputs and conditions under which each function should be triggered. The facility for hosting and managing these functions is termed "Functions as a Service" (FaaS), and colloquially, as "serverless" computing since developers need not provision virtual servers or software stacks directly for their functions – the FaaS platform handles all the complexity associated with server management.

Serverless application functions are short-lived, stateless, and ephemeral (executing only in response to events), and they use cloud datastore services for persistence. A subset of cloud services act as event sources (triggers), including datastore and queue updates, notifications, log activity, HTTP requests, blob/object store updates, and function invocations (scheduled or triggered by other functions). Serverless platforms place limits on the execution duration, memory use, request/response payload size, local temporary disk space, code storage, and other resources that functions use [1] to achieve simplicity, scale, and system stability. Users pay only for the resources their functions use and the system automatically scales the number of function executions according to event load.

Because of its low cost and simplified model, serverless is increasingly employed for a wide range of applications, from websites, web APIs, test frameworks, and log analysis, to large-scale batch and streaming data fusion, transformation, and analytics [2], [3], [4], [5]. Given the popularity of this model, all public cloud providers (and many private cloud systems) offer serverless platforms with similar functionality [6], [7], [8], [9], [10].

Despite this interest and availability, programming tools and debugging support for serverless applications are nascent. Because serverless applications can consist of thousands of concurrent functions, tools are needed to track their interdependencies and to identify when problems occur and why. Few tools exist that provide such tracking, and those that do, are cloud-specific, suffer data loss, and only capture performance metrics [11], [12], [13].

To address this need, we investigate cross-cloud, end-to-end causal order tracking for serverless applications, called *Lowgo* (an acronym for **Lo**gging for the **W**ide-area in **Go**). Like previous distributed event logging systems [14], [15], [16], *Lowgo* produces a distributed, eventually consistent, partially ordered log of events. However, it also tracks and captures event dependencies, both explicitly between functions and through cloud service "triggers" that invoke functions as a result of a native cloud service request. Causality is an important building block employed in concurrent and distributed systems [17], [18], [19], [20]. However, the lack of support for causal event tracking in serverless settings limits the degree to which developers can understand their applications and identify the root cause of errors, performance bottlenecks, and potential cost optimizations.

To track causal dependencies efficiently across clouds, a *Lowgo* instance operates a multi-stage pipeline co-located in each cloud with serverless functions. Multiple *Lowgo* instances interoperate transparently across clouds to share a uniform view of the causal dependencies within an application. Functions report events to their local instance, which propagates event records and causal relationships to instances in other clouds. *Lowgo* reorders event records based on causal dependencies specified by developers as part of application configuration and deployment, using this per-instance pipeline, to maintain a geo-replicated, causally consistent log across clouds.

We empirically evaluate *Lowgo* using serverless microbenchmarks and multi-function and multi-cloud serverless applications. In our tests, *Lowgo* is able to capture causal dependencies with minimal overhead. The overhead that *Lowgo* introduces ranges from 2-12% which is less than half that of a competitive, single-cloud approach for AWS [13]. We find that the overhead is proportional to the number of events;

short-running applications tend to have higher overhead, while computation-heavy applications typically have lower overhead. The throughput of *Lowgo* ranges from 109K to 30K records per second, depending on dependency depth.

II. Background and Related Work

Serverless computing is a cloud execution model in which the cloud platform manages the underlying machine resources (servers) for developers. Typically, serverless applications consist of fine-grained, short-lived functions that are invoked by the platform in response to system-wide events (e.g. storage updates, notifications, messages received, changes in state, custom events, etc.). For this reason, serverless is also referred to as Functions-as-a-service (FaaS). We consider two popular, public cloud serverless offerings in this work: AWS Lambda and Microsoft Azure Functions.

AWS Lambda is a serverless platform for functions written in Python, Java, C#, and Node.js. The AWS Lambda runtime has built-in AWS SDK support, which developers use to invoke other AWS services. Lambda functions can be triggered by updates to the Simple Object Storage(S3) and DynamoDB, Simple Notification Service (SNS) events, and HTTP requests, among others. The Lambda platform deploys user functions using extensible, isolated Linux containers equipped with essential runtime and library support.

Microsoft Azure Functions provides a similar platform for the Azure cloud. The Azure Function App enables developers to upload function code bodies to the service and interact with Azure services. The officially supported runtimes are C# and JavaScript, while Python and other languages are available as experimental options. When creating a Function App, users choose between Windows NT or Linux (preview) instances to host their function(s). We use Windows NT for this work. Azure Functions can be triggered by updates to Azure Blob Storage, Cosmos DB, HTTP requests, and Queue Storage, among other services.

