Fault-tolerant and Transactional Stateful Serverless Workflows

Haoran Zhang, Adney Cardoza[†], Peter Baile Chen, Sebastian Angel, and Vincent Liu
University of Pennsylvania [†]*Rutgers University-Camden*

Abstract

This paper introduces Beldi, a library and runtime system for writing and composing fault-tolerant and transactional stateful serverless functions. Beldi runs on existing providers and lets developers write complex stateful applications that require fault tolerance and transactional semantics without the need to deal with tasks such as load balancing or maintaining virtual machines. Beldi's contributions include extending the log-based fault-tolerant approach in Olive (OSDI 2016) with new data structures, transaction protocols, function invocations, and garbage collection. They also include adapting the resulting framework to work over a federated environment where each serverless function has sovereignty over its own data. We implement three applications on Beldi, including a movie review service, a travel reservation system, and a social media site. Our evaluation on 1,000 AWS Lambdas shows that Beldi's approach is effective and affordable.

1 Introduction

Serverless computing is changing the way in which we structure and deploy computations in Internet-scale systems. Enabled by platforms like AWS Lambda [2], Azure Functions [3], and Google Cloud Functions [18], programmers can break their application into small functions that providers then automatically distribute over their data centers. When a user issues a request to a serverless-based system, this request flows through the corresponding functions to achieve the desired end-to-end semantics. For example, in an e-commerce site, a user's purchase might trigger a product order, a shipping event, a credit card charge, and an inventory update, all of which could be handled by separate serverless functions.

During development, structuring an application as a set of serverless functions brings forth the benefits of microservice architectures: it promotes modular design, quick iteration, and code reuse. During deployment, it frees programmers from the prosaic but difficult tasks associated with provisioning, scaling, and maintaining the underlying computational, storage, and network resources of the system. In particular, app developers need not worry about setting up virtual machines or containers, starting or winding down instances to accommodate demand, or routing user requests to the right set of functional units—all of this is automated once an app developer describes the connectivity of the units.

A key challenge in increasing the general applicability of serverless computing lies in correctly and efficiently composing different functions to obtain nontrivial end-to-end applications. This is fairly straightforward when functions are stateless, but becomes involved when the functions maintain their own state (e.g., modify a data structure that persists across invocations). Composing such *stateful serverless functions* (SSFs) requires reasoning about consistency and isolation semantics in the presence of concurrent requests and dealing with component failures. While these requirements are common in distributed systems and are addressed by existing proposals [8, 28, 33, 35, 46], SSFs have unique idiosyncrasies that make existing approaches a poor fit.

The first peculiarity is that request routing is stateless. Approaches based on state machine replication are hard to implement because a follow-up message might be routed by the infrastructure to a different SSF instance from the one that processed a prior message (e.g., an "accept" routed differently than its "prepare"). A second characteristic is that SSFs can be independent and have sovereignty over their own data. For example, different organizations may develop and deploy SSFs, and an application may stitch them together to achieve some end-to-end functionality. As a result, there is no component in the system that has full visibility (or access) to all the state. Lastly, SSF workflows (directed graphs of SSFs) can be complex and include cycles to express recursion and loops over SSFs. If a developer wishes to define transactions over such workflows (or its subgraphs), *all* transactions (including those that will abort) must observe consistent state to avoid infinite loops and undefined behavior. This is a common requirement in transactional memory systems [20, 23, 32, 37, 38], but is seldom needed in distributed transaction protocols

To bring fault-tolerance and transactions to this challenging environment, this paper introduces *Beldi*, a library and runtime system for building workflows of SSFs. Beldi runs on existing cloud providers without any modification to their infrastructure and without the need for servers. The SSFs used in Beldi can come from either the app developer, other developers in the same organization, third-party open-source developers, or the cloud providers. Regardless, Beldi helps to stitch together these components in a way that insulates the developer from the details of concurrency control, fault tolerance, and SSF composition.

A well-known aspect of SSFs is that even though they can persist state, this state is usually kept in low-latency NoSQL databases (possibly different for each SSF) such as DynamoDB, Bigtable, and Cosmos DB that are already fault tolerant. Viewed in this light, SSFs are clients of scalable fault-tolerant storage services rather than stateful services themselves. Beldi's goal is therefore to guarantee *exactly-once* semantics to workflows in the presence of clients (SSFs) that fail at any point in their execution and to offer *synchro-*

nization primitives (in the form of locks and transactions) to prevent concurrent clients from unsafely handling state.

To realize this vision, Beldi extends Olive [36] and adapts its mechanisms to the SSF setting. Olive is a recent framework that exposes an elegant abstraction based on logging and request re-execution to clients of cloud storage systems; operations that use Olive's abstraction enjoy exactly-once semantics. Beldi's extensions include support for operations beyond storage accesses such as synchronous and asynchronous invocations (so that SSFs can invoke each other), a new data structure for unifying storage of application state and logs, and protocols that operate efficiently on this data structure (§4). The purpose of Beldi's extensions is to smooth out the differences between Olive's original use case and ours. As one example, Olive's most critical optimization assumes that clients can store a large number of log entries in a database's *atomicity scope* (the scope at which the database can atomically update objects). However, this assumption does not hold for many databases commonly used by SSFs. In DynamoDB, for example, the atomicity scope is a single row that can store at most 400 KB of data [14]—the row would be full in less than a minute in our applications.

Beldi also adapts existing concurrency control and distributed commit protocols to support transactions over SSF workflows. A salient aspect of our setting is that there is no entity that can serve as a coordinator: a user issues its request to the first SSF in the workflow, and SSFs interact only with the SSFs in their outgoing edges in the workflow. Consequently, we design a protocol where SSFs work together (while respecting the workflow's underlying communication pattern) to fulfill the duties of the coordinator and collectively decide whether to commit or abort a transaction (§6).

To showcase the costs and the benefits of Beldi, we implement three applications as representative case studies: (1) a travel reservation system, (2) a social media site, and (3) a movie review service. These applications are based on DeathStarBench [12, 16], which is an open-source benchmark for microservices; we have ported and extended these applications to work without servers using SSFs. Our evaluation on AWS reveals that, at saturation, Beldi's guarantees come at an increase in the median request completion time of 2.4–3.3×, and 99th percentile completion time of 1.2–1.8× (§7.4). At low load, the median completion time increase is under 2×.

In summary, Beldi helps developers build fault-tolerant and transactional applications on top of SSFs at a modest cost. In doing so, Beldi simplifies reasoning about compositions of SSFs, runs on existing serverless platforms without modifications, and extends an elegant fault-tolerant abstraction.

2 Background and Goals

In this section, we describe the basics of serverless computing (sometimes known as Function-as-a-Service), the challenge of deploying complex serverless applications that incorporate state, and a list of requirements that Beldi aims to satisfy.

2.1 Serverless functions

Serverless computing aims to eliminate the need to manage machines, runtimes, and resources (i.e., everything except the app logic). It provides an abstraction where developers upload a simple function (or 'lambda') to the cloud provider that is invoked on demand; an identifier is provided with which clients and other services can invoke the function.

The cloud provider is then responsible for provisioning the VMs or containers, deploying the user code, and scaling the allocated resources up and down based on current demand—all of this is transparent to users. In practice, this means that on every function invocation the provider will spawn a new *worker* (VM or container) with the necessary runtime and dispatch the request to this worker ('cold start'). The provider may also use an existing worker, if one is free ('warm start'). Note that while workers can stay warm for a while, running functions are limited by a timeout, after which they are killed. This time limit is configurable (up to 15 min in 1 s increments on AWS, up to 9 min in 1 ms increments on Google Cloud, and unbounded time in 1 s increments on Azure) and helps in budgeting and limiting the effect of bugs.

Serverless functions are often used individually, but they can also be composed into *workflows*: directed graphs of functions that may contain cycles to express recursion or loops over one or more functions. Some ways to create workflows include AWS's *step functions* [41] and *driver functions*. A step function specifies how to stitch together different functions (represented by their identifiers) and their inputs and outputs; the step function takes care of all scheduling and data movement, and users get an identifier to invoke it. In contrast, a driver function is a single function specified by the developer that invokes other functions (similar to the main function of a traditional program). Control flow can form a graph because functions (including the driver function) can be multi-threaded or perform asynchronous invocations.