A. Serverless Tracing Systems

There are multiple tools for tracking the performance of serverless applications and tracing their interdependencies. X-Ray [11] is a tracing tool for AWS that samples the entry and exit of Lambda function instances using unique trace identifiers. It records function duration and times SDK calls and HTTP accesses that a function makes. This data is sent to an X-Ray logging service via UDP. The X-Ray logging service visualizes and presents data to developers as logs and dependency trees, called service graphs. Google employs a similar tracing tool for its cloud-wide services called Dapper [12].

Both X-Ray and Dapper can infer relationships between serverless functions but cannot capture causal order of events. Causality is an important tool employed in concurrent and distributed systems that enables developers to reason about, analyze, and draw inferences from their applications [19], [21], [22]. Dapper and X-Ray are unable to capture causal dependencies in serverless applications because (i) they only sample events, missing uncommon and rare events, and (ii) they do not trace *through* the services that functions access (which trigger other functions). For example, if a function F writes to an S3 bucket which then triggers function G, X-ray (and Dapper in the Google cloud) only record the relationship between F and S3 (or F and Google Cloud Storage (GCS) in Google). They do not capture the dependency chain from F to S3 to G (or F to GCS to G) so G appears unrelated.

GammaRay [13] attempts to solve these problems for AWS Lambda. It captures causal dependencies in Lambda applications by profiling events. GammaRay uses X-Ray to extract (sampled) performance data for each event and DynamoDB Streams to record their dependencies. By doing so, GammaRay is able to capture causal relationships and timings for AWS Lambda functions across serverless applications and AWS regions, as well as through AWS services. That is, in the example above, GammaRay captures the dependency from F to S3 to G. Unfortunately, GammaRay is tightly coupled to AWS and cannot be used across clouds. Moreover, it introduces significant overhead (12-43%) on Lambda function performance (for synchronous writes to DynamoDB via the Streams functionality).

B. Causal Ordering for Non-Serverless Settings

Causal log ordering is a popular subject of research in non-serverless settings. Chariots is a distributed logging service [14] designed for multi-datacenter settings. It supports multiple datacenters by providing a global shared log that consists of all records generated at all datacenters. The log records are maintained by a group of *log maintainers*, which span multiple datacenters and collectively persist to a single shared log. Clients send records to a local Chariots instance, which records and replicates them to other datacenters. In the process, geo-replication preserves the causal order of records. When persisted in a log maintainer, a record is assigned a total order ID. All replicas of the record share the same total order ID. Chariots uses this ID to order records from the same datacenter.

Different from other geo-replicated shared logs such as Google Megastore [23], Chariots does not require all clients to write to the head of log, thus eliminating the single point of contention. In addition, Chariots' pipeline design allows each stage to be scaled seamlessly and independently. This design enables the system to adapt to performance bottlenecks automatically.

Similar logging systems include X-trace [24], Kronos [25], LogBase [16], and Corfu [15]. X-trace reconstructs Internet services dependency trees. X-Trace ap-

pends metadata to network operations and propagates them across layers and applications. X-Trace requires a specialized TCP/IP stack and application instrumentation for use. Kronos tracks dependencies and provides time-ordering for distributed applications. Kronos builds and maintains a service dependency graph internally. It relies on a centralized implementation and was not designed for cross-cloud use. LogBase is a multi-version log-structured database. It adopts an append-only approach to eliminate write bottlenecks. It reduces write overhead by appending all write operations to the head of the log. Corfu is a similar append-only shared log that is built on distributed flash devices. Corfu maintains a static mapping between log position and flash page. Clients ask a sequencer to determine the next available position of the log.

III. *Lowgo*

Lowgo is a cloud-agnostic *casual event tracing* system for multi-function, multi-cloud serverless applications. *Lowgo* captures causal dependencies across functions and through cloud services. It automatically infers dependencies within applications by *Lowgo* capturing calls that functions make via cloud software development kits (SDKs), which access cloud services that are potential event sources or to invoke other functions. Alternatively, developers can specify a subset of such dependencies to trace as part of application deployment using *Lowgo* tools.

Key to the *Lowgo* design is that it avoids synchronized writes, does not block application progress for record sequence assignment, and minimizes cross-cloud communication. To enable this, *Lowgo* comprises multiple instances co-located with serverless functions in different clouds. Functions in one cloud only communicate with their co-located *Lowgo* instance. Records generated in different clouds are replicated across instances by *Lowgo* to compose a consistent, distributed shared log.

Lowgo avoids application delays by determining event position *after* events are reported to *Lowgo*. Functions generate event records that contain event details and dependency information and send them via the *Lowgo* interface. Upon receipt, *Lowgo* reorders records using the dependency information prior to persisting them to the log so that log order reflects causal order.

To enable both causal event tracing and causal log order, *Lowgo* implements a three-stage pipeline similar to that proposed by Chariots. However, *Lowgo* is significantly simpler than Chariots, since it needs only capture a partial order of events (those with detected or with specified dependencies). *Lowgo* also uses a simplified record structure which replaces Chariot's total order ID (which does not capture event dependency) with a record ID and parent ID pair representing the event dependency. We detail both the pipeline and record

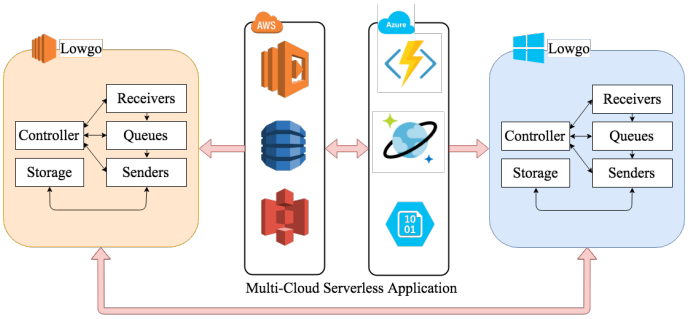

Fig. 1: *Lowgo* overview. The figure shows a distributed application spanning two clouds. In each cloud there is a *Lowgo* instance deployed. When an event generated by cloud service, a corresponding record with causality information is sent to local *Lowgo* instance. Records are replicated across remote *Lowgo* instances with their causal dependencies preserved maintain a consistent record history.

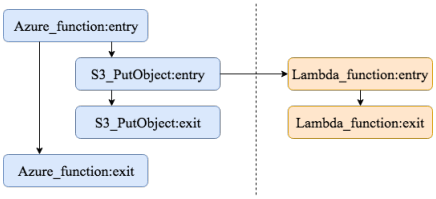

Fig. 2: Multi-cloud serverless application example using Microsoft Azure Functions and AWS Lambda.

structure in the next section. Events pass through the pipeline and are stored in a local data storage in causal order as depicted in Figure 1.

The figure also depicts an example serverless application that executes using both AWS and Azure. Cloud functions access cloud services via cloud SDKs. A subset of service operations also serves as event sources that trigger functions (defined by each serverless platform). Upon invocation of this application, a function in Microsoft Azure writes to an AWS S3 bucket via the AWS SDK. This write triggers a function invocation in AWS Lambda.

To trace events through cloud services and across clouds, *Lowgo* records function entry and exit and each SDK call that a function makes. These records contain information about the event source (for functions), the SDK operation, and the SDK arguments. *Lowgo* records SDK calls (only those that are potential event sources) via minor modifications to each cloud SDK. Function entry/exit instrumentation (function wrapping) occurs during serverless application deployment to each cloud platform via a *Lowgo* tool.

Figure 2 shows the service graph that *Lowgo* records for this application. The arrows indicate causal relationships, i.e. A→B means event A causes event B. The records on the left of the graph are generated in Azure, while the records on the right are generated in AWS. Records generated locally are propagated to

other *Lowgo* instance. All instances observe the same causal relationship graph.

A. Implementation

Events generated by application are represented as records in *Lowgo*. Records contain application context and causal information. Context is used for debugging and analysis; causal information identifies details about the event that caused invocation of the function. Each *Lowgo* record has the following fields:

- **Record ID**: A 128-bit statistically unique ID (UUID v4) that serves as identifier of the record.
- **Host**: An integer that identifies the *Lowgo* instance for which this record is generated. For example, in Figure 2, records generated in Azure have the same host, while records generated in AWS have a different host.
- **Trace ID**: A 128-bit ID shared by a series of records that are part of the same causal event chain. A Trace ID is generated when a function without a parent (i.e. not triggered directly or indirectly by another function) is invoked. A Trace ID is transmitted downstream to all records generated by this *root* function and downstream functions via *Lowgo* SDK use.
- **Parent ID**: The record ID of the parent (i.e. the function that directly or indirectly triggered this function). *Lowgo* ensures that parent records are persisted prior to their child records in the log. This field is empty for a root function.
- **Payload**: Application context information. *Lowgo* allows developers to customize what information is recorded.
- **Log ID**: A unique and monotonically increasing number that represents the record's position in the log. Log ID provides an alternative way to infer causal dependency other than checking each record's parent ID. The order of multiple records which have the same trace ID can be obtained by comparing their log IDs.

Each of these fields, except for Log ID, is filled in as part of *Lowgo* SDK use. Since Log ID assignment (i.e. determining record position in the log) is a complex task, *Lowgo* separates assignment from SDK to keep overhead low. The overhead that *Lowgo* imposes on serverless applications thus includes record construction and round trip communication of records and acknowledgments between functions and the co-located *Lowgo* instance.