Stateful serverless functions (SSFs). Serverless functions were originally designed to be stateless. As such, state is not guaranteed to persist between function invocations—even when writing to a worker's local disk, the function's context can be terminated as part of dynamic resource management, or load balancing might direct follow-up requests to different or new instances. Accordingly, a common workaround to persist data is to store it in fault-tolerant low-latency NoSQL databases. For example, AWS Lambdas can persist their state in DynamoDB, Google cloud functions can use Cloud Bigtable, and Azure functions can use Cosmos DB. Through these intermediaries, *stateful serverless functions* (SSFs) can save state and expose it to other instances.

Unfortunately, the above approach to state interacts poorly with the way that serverless platforms handle failures. If a function in a workflow crashes or its worker hangs, the provider will either (1) do nothing, leaving the workflow incomplete, or (2) restart the function on a different worker, potentially incrementing a counter twice, popping a queue multiple times, or corrupting database state and violating application semantics. Indeed, serverless providers currently recommend that developers write SSFs that are idempotent to ensure that re-execution is safe [17]. While helpful, these recommendations place the burden entirely on developers. In contrast, Beldi simplifies this process so developers need only worry about their application logic and not the low-level details of how serverless providers respond to failures.

2.2 Requirements and assumptions

We strive to design a framework that helps developers build serverless applications that tolerate failures and handle concurrent operations correctly. Our concrete goals are:

Exactly-once semantics: Beldi should guarantee *exactly-once* execution semantics in the presence of SSF or worker crash failures. That is, even if an SSF crashes in the midst of its execution and is restarted by the provider an arbitrary number of times, the resulting state must be equivalent to that produced by an execution in which the SSF ran exactly once, from start to finish, without crashing.

SSF data sovereignty: Beldi should support SSFs that are developed and managed independently. For example, multiple instances of an SSF may all access the same database, but they might not have access to the databases of other SSFs, even those in the same workflow. Instead, state should only be exposed by choice through an SSF's outputs. This type of encapsulation is important to support a paradigm in which different developers, organizations, and teams within the same organization are responsible for designing and maintaining their own SSFs. An application developer can then contract with SSF developers (or teams) to integrate their SSFs into the application's workflow via the SSF's identifier (§2). Furthermore, data sovereignty is key to enabling developers to offer proprietary functions-as-a-service to others, and is a best practice in microservice architectures [11, §4]. For example, Microsoft's eShopOnContainers [29] serves as a blueprint for applying these ideas to real-world applications.

SSF reusability: Beldi should allow multiple applications to use the same SSFs in their workflows at the same time. This may require each SSF to have different tables or databases to maintain the state of each application separately, though cross-application state should also be supported.

Workflow transactions: Beldi should support an optional transactional API that allows an application to specify any subgraph of a workflow that should be processed transactionally with ACID semantics. We target *opacity* [20] as the isolation level. Opacity ensures that (1) the effects of concurrent transactions are equivalent to some serial execution, and (2) every transaction, including those that are aborted, always observes a consistent view of the database. We discuss why these requirements are important in SSFs in Section 6.2.

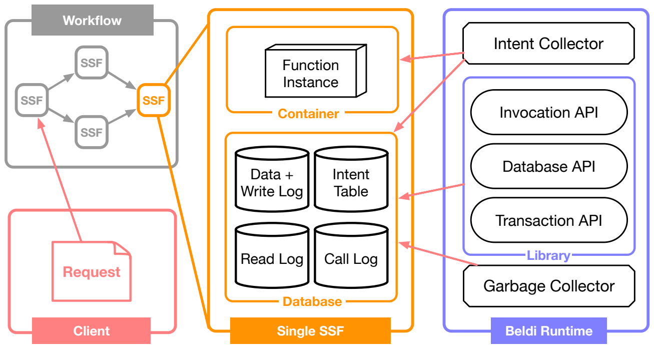


FIGURE 1—Beldi's architecture. Developers write SSFs as they do today, but use the Beldi API for transactions and externally visible operations. At runtime, operations for each SSF are logged to a database, which, when combined with a per-SSF intent and garbage collector, guarantees exactly-once semantics.

Deployable today: Beldi should work on existing serverless platforms without any modifications on their end. This allows developers to use Beldi on any provider of their choosing (or even a multi-provider setup), and lowers the barrier to switch providers. Additionally, developers should not need to run any servers in order to use Beldi. After all, a big appeal of serverless is that it frees developers from such burdens.

Assumptions. To achieve these goals, Beldi makes some assumptions about the storage provided to SSFs: that it supports strong consistency, tolerates faults, supports atomic updates on some atomicity scope (e.g., row, partition), and has a scan operation with the ability to filter results and create projections. These assumptions hold for the NoSQL databases commonly used by SSFs: Amazon's DynamoDB, Azure's Cosmos DB, and Google's Bigtable.

3 Design Overview

Beldi consists of a library that developers use to write their SSFs and a serverless-based runtime system to support them. Beldi's approach to handling SSF failures is based on an idea most recently explored by Olive [36] and inspired by decades of work on log-based fault tolerance [19, 30]. Specifically, Beldi executes SSF operations while atomically logging these actions and periodically re-executes SSFs that have not yet finished. The logs prevent duplicating work that has already been done, guaranteeing *at-most-once* execution semantics, while the re-execution ensures *at-least-once* semantics.

Figure 1 depicts Beldi's high-level architecture. Beldi consists of four components: (1) the Beldi library, which exposes APIs for invocations, database reads/writes, and transactions; (2) a set of database tables that store the SSF's state, as well as logs of reads, writes, and invocations; (3) an *intent collector*, which is a serverless function that restarts any instances of the corresponding SSF that have stalled or crashed; and (4) a *garbage collector*, which is a serverless function that keeps the logs from growing unboundedly.

To ensure data sovereignty (§2.2), the runtimes and logs of different SSFs are independently managed and stored; however, all instances of *related* SSFs may share the same Beldi infrastructure. An app developer composes multiple SSFs into a workflow by chaining them together using a driver function, a step function, or a combination of the two. In the following sections we expand on each of these components.

3.1 Initial inspiration: Olive

Olive [36] guarantees exactly-once execution semantics for clients that may fail while interacting with a fault-tolerant storage server. This is a similar objective as ours, though our setting makes applying Olive's ideas nontrivial. An intent in Olive is an arbitrary code snippet that the client intends to execute with exactly-once semantics. Each intent is assigned a unique identifier (*intent id*), which Olive uses as the primary key to save its progress. A client in Olive enjoys *at-most-once* semantics by checking the intent's progress and skipping completed operations during re-execution. Intents consist of local and external operations. For example, incrementing a local variable is a local operation, whereas reading a value from storage is an external operation. Each external operation in the intent is assigned a monotonically increasing *step number*, starting at 0, that uniquely identifies it.

There are two key requirements for intents. First, intents must be deterministic; developers can make non-deterministic operations (e.g., a call to a random number generator) deterministic by logging their results and replaying the same values in the event of a re-execution. Second, intents must be guaranteed to always complete in the absence of failures (e.g., they must be free from bugs such as deadlock or infinite loops).

After an intent has been successfully logged, the client in Olive executes the intent's code normally until it reaches an external operation (e.g., reading or writing to the database). Then, the client: (1) determines the operation's step number; (2) performs the operation (e.g., writes to the database); (3) logs the intent id, step number, and the operation's return value (if any) into a separate database table called the *operation log*. When the client completes all operations, it marks the intent as 'done' in the intent table.

To ensure at-most-once execution semantics, the client in Olive must perform actions (2) and (3) above atomically. This ensures that if Olive re-executes an intent, there will be a record in the operation log showing that a particular step has already been completed and should not be re-executed. Instead, the entity re-executing the intent should resume execution from the last completed step, using logged return values from previous steps as needed. To make these two actions atomic, Olive introduces a technique called *Distributed Atomic Affinity Logging* (DAAL), which collocates log entries for an item in the same *atomicity scope* (the scope at which the database supports atomic operations) with the item's data. For example, in a storage system where operations are atomic at the row level, Olive would store the item's value and its log entries in different columns of the same row.