Figure 3 illustrates the architecture of a *Lowgo* instance. The solid arrows represent the flow of records and the dotted arrows represent information exchange. A *Lowgo* pipeline isolates intra-datacenter communication, record ordering, and geo-replication using three

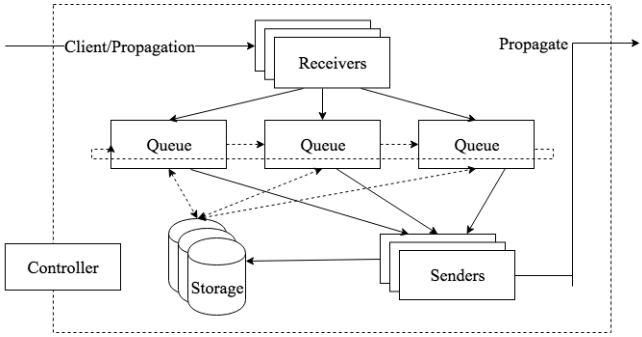

Fig. 3: The architecture of a *Lowgo* instance. The solid arrows show the forwarding path of records. Queues form a ring. A token is passed around the queue ring to update the latest Log ID. Queues also send queries to the storage to see if a buffered records' parent is persistent in storage.

stages: **receiver**, **queue**, and **sender**. Each stage can be scaled independently based on network congestion and workload.

A **Receiver** receives records sent by a local SDK or propagated by other *Lowgo* instances. Application clients can send records to any receiver using either gRPC or HTTP POST. Co-located receiver instances are load balanced via HAProxy. Also, since records are sent individually by each SDK, receiver buffer incoming records to reduce communication overhead. Receivers send batched records to a queue when the buffer fills or is flushed.

The **Queue** orders records and assigns the record position (log ID) to each record. Queues buffer records until they can be sent (in batches) to the next stage. A record is batched when either (i) the record does not have a parent ID or (ii) the record's parent ID record is found in storage. The queue periodically queries *Lowgo* storage for parent IDs of buffered records. If there is a match, the queue assigns a log ID to the buffered record and forwards it to a sender. By doing so, queues ensure that log IDs of records coming from same root function (trace ID) are in order. Thus, the log ID of a dependent record is larger than that of its parent.

For concurrency, multiple queue instances process records simultaneously and receivers can send records to any of queue instances. Queues avoid duplicating log IDs using token passing (a database can be used as a sequencer but doing so introduces a significant bottleneck and single point of failure). The token carries the current maximum log ID for the *Lowgo* instance and passes between queue instances in a ring. Only the queue holding the token assigns and increments the log ID. It assigns log IDs for all of its batched records. The queue then passes the token to its neighbor and communicates the batch to a sender.

The **Sender** persists and geo-replicates records. Instances write records to the log in any order (avoid-

ing synchronization) and can employ log ID sharding for additional scale and performance. Senders send batched copies of records (replicas) to other *Lowgo* instances after removing their log ID. When a record is received by a *Lowgo* instance (receiver stage), the receiver determines whether it is a replica by checking its host ID. Replicated records proceed through the instance's pipeline and receive a new log ID.

Log IDs of the same record across clouds can differ and records from different event chains will be interleaved in the log of each *Lowgo* instance. This approach enables *Lowgo* to maintain causal order via parent IDs across instances (clouds) with very low overhead by enabling *Lowgo* instances to operate independently and concurrently at scale.

The final two components of each *Lowgo* instance are **Storage** and the **Controller**. Storage is where each *Lowgo* physically stores its log records; storage in different *Lowgo* instances are independent and do not interoperate. *Lowgo* storage responds to queries from queues and persists records sent from senders and so can be implemented using any database. Replication is not necessary since it is inherently supported across clouds by *Lowgo*. As mentioned previously, sharding and multi-index support can reduce the overhead of database use if available. The **controller** maintains *Lowgo*'s system-wide configuration. Communication between cross-cloud controllers occurs only at *Lowgo* startup or when resources/instances are added or removed.

B. Lowgo SDK

We provide a *Lowgo* SDK for developers to deploy serverless function with *Lowgo* causal tracing enabled. The *Lowgo* SDK consists of three components: a *Lowgo* instance deployment tool, *Lowgo* client API, and serverless function deployment tool for each cloud. Using these tools in combination enables developers to use *Lowgo* without modifying their serverless applications.

Developers use *Lowgo* instance deployment tool to generate configuration files used to deploy *Lowgo* instance with Docker Swarm. The configuration consists of the number of instances per pipeline stage, debugging flags, and port numbers to use. The tool generates Docker Swarm stack files and configuration files for instances in each cloud. The deployed *Lowgo* instance uses the configuration file to set up the pipeline and load balancing.

The *Lowgo* client API is a simple library that sends records to a *Lowgo* instance. It uses gRPC framework and HTTP POST to send records to *Lowgo* receivers. We choose gRPC for its performance and portability. If a programming language used to implement a serverless function does not have gRPC support, *Lowgo* uses HTTP POSTs. Currently, the *Lowgo* client API is available for Python, Node.js, and Java. Although application modification is not required to use *Lowgo*, developers can add custom tracing via the *Lowgo* client API. The *Lowgo* client module exposes two variables: *traceid* and *parentid*. The serverless application deployed using the *Lowgo* function deployment tool automatically updates these variables so developers can use them to tailor record information.

The *Lowgo* function deployment tool uploads functions to serverless cloud platforms. The tool takes the function source code and libraries, intercepts the function entry point with a *Lowgo* wrapper (to record entry/exit of functions and trace IDs), and constructs a deployment package. *Lowgo* currently supports AWS Lambda and Azure. A *Lowgo* SDK is also integrated within the package and loaded at runtime by the wrapper, for each cloud SDK that the application uses. A *Lowgo* SDK for a particular cloud is the cloud SDK modified to insert dependency information, to build name records, and to send records to a local *Lowgo* instance. The records sent by the *Lowgo* wrapper and SDKs provide the minimal amount of information necessary to enable *Lowgo* to track causal order between functions, through cloud services, and across clouds.

IV. Evaluation

We evaluate *Lowgo* performance using serverless microbenchmarks and applications (multi-function and multi-cloud). We first overview these applications and our experimental methodology, and then present our empirical results.

A. Experimental Methodology

In our evaluation, we co-locate a *Lowgo* instance in each cloud and region in which the serverless functions in an application are deployed. A *Lowgo* instance consists of a Docker Swarm (v17.12.0-ce) with five nodes. Nodes include a *Lowgo* receiver, controller, sender, and two queues. Each *Lowgo* instance also integrates a MongoDB (v2.6.10) database. In our experiments, we deploy Docker Swarm and MongoDB using instance type c4.large in AWS EC2 and our Eucalyptus private cloud, and instance type Standard.D2_v2 in Azure.

We evaluate the overhead of *Lowgo* using three microbenchmarks and three multi-function serverless applications. Functions that run in AWS are written in Python 2.7 and use the boto3 AWS SDK to access AWS services. Azure functions are written in node.js. All other *Lowgo* tools are written in Python 2.7. We execute each experiment 100 times with and without *Lowgo*. We attempt to isolate cold start overhead by running the applications 10 times prior to performing our measurements. Serverless cold starts occur when the system must load both the function and container in which it executes prior to function invocation. Cloud providers maintain the container for a short (unspecified) duration following function invocation (optimizing

for temporal locality), to avoid this overhead for potential repeat invocations.

The microbenchmarks are single function applications and include a function that simply returns (EMPTY), a function that performs a gRPC request with a payload of 1125 bytes (gRPC_SDK), and a function that performs an HTTP POST with a payload of 1125 bytes (REST_SDK). We use these applications to isolate the overhead of *Lowgo* SDK use.

The three multi-function applications are called `Map-Reduce`, `Rekognition`, and `Thumbnail`. Map-Reduce is an application implemented by AWS engineers and is based on one of the Big-Data-Benchmark programs. This application implements the map-reduce protocol via AWS Lambda and S3. The dataset is *pavlo/text/1node/uservisits*, which contains 24.4GB data of IP addresses that have visited particular websites. We place the dataset in S3 in the same region as the Lambda functions that accesses it (US West). The application invokes 29 mapper Lambda functions, each of which operates on a different section of the input. Each mapper function reads its data from S3, counts the number of access per IP prefix for a range of IPs and stores the results in S3. A coordinator function is triggered by this S3 write. The coordinator checks if all mapper functions have finished, i.e. all 29 partial results are available in S3. When all mappers complete, coordinator function invokes the reducer function, which downloads the partial results and performs a reduction over them to produce the final per-IP count which it stores in S3. We evaluate this application with and without *Lowgo* support using a single cloud (AWS).

Rekognition uses the AWS image processing service called Rekognition, to label images. A function is triggered when a file with jpg suffix is uploaded to S3 bucket in its region. The function calls AWS Rekognition API, which returns labels and probabilities that describe the image. The function uploads the result to an AWS DynamoDB table. We use a 152KB 640*427 jpeg file to trigger the application. We evaluate this application with and without *Lowgo* support using a single cloud (AWS).