Beldi Library Function	Description
`read(k)` $\rightarrow$ `v`	Read operation
`write(k, v)`	Write operation
`condWrite(k, v, c)` $\rightarrow$ T/F	Write if `c` is true
`syncInvoke(s, params)` $\rightarrow$ `v`	Calls `s` and waits for answer
`asyncInvoke(s, params)`	Calls `s` without waiting
`lock()`	Acquire a lock
`unlock()`	Release a lock
`begin_tx()`	Begin a transaction
`end_tx()`	End a transaction

FIGURE 2—Beldi's API for SSFs, which includes all of Beldi's primitives and its transactional support (§6).

Intent collector. To guarantee *at-least-once* semantics, Olive must ensure that some entity finishes the intent if the client crashes. This is the job of the *intent collector* (IC), which is a background process that periodically scans the intent table and completes unfinished intents by running their code. Before the IC executes an external operation, it consults the operation log table with the operation's step number to see if the operation has already been done and to retrieve any return value; regular clients also perform this check between actions (1) and (2). If the operation has not been done, the IC atomically executes the operation and logs the result to the operation log table. Even if multiple IC instances execute concurrently, or if the IC starts executing the intent of a client that has not crashed, this is safe because of Olive's assurance of at-most-once semantics.

Beldi vs. Olive. Beldi is inspired by Olive's high-level approach but makes key changes and introduces new data structures and tables, support for invocations so that SSFs can call each other (Olive only supports storage operations), and garbage collection mechanisms to keep overheads low.

An important difference between the two is the definition of an 'intent.' In Olive, intents are code snippets—logged by the client—and all intents are logged in the same intent table. In Beldi, the *client* (which is the SSF) is the code snippet. As a result, an intent in Beldi is not code but rather the parameters that identify a particular running instance of the SSF: its inputs, start time, whether it was launched asynchronously, etc. Accordingly, Beldi uses the term 'instance id' instead of 'intent id' to capture this distinction.

Another critical difference is that, as shown in Figure 1, each SSF in Beldi is backed by a different database and Beldi runtime to ensure data sovereignty, though different SSFs developed by the same engineering team may reuse these components if desired. We will expand on these details in the following sections, but we begin by introducing Beldi's API.

3.2 Beldi's API

Beldi exposes the API in Figure 2, which includes key-value operations such as `read`, `write`, and `condWrite` (a write that succeeds only if the provided condition evaluates to true), and functions to invoke other SSFs (`syncInvoke` and

Log	Key	Value
`intent`	instance id	done, async, args, ret, ts
`read`	instance id, step number	value
`write`	instance id, step number	true / false
`invoke`	instance id, step number	instance id of callee, result

FIGURE 3—Beldi maintains four logs for each SSF. The intent table keeps track of an instance's completion status, arguments, return value, type of invocation, and timestamp assigned by its garbage collector (ts). The read log stores the value read. The write log stores true for writes, or the condition evaluation for a conditional write. The invoke log stores the instance id of the callee and its result.

`asyncInvoke`). These operations are meant as drop-in replacements for the existing interface used by SSFs. Furthermore, Beldi supports the ability to `begin` and `end` transactions; operations between these calls enjoy ACID semantics.

Beldi's API hides from developers all of the complexity of logging, replaying, and concurrency control protocols that take place under the hood to guarantee exactly-once semantics and support transactions. For example, an SSF using Beldi's API automatically determines (from the input, environment, and global variables) the SSF's instance id, step number, and whether it is part of a transaction. Beldi takes actions before and after the main body of the SSF as well as around any Beldi API operations.

3.3 Beldi's runtime infrastructure

Developers write SSF code as they do today, but link Beldi's library and use its API. The rest of Beldi's mechanisms happen behind the scenes.

Intent table. Beldi associates with every SSF invocation an *instance id*, which uniquely identifies an intent to execute a given SSF. For the first SSF in a workflow, the instance id is the UUID assigned by the serverless platform to the initial request. For example, in AWS this UUID is called the 'request id,' in GCP it is called the 'event id,' and in Azure it is the 'invocation id.' For subsequent SSFs in the workflow, each caller in the graph will generate a new UUID to be used by the callee as its instance id. A new id is generated even if the SSF has been invoked earlier in the workflow or if the callee is another instance of the caller SSF (in the case of recursive functions). Thus, every SSF instance will have a distinct instance id, even if the instances are of the same SSF and in the same workflow.

Beldi keeps an intent table that contains the instance id, arguments, completion status, and other information listed in Figure 3 for every SSF instance that users and other SSFs intend to execute. It does this by modifying SSFs to ensure that the first operation is to check the intent table to see if their instance id is already present and, if not, to log a new entry. Beldi performs a similar modification to set the intent as 'done' at the end of the SSF execution.

Operation logs. In addition to the intent table, Beldi maintains three logs for each SSF: a *read log*, *write log*, and *invoke log*. Their schema is also in Figure 3. For each operation, the key into the log is the combination of the executing SSF's instance id and the step number, which (like in Olive) is a counter that identifies each unique external operation. Each read operation adds the value read from the database into the read log. Writes, meanwhile, write to the write log with a boolean flag that states whether the write operation took effect. Regular writes always set this flag to true, while conditional writes set it to the outcome of the condition at the time of the write. The actual data being written is stored in a data table, although in Section 4 we discuss a data structure that generalizes Olive's DAAL and collocates the write log in the same table as the data to avoid cross-table transactions. The invoke log is new to Beldi and ensures at-most-once semantics for calls to other SSFs; we describe it in Section 4.5.

Intent and Garbage Collectors. For each SSF, Beldi introduces a pair of serverless functions that are triggered periodically by a timer. The first function acts as the SSF's *intent collector* (IC). The IC scans the SSF's intent table to discover instances of the SSF that have not yet finished (lack the 'done' flag). The IC restarts each unfinished SSF by re-executing it with the original instance id and arguments. Note that it is safe for the IC to restart an SSF instance even if the original instance is still running and has not crashed, owing to Beldi's use of logs to guarantee at-most-once semantics for each step of the SSF. We implement two natural optimizations for the IC. First, the IC restarts instances only after some amount of time has passed since the last time they were launched to avoid spawning too many duplicate instances in cases where the IC runs very frequently. Second, the IC speeds up the process of finding unfinished instances among all instance ids in the intent table by maintaining a secondary index.

The second function acts as a *Garbage Collector* (GC) for completed intents, taking care to ensure safety in the presence of concurrent SSF instances, IC instances, and even GC instances. This component is described in Section 5.

4 Executing and Logging Operations in Beldi

As we mention in Section 3.1, guaranteeing exactly-once semantics requires atomically logging and executing operations. This section discusses how Beldi achieves this.

4.1 Linked DAAL

The logging approach taken by Olive (§3.1) requires an *atomicity scope* with high storage capacity, as otherwise few log entries can be added. In the context of Cosmos DB (the successor of the database used by Olive), the atomicity scope is a database partition, and the atomic operation is a transactional batch update. Olive's DAAL is a good fit for Cosmos DB because partitions can hold up to 20 GB of data [10], which is enough to collocate a data item and a large number of log entries. However, other databases adopt designs with more limited atomicity scopes. For example, the atomicity scope of DynamoDB and Bigtable is one row, which can hold up to

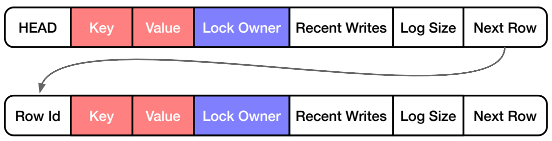


FIGURE 4—Linked DAAL for a single item. Each row contains the item's key, previous values (except the last row which contains the current value), lock information (used for transactions), a log of recent writes, and information for traversal and garbage collection.

400 KB [14] and 256 MB [7], respectively; the recommended limits are much lower. If we were to use Olive's DAAL with DynamoDB, an SSF could only perform hundreds of writes to a given key before filling up the row. At such point, Olive would be unable to make further progress until the logs are pruned. This is hard to do in our setting: reaching a state of quiescence where it is safe to garbage collect logs is challenging since existing platforms expose no mechanism to kill or pause SSFs (§5).

To support all common databases, Beldi introduces a new data structure called the *linked DAAL* that allows logs to exist on multiple rows (or atomicity scopes), with new rows being added as needed. There are three reasons why this simple data structure is interesting for our purposes: (1) linked DAALs continue to avoid the overheads of cross-table transactions and work on databases that do not support such transactions; (2) linked DAALs are a type of non-blocking linked list [21, 42, 47], allowing multiple SSFs to access them concurrently with the operations supported by the atomicity scope (e.g., atomic partial updates); (3) even with frequent accesses, our garbage collection protocol can ensure that the length of the list for each item is kept consistently small (§5).

Structure. Figure 4 gives an example of a linked DAAL for an item with two rows of logs. Every row stores the item's key, value, owner of the lock (used for transactions in Section 6), the log of writes, and metadata needed to traverse the linked DAAL and perform garbage collection. The first row is the 'head,' which has a special `RowId` and is never garbage collected. The primary key for rows is `RowId` + `Key`, the hash key is `Key`, and the sort key is `RowId`. When a row is full and the SSF issues a write operation, a new row is appended with the updated value and a log entry describing the write; the previous row's value and logs are not modified once filled. Thus, the tail always has the most recent value.

Traversal. Most operations in Beldi require traversal to the tail of the list. The simplest way to accomplish this is to start at the designated head row and iteratively issue read requests for each `NextRow` until the field is empty. While this procedure will eventually reach the tail, the number of database operations grows with the length of the list. Garbage collection can control this length, but Beldi applies an additional optimization that leverages the scan and projection operations available in the three NoSQL databases that we surveyed. Specifically, Beldi issues a single scan operation to

```
def read(table, key):
    linkedDAAL = rawScan(table,
        cond: "Key is {key}",
        project: ["RowId", "NextRow"])
    tail = getTail(linkedDAAL)
    val = rawRead(table, tail)
    logKey = [ID, STEP]
    STEP = STEP + 1
    ok = rawCondWrite(ReadLog, logKey,
        cond: "{logKey} does not exist"
        update: "Value = {val}")
    if ok:
        return val
    else:
        return rawRead(ReadLog, logKey)
```

FIGURE 5—Pseudocode for Beldi's read wrapper function. Functions beginning with "`raw`" refer to native (unwrapped) access to the database tables storing the data or the logs. Identifiers starting with capital letters indicate a member of the log structures.

the database that returns every row containing a target `Key`. On its own, the scan operation returns all contents of each row (including the values, write logs, etc.). To reduce this overhead, Beldi applies a projection that filters out all columns except for `RowId` and `NextRow`. This combination of scan and projection allows Beldi to download only 256 bits per row of the linked DAAL. From these rows, Beldi constructs a skeleton version of the linked DAAL locally, which it can quickly traverse to find the `RowId` of the tail.

We note that the individual reads in a scan are not executed atomically. For example, Beldi might see a row with no `NextRow`, and also receive a row that was subsequently appended to it. This operation might even retrieve rows that are orphaned from a failed append operation. Regardless, when these databases are configured to be linearizable [6, 9, 13], the set of rows traversed from the head to the first instance of an empty `NextRow` form a consistent snapshot of the linked DAAL—any write that completes strictly before the scan begins will be reflected in the constructed local linked DAAL.

While the linked DAAL is structurally simple, operating on it requires care. The following sections detail how Beldi's API functions read and modify the linked DAAL.

4.2 Read

We begin by discussing Beldi's `read` operation. While `read` has no externally visible effects on its own, the potential use of its non-deterministic results in a subsequent external operation means that Beldi must record the result of every `read` in a dedicated *ReadLog*. Unlike write operations, however, the read from the database and the log to the ReadLog need not happen atomically—if the SSF crashes before logging the outcome, it is fine to fetch a fresh value as the previous result did not have any externally visible effect.

Figure 5 shows the pseudocode of the `read` API function, which involves two steps: (1) read the most recent value of the key from the tail of the linked DAAL, and (2) log the result

```
def write(table, key, val):
    logKey = [ID, STEP]
    linkedDAAL = rawScan(table,
        cond: "Key is {key}"
        project: ["RowId", "NextRow",
                 "RecentWrites[{logKey}]"])
    if logKey not in linkedDAAL:
        tail = getTail(linkedDAAL)
        tryWrite(table, key, val, tail)
    STEP = STEP + 1
def tryWrite(table, key, val, row):
    logKey = [ID, STEP]
    ok = rawCondWrite(table, row[RowId],
        cond: "({logKey} not in RecentWrites)
              && (LogSize < N)",
        update: "Value = {val};
                 LogSize = LogSize + 1;
                 RecentWrites[{logKey}] = NULL")
    if ok: # Case B
        return
    row = rawRead(table, row[RowId])
    if logKey in row[RecentWrites]: # Case A
        return
    elif row[NextRow] does not exist: # Case D
        row = appendRow(table, key, row)
    else: # Case C
        row = rawRead(table, row[NextRow])
    tryWrite(table, key, val, row)
```

FIGURE 6—Pseudocode for Beldi's write wrapper function.

to the ReadLog if it has not yet been completed. For the first step, Beldi retrieves the tail as described in Section 4.1. For the second step, Beldi uses an atomic conditional update to efficiently log the operation without overwriting a previously executed read. If it encounters a conflict during the update, it returns the previous result from the ReadLog.

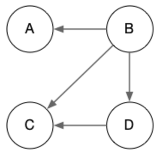
4.3 Write

A write is more complex as the update and logging must be done atomically—within the same atomicity scope—and Beldi needs to handle cases where other SSFs are accessing and appending to the linked DAAL concurrently. At a high level, the write operation must find the tail of the linked DAAL, check if the write has been previously executed, log/update the tail if it has not, and extend the linked DAAL if the current tail is full. Like `read`, Beldi can use scan and projection to assemble a minimal local version of the linked DAAL. Unlike `read`, Beldi cannot skip directly to the tail; instead, Beldi must check that none of the scanned rows contains a record of the current operation. Furthermore, once Beldi has a candidate for the tail, Beldi needs to update its value and add an entry to its log atomically. For a given tail candidate there are exactly four possible scenarios:

A. The operation has already been executed and the [instance ID, step number] tuple can be found in the current row. Beldi can return immediately in this case.
B. The operation is not in the log and there is still space. This indicates that Beldi is at the tail, the operation has never been executed previously, and there is room in the current row to execute/log the write.
C. The operation is not in the log, but the log is full and there is a pointer to the next row. Beldi should follow the provided pointer toward the tail.
D. The operation is not in the log and the log is full, but there is no next row. Beldi should append a new row and advance to that new row.

	logKey $\in$ logs	logSize $< N$	$\exists$ nextRow
A	True	$*$	$*$
B	False	True	False
C	False	False	True
D	False	False	False

(a) Cases



(b) Transitions

FIGURE 7—Possible cases for the state of a candidate tail in the linked DAAL during a `write` and its potential transitions.

We formulate a lock-free algorithm to handle all the cases above by examining the transitions induced by concurrent SSF accesses. For example, if Beldi is in case B, where the operation is not in any log and there is still space to execute it in the current row, a concurrent SSF can, without warning, execute the current operation ($\rightarrow$A) or fill the remaining space in the log ($\rightarrow$C/D). The reverse is not true: once there is a `NextRow` pointer, the linked DAAL will never revert to having extra space for logs. The cases and their transitions are summarized in Figure 7, where N is the maximum number of log entries that can fit in a row when accounting for the size of the key, value, and other metadata. The exception is garbage collection (not covered in Figure 7), whose operation and correctness we describe in Section 5. An arrow in Figure 7b indicates a possible effect of concurrent SSF instances.

To safely identify the state of a row, Beldi checks for each case starting at the node(s) in the transition graph without incoming edges. In this case there is only one such node (B), so Beldi performs a conditional write with the condition given in case B of Figure 7a (i.e., that the logKey is not in the logs, that the logSize is less than N, and that there is no nextRow). If the conditional check fails, the state of the row will not revert back to case B later because B has no incoming edges. Therefore, it is safe to remove B from the transition graph and check the remaining cases. Beldi repeats the above process with cases A and D (in any order) because they they have no incoming edges in the remaining graph. Finally, if all prior conditions fail, the row is in case C.