Thumbnail is a multi-cloud application. We deploy a Python script in a private cloud instance that uploads images to AWS S3. The S3 upload event triggers a function in AWS, which creates a thumbnail of the uploaded image. The function uploads the thumbnail to Azure Blob Storage. This Blob update triggers an Azure function, which loads the image from Blob Storage module and measures the time for doing so. We use a 2.6MB (6000x4000) jpeg file as input. The resulting jpeg thumbnail size is 12.5KB (300x200).

B. Performance With and Without Lowgo

We first evaluate the overhead of using the *Lowgo* SDK by measuring the time to send records to *Lowgo*.

	AWS Time	**AWS Memory**
EMPTY	0ms	19MB
gRPC_SDK	2085ms	36MB
REST_SDK	2391ms	27MB

TABLE I: *Lowgo* microbenchmark average duration and memory use in AWS.

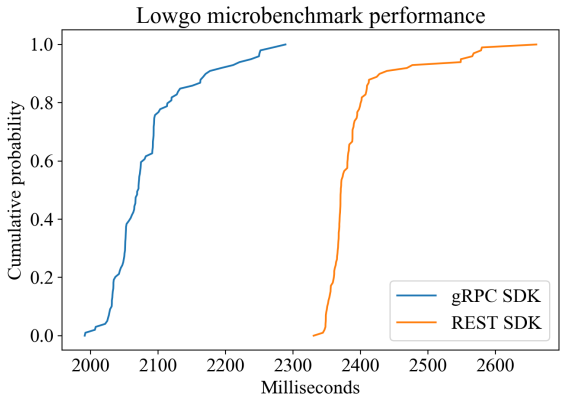

Fig. 4: *Lowgo* microbenchmark performance CDFs (gRPC_SDK and REST_SDK). The means are 2085ms and 2391ms, respectively.

For this test, we employ the serverless microbenchmarks and AWS. We present the average execution times in milliseconds (ms) and memory used in megabytes (MB) in Table I for each microbenchmark. To measure time, we insert timers at function entry and exit; the function logs the duration prior to exiting. The *Lowgo* tools collect, aggregate, and summary these log entries, as well as the timings and memory usage recorded by the service (which is CloudWatch in AWS) and used for billing.

Since EMPTY simply returns upon being invoked, its duration is 0ms, the durations of other two applications translate to the cost of sending 100 records using different protocols. It takes an average of 2085ms to send 100 records using gRPC protocol, and 2391ms to send 100 records using HTTP POST, which translates to approximately 21ms and 24ms to send a record to *Lowgo* using gRPC and HTTP POST, respectively. Figure 4 shows the full distribution of times as empirical cumulative distribution functions (CDFs) for both tests. The gRPC SDK test has a standard deviation of 63ms with 1991ms minimum and 2289ms maximum. The REST SDK test has a standard deviation of 58ms with 2331ms minimum and 2661ms maximum. In terms of memory use, gRPC uses 17MB, and HTTP uses 8MB, respectively, over EMPTY. The difference between *Lowgo* and CloudWatch log measurements represents the function setup time. This difference is 6.3ms (stdev=8.7ms) for gRPC_SDK and 0.48ms (stdev=1.70ms) for REST_SDK.

We next investigate the dollar cost of importing the *Lowgo* SDK. We find that AWS Lambda only imports modules during cold starts (when a container is not reused). In such cases, we also find that Amazon does not charge for codes executed outside Lambda handler

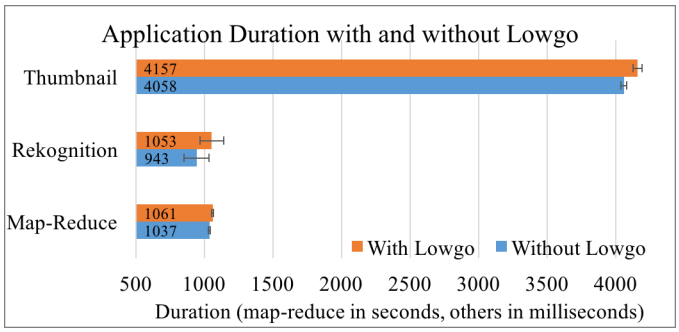

Fig. 5: Average application performance with and without *Lowgo*. Error bars show the 95% confidence intervals.

when under 10 seconds. Since the time Lambda takes to import *Lowgo* SDK is well under 10 seconds threshold, importing the *Lowgo* SDK does not introduce extra cost. The average time to import *Lowgo* SDK is 128ms for gRPC and 61ms for REST in AWS Python 2.7, and 13ms for REST in Node.JS 6.10.