4.4 Conditional write

Beldi also provides support for conditional writes, which only execute if a user-defined condition is true at the time of the write. The initial scan and subsequent scenarios are similar to the scenarios for unconditional writes. The only exception is the case where the operation has not previously executed and the current row still has remaining space in the log (i.e., case B from Section 4.3). We split this case into two: in B_1, the condition is true, and in B_2, the condition is false.

```
def syncInvoke(callee, input):
    calleeId = UUID()
    logKey = [ID, STEP]
    STEP = STEP + 1
    ok = rawCondWrite(InvokeLog, logKey,
        cond: "{logKey} not in InvokeLog"
        update: "Id = {calleeId};
                Result = NULL")
    if not ok:
        record = rawRead(InvokeLog, logKey)
        calleeId = record[Id]
        result = record[Result]
    if result does not exist:
        return rawSyncInvoke(callee,
            [calleeId, input])
# When the Callee is done it issues a callback
# to the caller. Below is the callback handler.
def syncInvokeCallbackHandler(calleeId, result):
    rawWrite(InvokeLog, cond: "Id = {calleeId}",
            update: "Result = {result}")
```

FIGURE 8—Pseudocode for synchronous invocation of other SSFs. Asynchronous invocations are similar, but since they do not have return values, the callback is invoked as soon as the callee logs the intent in its intent table. We give the code for the callee's actions in Appendix A of our tech report [45].

Beldi handles these cases by first checking B_1 and B_2 with conditional writes before covering the other states exactly as in the unconditional-write case. We give a detailed description in Appendix A of our extended technical report [45].

4.5 Invocation of SSFs and local functions

Finally, Beldi supports three types of function invocations: synchronous calls (`syncInvoke`), which block and return a value; asynchronous calls (`asyncInvoke`), which return immediately; and calls to functions that do not use Beldi's API (e.g., legacy libraries or legacy SSFs). In the first two cases, Beldi guarantees exactly-once semantics. In the last, it only guarantees that the operation is performed at least once.

Figure 8 shows pseudocode for synchronous SSF invocations. As mentioned in Section 3.3, to help SSFs that are being invoked ("callees") differentiate between re-executions and new executions, Beldi passes an instance id to the callee (the "callee id") along with the parameters of the call. In the first invocation, the callee id is generated using `UUID()`; for re-executions, it retrieves the id from the invoke log. If there is already an entry in the invoke log for this caller id and step number, there are two cases: (1) a result is already present, in which case the caller reuses that result; or (2) the entry is present but there is no result, in which case the caller re-invokes the callee with the existing callee id.

Callbacks. Note that `syncInvoke` (Figure 8) does not log the result of the actual call or otherwise mark the call as complete. To see why this is important, consider the example trace in Figure 9, which shows the result of a failure of the callee (SSF2) after it marks itself as done in the intent table

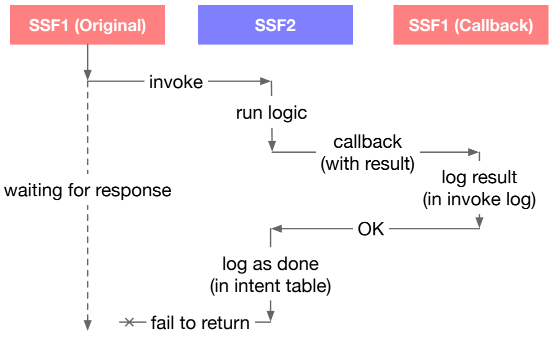


FIGURE 9—SSF1 synchronously invokes SSF2, which then fails to return after logging the operation as done. The callback ensures that SSF1 has the result of SSF2 before SSF2 marks itself as done.

but before it returns the result to the caller (SSF1). Suppose that there is no callback, i.e., that SSF2 logs itself as complete immediately after completing execution. Beldi's federated setup means that each SSF has a garbage collector running at its own pace. If SSF2 were to fail after logging itself as done, it is, therefore, possible that SSF2's GC will garbage collect the intent before SSF1 gets any value. Later, when SSF1's IC re-executes the unfinished SSF1 instance, the caller will see the lack of result in the invoke log, re-invoke SSF2 (with the existing callee id), and SSF2 will mistakenly perform the operation again. In some ways, this is similar to why `write` operations in Beldi must be atomically logged and executed (§3.1). Unfortunately, there are no mechanisms for atomically logging into a database and executing other SSFs.

We address this issue by decomposing an invocation into two steps: (1) the invocation itself, performed by the caller; and (2) the recording of results, done via a second, automatic invocation by the callee to *some* instance of the caller. We emphasize 'some' and 'original' because request routing in serverless is stateless: if SSF1 invokes SSF2, and SSF2 then invokes SSF1, the two SSF1 instances could be different (§2.1). We call this automatic invocation a *callback*. When the second instance of the caller receives the callback, it logs the provided result in its invoke log and returns. At this point, it is safe for the callee to mark its intent as done since it knows the caller's invoke log already contains the result. Note that callbacks only require at-least-once semantics, so there is no need for additional logging of the callback invocation.

Figure 9 illustrates the idea of Beldi's callback mechanism. The callback ensures that the result of SSF2 is properly received by SSF1. As such, we note that SSF2's response to SSF1 is merely an optimization and not necessary for correctness. We also note that if SSF2 fails after a successful callback but before logging the completion of the intent, it may result in a case where SSF1 completes, gets garbage collected, and then a re-execution of SSF2 invokes a spurious callback. SSF1 can detect and ignore this case when a callback occurs for an invoke that does not exist.

Asynchronous invocations. This procedure is similar to that of synchronous invocations, but with the two steps flipped on the callee. The caller first makes a `rawSyncInvoke` call to the callee, but rather than execute the function, the callee (observing an 'async' flag) simply registers the intent in its intent table, issues a callback, and then immediately returns to the caller. In the second step, Beldi performs the actual asynchronous invocation of SSF2's logic. We describe this operation in detail in Appendix A of our tech report [45].

5 Garbage Collection

If left alone, the linked DAAL will grow indefinitely. While Beldi's use of scans means that the linked DAAL's length is generally not the performance bottleneck, unbounded growth of the linked DAAL and logs (intent table, read log, invoke log) can lead to significant overheads and storage costs. Beldi ensures that logs are pruned and the linked DAAL remains shallow with a garbage collector (GC) that deletes old rows and log entries without blocking SSFs that are concurrently accessing the list. The GC is an SSF triggered by a timer.

At a high level, the protocol has six parts. First, the GC finds intents that have finished since the last time a GC instance ran and assigns them the current time as a finish timestamp. Second, the GC looks up all intents whose finish timestamp is 'old enough' (we expand on this next), and marks them as 'recyclable.' Third, the GC removes log entries (in the read and invoke logs) that belong to recyclable intents. Fourth, the GC disconnects, for every item, the non-tail rows of their linked DAAL that have empty logs, marks these rows as 'dangling', and assigns them the current time as a dangling timestamp. Fifth, the GC removes all rows whose dangling timestamp is 'old enough.' Finally, the GC removes the log entries from the intent table. The algorithm is given in Figure 10, with more details in Appendix A of our tech report [45]. Note that GCs only need at-least-once semantics to avoid memory leaks in the presence of crashes; they do not use Beldi's exactly-once API. Instead, GCs defer the removal of entries in the intent table until the end.

Assumption. The safety of garbage collection relies on a synchrony assumption. In particular, it assumes that an individual SSF instance terminates, one way or another, in at most T time. This allows the GC to delete the logs of completed intents after waiting T time for all running instances of the completed intents to finish. Note that no new instances will be started by an SSF's IC after the intent is marked as 'done.'

Our assumption is based on the observation that serverless providers enforce user-defined execution timeouts on SSF instances (§2.1), but otherwise provide no interface for developers to kill or stop running functions. We can derive a conservative bound for T from these user-defined timeouts. Note that even if providers refuse to kill SSFs after the timeout, we can work around this issue (at high cost) by having the GC change the database's permissions or rename tables so that ongoing SSF instances (including stragglers that stick around after the intent is done) fail to corrupt the database; instances that start after the change are fine.

```
def garbageCollection():
    time = now()
    recyclable = []
    for id, intent in IntentTable:
        if intent[Done]:
            if FinishTime not in intent:
                intent[FinishTime] = time
            elif time - intent[FinishTime] > T:
                recyclable.append(id)
    for id in recyclable:
        remove from ReadLog
            where "LogKey[Id] == {id}"
        remove from InvokeLog
            where "LogKey[Id] == {id}"
    for table, key in getAllDataKeys():
        rows = rawScan(table,
                       cond: "Key == {key}")
        for row in rows:
            for log in row[RecentWrites]:
                mark if log[Id] in recyclable
            if fullyMarked(row[RecentWrites])
                    and row[NextRow] exists:
                prev(row)[NextRow] = row[NextRow]
                if DangleTime not in row:
                    row[DangleTime] = time
        rows = rawScan(table, cond: "Key == {key}
                    && {time} - DangleTime > T")
        for row in rows:
            if row not reachable from head(key)
                delete row
    for id in recyclable:
        remove from IntentTable
            where "LogKey[Id] == {id}"
```

FIGURE 10—Pseudocode for Beldi's lock-free, thread-safe garbage collection algorithm. T is the maximum lifetime of an SSF instance.

Safety of concurrent access. With the above assumption, Beldi's GC preserves exactly-once semantics without needing to interrupt SSF instances. First, observe that an intent is marked as recyclable only after Beldi is sure that no live SSF instance requires the intent. Accordingly, the read log, invoke log, and intent table entries for the intent will never be accessed again. For the linked DAAL, the GC only disconnects a row when all of the contained logs are marked as recyclable and it is not the tail. New traversals of the linked DAAL for read or write operations will not observe the disconnected row (technically the `rawScan` operation will return these disconnected rows, but they will be ignored during the traversal of the local linked DAAL). Running SSF and GC instances, however, may be in the process of traversing the disconnected row—if Beldi deleted it immediately, the SSF or GC might become stranded. To prevent this, Beldi keeps the disconnected row for an additional T time to ensure that instances with such references terminate successfully.

Safety of concurrent modifications. The linked DAAL also supports garbage collection in the presence of concurrent appends from SSFs and deletions from other GC instances owing to it being a type of non-blocking linked list. In fact, it is simpler than traditional non-blocking linked lists [21, 42, 47] because new rows are always appended to the tail, and GCs never touch the tail. The only interesting case is the concurrent disconnection of neighboring rows such as X and Y in $A \rightarrow X \rightarrow Y \rightarrow B$. In this case, the disconnection of X succeeds, but the disconnection of Y will not be visible because the updated `NextRow` pointer in X is no longer part of the linked DAAL. The next GC run disconnects Y permanently.

6 Supporting Locks and Transactions

In addition to exactly-once semantics, Beldi also provides support for locks and transactions with user-generated aborts.

6.1 Locks

Beldi's approach to mutual exclusion borrows an abstraction in Olive called "locks with intent", where locks over data items are owned by an intent rather than a specific client. This means that, if an SSF instance calls `lock(item)` and then crashes, the lock is not lost and held indefinitely; rather, the IC will soon restart the instance. The re-executed instance, upon arriving at the `lock(item)` call, will see that it already acquired the lock and be able to continue with the remaining operations as if the original SSF instance had never crashed.

In Beldi, the ownership of a lock on a given item is kept alongside the data and logs in the "lock owner" column of the item's linked DAAL. Lock acquisition and release are logged to the DAAL as writes to the item using Beldi's `condWrite` semantics, where the condition is that the lock is either owned by the current SSF or has an empty lock-owner column in the DAAL. The exactly-once semantics are needed for cases where an SSF is re-run after successfully releasing a lock.

Note that Beldi only guarantees exactly-once semantics—it does not absolve the developer from writing bug-free code. Thus, problems like infinite loops within critical sections and deadlock need to be handled with higher-level mechanisms (like the one below) if the user wishes to guarantee liveness.

6.2 Transactions

Beldi uses an extension of the locking mechanism of the preceding section to implement transactions within and across SSF boundaries. Beldi transactions are based on a variant of 2PL with wait-die deadlock prevention and two-phase commit. Note that the choice of wait-die (rather than something like wound-wait) is deliberate as SSF instances generally cannot kill other instances. To implement this, we need to track the intent-creation time of each SSF. We do so by adding to the lock-owner column an intent-creation timestamp and checking upon lock-acquisition failure whether the existing lock owner is older or younger than the current SSF instance; if older, abort, otherwise, try again (see Figure 11).

There are three main parts to Beldi's transaction-handling protocol: (1) creating and forwarding a *transaction context*, (2) executing Beldi calls inside a transaction, and (3) propagating abort/commit signals throughout a workflow. Note that Beldi does not currently support `asyncInvoke` in transactions; however, it does support spawning threads that issue `syncInvoke` operations and are then joined.

```
def lock(table, key):
  ok = condWrite(table, key,
      cond: "LockOwner = NULL
             || LockOwner.id = TXNID",
      update: "LockOwner = [TXNID, START_TIME]")
  if not ok:
    row = read(table, key)
    ownerId, ownerTime = row[LockOwner]
    if ownerTime <= TXNID:
      abort
    else:
      lock(table, key)
```

FIGURE 11—Pseudocode for the lock operation with wait-die deadlock prevention used during the 'Execute' mode of a transaction.

Transaction contexts. In Beldi, transactions are defined with the `begin_tx` and `end_tx` API calls. Beldi assumes that both the begin and the end statements are placed in the same SSF, but SSFs can invoke other SSFs inside a transaction, so transactions can span across multiple SSFs. When an SSF calls `begin_tx` it creates a new top-level *transaction context* which consists of a unique transaction id and a mode ('Execute', 'Commit', or 'Abort'). Contexts start in 'Execute' mode. The SSF instance will also, upon creating a new context, execute the transaction's operations in a new thread/goroutine to catch any runtime exceptions. The matching `end_tx` waits for the result and runs either a commit or abort protocol depending on the outcome of the contained operations.

Transaction contexts are passed along with any SSF invocations that occur inside the transaction. Thus, whenever a Beldi-enabled SSF starts, it first determines whether it is a part of an ongoing top-level transaction by checking whether a context was provided as part of the input. This is necessary even if the SSF never creates a transaction itself. If the SSF does create a transaction, the `begin_tx`/`end_tx` statements will be ignored and all operations will be inherited by the top-level transaction context. Beldi does not currently support *nested transaction* semantics [31] (e.g., a sub-transaction can abort without causing the top-level transaction to abort).

Opacity. Beldi chooses *opacity* as the isolation level for transactions. Opacity [20] captures strict serializability [5, 34] with the additional requirement that even transactions that abort do not observe inconsistent state. The rationale is that observing inconsistent state can lead to undefined behavior and infinite loops. For example, if an SSF instance reads inconsistent state that results in division by zero, it may crash. Beldi's IC will restart the SSF instance and deterministically replay the (inconsistent) values to ensure exactly-once semantics, re-triggering the crash. Figure 12 gives another example of how OCC [26], which provides serializability but not opacity,

```
begin_tx()
x = read("x"); y = read("y")
while (x != y):
  // some logic
  x++
write("x", x + 2); write("y", y+4)
end_tx()
```

FIGURE 12—OCC leads to an infinite loop when two instances of the above transaction, T_1 and T_2, execute concurrently. Suppose $x = 0, y = 1$ initially. T_1 reads $x = 0, y = 1$, executes the logic, acquires locks on x and y, validates the read set, and writes $x = 3, y = 4$. T_2 reads $x = 3, y = 1$ (corresponding to a state after which T_1 updated x but before it updated y), and is stuck in an infinite loop. Even though T_2 is destined to abort, it will never reach the read set validation step.

leads to infinite loops. These issues are not present with isolation levels that guarantee that all transactions read from a consistent snapshot.

Operation semantics inside a transaction. If an SSF is in a transactional context, Beldi modifies the semantics of its API based on the mode to ensure ACID semantics. We have already discussed two operation modifications that occur in 'Execute' mode—one to locks in Figure 11 and another to `begin_tx`/`end_tx`, which are ignored. 'Execute' mode also causes Beldi to call `lock` before every `read`, `write`, and `condWrite` operation, using the transaction id as the lock holder. In addition to acquiring locks, Beldi also changes where reads and writes look up and record values. While lock acquisition still goes to the original tables, Beldi redirects written values to a *shadow table* that acts as a local copy of state for the transaction. Like the original table, this shadow table is also stored as a linked DAAL and is garbage collected along with the normal DAAL (except the GC also deletes the head and tail). Unlike the original, the shadow table is partitioned by transaction id, with `Key` relegated to a secondary index. All `read` operations check the shadow table first before consulting the real table to ensure that transactions read their own writes. If, before an operation, an SSF fails to acquire a lock and must kill itself (due to wait-die), it returns to its caller with an 'abort' outcome.