C. Application Performance

We next evaluate the overhead imposed by *Lowgo* for the multi-function applications. Figure 5 shows the average execution time for each with and without *Lowgo*. For Map-Reduce, *Lowgo* introduces an average overhead of 24 seconds which is 2.3%. This difference is small but statistically significant according to a t-test with $\alpha = 0.05$.

For Rekognition, *Lowgo* adds an average of 110ms (11.7%). Because this application is very short running and invokes multiple services that are potential event triggers instead of computing, *Lowgo* SDK overhead plays larger role in overall execution time. There are 6 records sent to *Lowgo* by each application instance. Using our microbenchmark results, we expect an average overhead of 126ms, which is inline with what we observe for this application. Specifically, Figure 6 shows the histogram of execution times with and without *Lowgo* for Rekognition. Note the shift of approximately 100ms with and without *Lowgo*.

Next, we evaluate how *Lowgo* performs in multi-cloud setting using the Thumbnail application. Figure 7 shows the results. The average total overhead introduced by *Lowgo* is 99ms or 2.4% over the uninstrumented version. According to a t-test with $\alpha = 0.05$, the difference is statistically significant.

Summarizing, *Lowgo* introduces an average of between 2 and 12% overhead for the applications we study. This is significantly lower than the competitive AWS-only approach (GammaRay [13]) which introduces 12 to 43% overhead for similar applications. The benefits come from *Lowgo*'s multi-stage pipeline for log ID assignment and record persistence instead of using GammaRay's synchronized, inlined call to DynamoDB with DynamoDB Streams support.

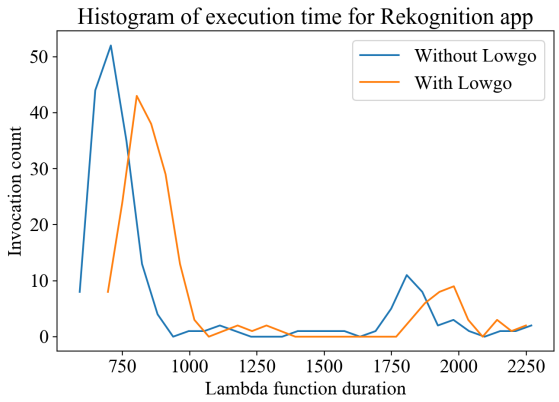

Fig. 6: A histogram of execution time for Rekognition with and without *Lowgo*.

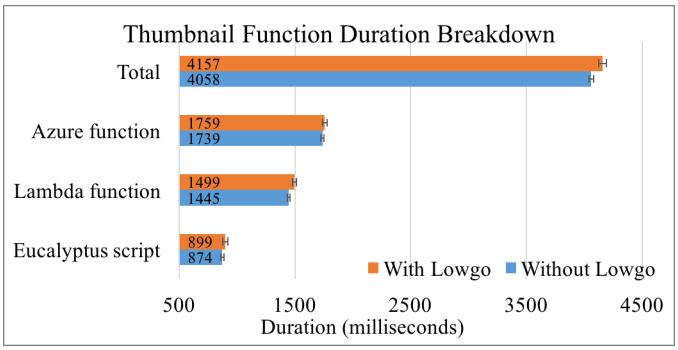

Fig. 7: Time spent in each component on average for the Thumbnail application with and without *Lowgo*. Error bars show the 95% confidence intervals.

D. Lowgo Throughput

To understand the impact of causal dependencies between functions on *Lowgo* performance, we next evaluate system throughput. We perform this test using AWS EC2. We launch 6 c4.large EC2 instances in US West to form a Docker swarm, which consists of 1 manager node and 5 worker nodes. Each node is responsible for hosting a *Lowgo* stage: controller, receiver, sender, while there are 2 queues hosted on 2 nodes. The remaining node is responsible for hosting MongoDB. In addition to the Docker swarm nodes, we launch a c4.large instance in the same region to drive the throughput benchmark.

Figure 8 shows *Lowgo* throughput for different dependency depths. We define dependency depth as the number of event records with same root cause. An event chain with dependency depth 1 means that an event is independent. If one event causes another event, the event chain has dependency depth 2, and so on. When there is no dependency among records, *Lowgo* throughput is 106 kilo-records per second (Krps) compared to 142 Krps for a version of Chariots we implemented. Recall that Chariots does not capture dependencies so this difference can be considered the "base" overhead cost of dependency

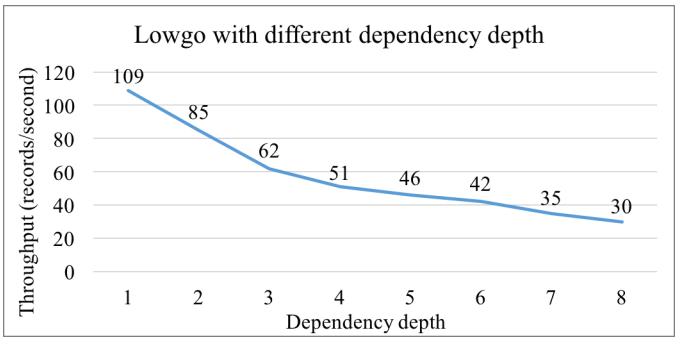

Fig. 8: *Lowgo* throughput. The X-axis is dependency depth. The Y-axis is records per second.

recognition in *Lowgo* versus a top-performing logging system without dependency tracking.

As dependency depth grows, *Lowgo* throughput decreases slowly. This is caused by the dependency resolving mechanism in the queue stage. In this stage in Chariots, the system checks whether the maximum total order ID is greater than the buffered parent total order ID to decide if a buffered record can be appended to storage. In this stage in *Lowgo,* the system checks whether buffered records' parent record is in storage. The MongoDB query operation to perform this check results in the overhead introduced by *Lowgo* for dependency resolution. As part of future work, we are considering ways to reduce this overhead through the use of more optimized mechanisms for *Lowgo* local data persistence.

V. Conclusions

Serverless computing is an emerging computing model that facilitates the development of scalable distributed application. In this paper, we present a tracing system for serverless, called *Lowgo,* for multi-cloud deployments. *Lowgo* traces application dependencies through services and across clouds and automatically integrates itself into serverless applications as part of the deployment process. We evaluate *Lowgo* using microbenchmarks and applications and find that the overhead introduced by *Lowgo* ranges from 2-12%. As part of future work, we are investigating data storage options for precedent records querying and to improve system throughput.

Acknowledgments. This work is funded in part by NSF (CNS-1703560, OAC-1541215, CCF-1539586, CNS-1218808, ACI-1541215), ONR NEEC (N00174-16-C-0020), and the Huawei Corporation.

References

[1] "AWS Lambda Limits," http://docs.aws.amazon.com/lambda/latest/dg/limits.html, [Online; accessed 15-Nov-2016].

[2] Contino, "5 killer use cases for aws lambda," Sep 2017.

[3] "A guide to serverless computing with aws lambda," Feb 2017. [Online]. Available: http://www.cuelogic.com/blog/a-guide-to-serverless-computing-with-aws-lambda/

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in SOCC, 2017.

[5] N. K. Mukhi, S. Prabhu, and B. Slawson, "Using a serverless framework for implementing a cognitive tutor: Experiences and issues," in Intl Workshop on Serverless Computing, 2017.

[6] "AWS Lambda," https://aws.amazon.com/lambda/, [Online; accessed 15-Nov-2016].

[7] "Google Cloud Functions," https://cloud.google.com/functions/docs/, [Online; accessed 15-Nov-2016].

[8] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, [Online; accessed 15-Nov-2016].

[9] "IBM OpenWhisk," https://developer.ibm.com/openwhisk/, [Online; accessed 15-Nov-2016].

[10] "Iron.io," https://www.iron.io, [Online; accessed 15-Nov-2016].

[11] "AWS X-Ray," https://aws.amazon.com/xray/, [Online; accessed 11-September-2017].

[12] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[13] W.-T. Lin, C. Krintz, R. Wolski, and M. Zhang, "Tracking Causal Order in AWS Lambda Applications," in IEEE International Conference on Cloud Engineering, 2018.

[14] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A scalable shared log for data management in multi-datacenter cloud environments." in EDBT, 2015, pp. 13–24.

[15] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," in USENIX NSDI, 2012.

[16] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "Logbase: a scalable log-structured database system in the cloud," VLDB, vol. 5, no. 10, 2012.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, Jul. 1978.

[18] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," Distributed Computing, vol. 9, no. 1, Mar 1995.

[19] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," Distrib. Comput., vol. 7, no. 3, Mar. 1994.

[20] N. M. Chakarat Skawratananond and V. K. Garg, "A Lightweight Algorithm for Causal Message Ordering in Mobile Computing Systems," 1999, "http://www.utdallas.edu/ neerajm/publications/conferences/causal.pdf" Accessed 15-Sep-2017.

[21] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," Inf. Process. Lett., vol. 39, no. 6, Oct. 1991.

[22] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in International Workshop on Distributed Algorithms, 1989.

[23] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in Proceedings of the Conference on Innovative Data system Research (CIDR), 2011, pp. 223–234. [Online]. Available: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[24] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in USENIX NSDI, 2007.

[25] R. Escriva, A. Dubey, B. Wong, and E. Sirer, "Kronos: The design and implementation of an event ordering service," in European Conference on Computer Systems, ser. EuroSys '14, 2014.