Propagation of commit or aborts. Eventually, a `begin_tx`/`end_tx` code block will reach the `end_tx` with an abort/commit decision. For commit, Beldi changes the mode of the context to 'Commit', flushes the final values of the items in the shadow table to the real linked DAAL, and releases any held locks. Beldi then calls the SSF's callees and passes them the transaction context in Commit mode. Note that if an SSF instance fails between flushing the shadow table and notifying the callees of the commit decision, Beldi's exactly-once semantics ensure that once the SSF instance is re-executed, it will pick up from where it left off. For abort, none of the values have been written to the actual table, so Beldi just releases all locks and invokes all callees in 'Abort' mode.

When an SSF is invoked with a transaction context that includes a Commit mode, Beldi skips the SSF's logic, and instead performs only the aforementioned commit protocol: flushes the final value of the items, releases any held locks, and notifies its own callees by invoking them with the provided transaction context. An Abort mode similarly skips the SSF's logic, releases all locks, and notifies its callees. This recursive invocation of callees with a Commit or Abort mode mimics the role of a coordinator in two-phase commit.

Supporting step functions. The previous discussion assumes a `begin_tx` and `end_tx` in the same SSF. To support transactions across SSFs defined in *step functions*, the developers must introduce 'begin' and 'end' SSFs in their workflow (we give an example in Appendix A of our tech report [45]). These SSFs create the transaction context and kickstart the commit or abort protocol. SSFs that fall between the 'begin' and 'end' SSFs in the workflow execute transactionally. If an SSF aborts it sends 'abort' on its outgoing edges in the workflow; an SSF that receives an abort as input skips its operations and propagates the abort message on its outgoing edges. This continues until the abort message reaches the 'end' SSF, which then sets the transaction context mode to Abort and invokes the 'begin' SSF. If 'end' executes without receiving any abort message, it sets the context mode to Commit instead. This invocation initiates the second phase of 2PC over the transactional subgraph of the workflow.

Non-transactional SSFs inside transactions. While an SSF that does not use transactions can be invoked inside a transaction by another SSF (which automatically forces the non-transactional SSF to acquire locks before any accesses), app developers must ensure that the non-transactional SSF is *only* used inside transactional contexts. Otherwise, non-transactional instances may access the database without acquiring locks or obeying the wait-die protocol.

7 Evaluation

Beldi brings forth an array of programmability and fault-tolerance benefits, but with these benefits come costs. In this section we are interested in answering three questions:

1. What is the cost of maintaining and accessing the linked DAAL, and how does it compare to applicable baselines?
2. What are the latency and throughput of representative applications running on Beldi, and how does Beldi compare to existing serverless platforms that provide neither exactly-once semantics nor transactional support?
3. What effect does Beldi's GC have on linked DAAL traversal, and how does it change as we adjust the timeout (T)?

We answer the above questions in the context of the following implementation, applications, and experimental setup.

7.1 Implementation

We have implemented a prototype of Beldi for Go applications that runs transparently on AWS Lambda and DynamoDB. In total, Beldi's implementation consists of 1,823 lines of Go for the API library and the intent and garbage collectors.

Case studies. To evaluate Beldi's ability to support interesting applications at low cost, we implement three case studies: a social media site, a travel reservation system, and a media streaming and review service. We adapt and extend these applications from DeathStarBench [12, 16], which is a recent open-source benchmark suite for microservices, and port them to a serverless environment (using Go and AWS Lambda). This port took around 200 person-hours. Combined, our implementations total 4,730 lines of Go. We provide details of the corresponding workflows in Appendix B of our tech report [45], and give a brief description below.

Movie review service (Cf. IMDB or Rotten Tomatoes): Users can create accounts, read reviews, view the plot and cast of movies, and write their own movie reviews and articles. Our implementation of this app consists of a workflow of 13 SSFs.

Travel reservation (Cf. Expedia): Users can create an account, search for hotels and flights, sort them by price/distance/rate, find recommendations, and reserve hotel rooms and flights. The workflow consists of 10 SSFs, and includes a cross-SSF transaction to ensure that when a user reserves a hotel and a flight, the reservation goes through only if both SSFs succeed. Note that we extend this app to support flight reservations, as the original implementation [12] only supports hotels.

Social media site (Cf. Twitter): Users can log in/out, see their timeline, search for other users, and follow/unfollow others. Users can also create posts that tag other users, attach media, and link URLs. The workflow consists of 13 SSFs that perform tasks like constructing the user's timeline, shortening URLs, handling user mentions, and composing posts.

7.2 Experimental setup

We run all of our experiments on AWS Lambda. We configure lambdas to use 1 GB of memory and set DynamoDB to use autoscaling in on-demand mode. All of the read and scan operations for Beldi and the baseline use DynamoDB's strong read consistency. We turn off automatic Lambda restarts and let Beldi's intent collectors take care of restarting failed Lambdas. Our garbage and intent collectors are triggered by a timer every 1 minute, which is the finest resolution supported by AWS. Note that AWS currently has a limit of 1,000 concurrent Lambdas per account. As we will see in some of our experiments, this limit is often the bottleneck in both the baseline and Beldi. Finally, consistent with our deployability requirement (§2.2), Beldi uses no servers.

The baseline for our experiments is running our ported applications on AWS Lambda without Beldi's library and runtime. Consequently, these applications will not enjoy exactly-once semantics or support transactions: when running on the baseline, the travel reservation app outputs inconsistent results, and all apps can corrupt state in the presence of crashes.

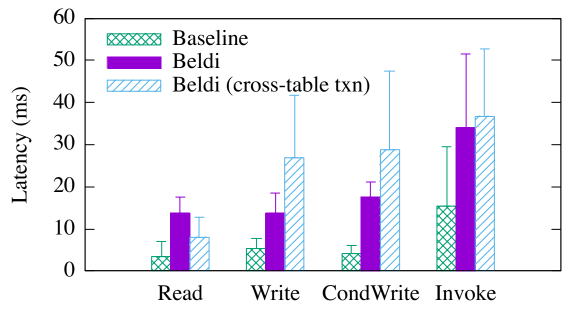


FIGURE 13—Median latency of Beldi's operations. Error bar represents the 99th percentile, and "cross-table tx" is an implementation of Beldi that uses cross-table transactions instead of the linked DAAL.

7.3 What are the costs of Beldi's primitives?

We start our evaluation with a microbenchmark that measures the cost of each of Beldi's primitive operations: `read`, `write`, `condWrite`, and `invoke`. The keys are one byte and the values are 16 bytes. We measure the median and 99th percentile completion time of the four operations over a period of 10 minutes at very low load (1 req/s). As baselines, we also measure the completion time (1) without Beldi's exactly-once guarantees and (2) using a design that logs writes to a separate table using cross-table transactions. Since Beldi's database operations depend on the length of the linked DAAL, we populate the chosen key's linked DAAL with a conservative value of 20 rows, which corresponds to the length of the linked DAAL after 30 minutes without garbage collection as described in the experiment of Section 7.5.

Figure 13 shows the overhead of Beldi's reads/writes compared to those of the baseline stem from two sources: scanning the linked DAAL (instead of reading a single row) and logging. For invoke, the overheads come from our callback mechanism and logging to the invoke log. Consequently, all of Beldi's operations are around 2–4× more expensive than the baseline. In contrast, the approach using cross-table transactions does not use a DAAL so `reads` avoid the scan (but not the logging), and `writes` perform an atomic transaction where the value is written to one table and the log entry is added to another. The cost of this operation is 2–2.5× higher than Beldi's linked DAAL. Appendix C in our tech report [45] describes the same experiment with a more optimistic setting (5 rows in the linked DAAL); the results are similar.

Note that not all existing databases (e.g., Bigtable) support cross-table transactions. Even for those that do, the performance gain that cross-table transactions have on read operations over using a linked DAAL goes away whenever SSFs use transactions because read locks use `condWrite` which is a cheaper operation on the linked DAAL.

Other costs. Another consideration beyond performance is the additional storage and network I/O required by Beldi to maintain and access all logs and linked DAAL metadata. For our setup above, the 20-row DAAL for the item takes up

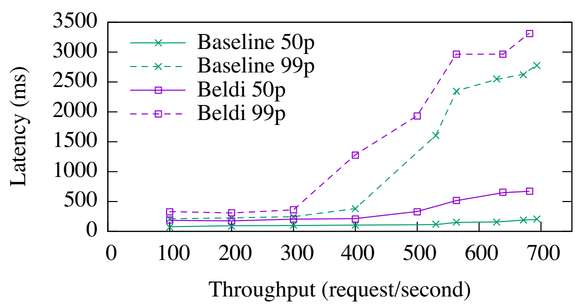


FIGURE 14—Median response time and throughput for our movie review service. Dashed lines represent 99th-percentile response time.

8 MB of storage. Counting all logs and metadata, each operation requires storing between 20 to 36 bytes in addition to the value. In terms of the network overhead introduced by the scan and projection approach that we use to traverse Beldi's linked DAAL, for a 20-row DAAL, each scan fetches 2 KB more data than a baseline read to a single cell when measured at the network layer. Compared to the baseline, Beldi induces one extra scan and write for each read operation, at least one scan for an unconditional write (and potentially more scans and writes depending on the scenario), and one read and two writes for a function invocation. In DynamoDB's on-demand mode, each read costs an additional $\$2.5 \times 10^{-7}$, whereas writes cost an additional $\$1.25 \times 10^{-6}$. In provisioned-capacity mode, costs are lower but depend on the specified capacity.

7.4 How does Beldi perform on our applications?

In this section, we discuss the results of our large-scale experiments for the movie review and travel reservation services; the social networking site has similar results, so we defer its results to our tech report [45]. The workloads that we use are adapted from DeathStarBench [12, 16] with a minor modification to support our extended travel reservation service: the transactions to reserve a hotel and flight randomly pick a hotel and a flight out of 100 choices each following a normal distribution. Requests contain random content within the expected schema and are generated and measured using wrk2 [44].

We issue load at a constant rate for 5 minutes, starting at 100 req/s and increasing in increments of 100 req/s until the system is saturated. For our applications, we achieve saturation at around 800 req/s. The primary bottleneck in all cases is compute: AWS enforces a limit of 1,000 concurrent Lambdas per account (even if the Lambdas are for different functions), and the HTTP Gateway (or some internal scheduler) rejects requests in excess of this limit.

Figures 14 and 15 depict the results. In all cases (including the social media app), we observe that, until around 400 req/s (34M per day), Beldi's median and 99th-percentile response time are each around 2× higher than that of the baseline. At the highest loads that we could test on AWS, Beldi still achieves the same throughput as the baseline at a slightly higher median response time (around 3.3× for the travel

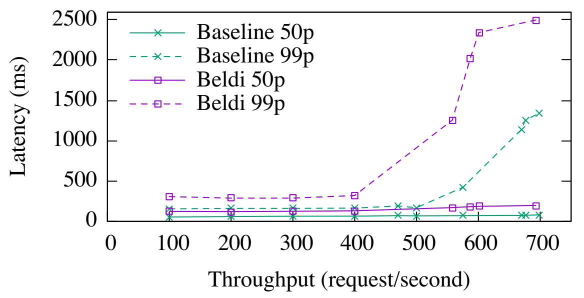


FIGURE 15—Median response time and throughput for travel reservation service. Dashed lines represent 99th-percentile response time. Beldi performs transactions over multiple SSFs to reserve a hotel room and a flight, while the baseline returns inconsistent results.

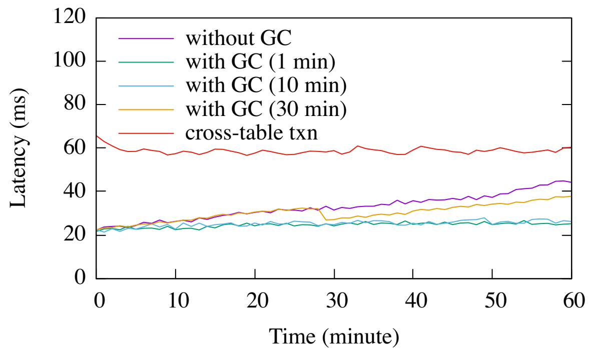


FIGURE 16—Median response time for an SSF that uses one `write` operation under different GC configurations. Without GC, the linked DAAL grows over time. As a baseline, we configure Beldi with cross-table transactions that do not use a linked DAAL.

reservation). At this high load, Beldi's 99th-percentile latency is only 20% higher for the movie review service, and 80% higher for the transaction-enabled travel site. We also test a configuration of the travel site that uses Beldi for fault-tolerance but without transactions. The median latency at saturation for that configuration is 16% lower and the 99th-percentile latency is 20% lower than Beldi with transactions.

7.5 What is the effect of garbage collection?

Finally, we evaluate the importance of the choice of garbage collector timeout (T) on performance. Note that this is different from the 1-minute timer that triggers the GC SSF (§7.2). T is instead proportional to the maximum lifetime of an SSF and determines when a GC can remove a row from the Linked DAAL. Thus, this value is important for safety, whereas the trigger only determines when the GC runs.

Since T is important to ensure exactly-once semantics, we could imagine performing a similar actuarial analysis to those involved in setting the end-to-end timeouts of reliable failure detectors [1, 27]. However, as Figure 16 shows, the median response times for SSFs that access the linked DAAL are only lightly impacted by the choice of T, even as we run the system for 30 minutes at constant load under pessimistic conditions (all SSF instances write to the same key). As a result, we can be relatively conservative about T. To be clear, this is a testament to the heroic efforts of DynamoDB engineers that have optimized its scan, filter, and projection operations. Nevertheless, we take some slight credit for ensuring that Beldi's linked DAAL is compatible with such operators.

It is worth noting, however, that while T has a minor impact on performance, it does impact storage overhead and I/O, since `read` and `write` operations still fetch a projection of the linked DAAL which scales with the number of rows (§7.3).

8 Discussion

We now discuss a few aspects of Beldi, such as the implication of relying on strongly consistent databases, the potential benefit of using SQL databases like Amazon Aurora, and the security implications of SSF federation and reusability.

Strongly consistent databases. Beldi enables developers to write stateful serverless applications without having to worry about concurrency control, fault tolerance, or manually making all of their functions idempotent. In doing so, Beldi leverages one or more fault-tolerant databases configured to be strongly consistent. If these databases were to become unavailable, for example due to network partitions, SSFs that write to these unavailable databases would also become unavailable until the partition was resolved.

ACID databases. A natural question is whether SSFs that use ACID databases need all of Beldi. For such SSFs, the benefit is not having to maintain a read or write log (or a linked DAAL) since the database does its own logging. However, ACID databases are not enough to guarantee exactly-once semantics for function invocations since they provide atomicity for read and write operations, but have no support for invocations. As a result, Beldi would still need to implement mechanisms such as callbacks (§4.5) to ensure that a failed SSF is not mistakenly re-executed despite independent garbage collectors. Furthermore, workflows that contain transactions *across* SSFs would still need a collaborative coordination protocol such as the one proposed in Section 6.2.

Independence of separate applications. We view SSFs as owning all the data on which they operate, similar to microservice architectures [11]. SSFs can isolate the state of different applications by storing each application's state on a different database. To ensure that a malicious request from one application cannot observe the state of another, standard authentication mechanisms such as capabilities and public key encryption could be used.

9 Related Work

We already discuss Beldi's differences with Olive [36] throughout. To summarize, Beldi builds upon Olive's elegant approach to fault tolerance and mutual exclusion, and adapts it to an entirely new domain. This adaptation is nontrivial and requires us to introduce new data structures, algorithms, and abstractions (e.g., transactions across SSFs). The result of our innovations is a simple API that SSF developers can use to build exciting applications without worrying about fault tolerance, concurrency control, or managing any infrastructure!

In the context of serverless, the observation that existing designs are currently a poor fit for applications that require state has been the subject of much prior work [15, 22, 24, 25, 43]. For example, Cloudburst [40] proposes a new architecture for incorporating state into serverless functions, and gg [15] proposes workarounds to state-management issues that arise in desktop workloads that are outsourced to thousands of serverless functions. However, the general approach to fault-tolerance in these works is to re-execute the entire workflow when there is a crash or timeout—violating exactly-once semantics if any SSF in the workflow is not idempotent.

AFT [39] is the closest proposal to Beldi and introduces a fault-tolerant shim layer for SSFs. However, AFT's deployment setting, guarantees, and mechanisms are very different. First, Beldi runs entirely on serverless functions, whereas AFT requires servers to interpose and coordinate all database accesses. As a result, Beldi can run on any existing serverless platform (or even in a multi-provider setup) without requiring any modification on their part and without the user needing to administer their own VMs. Second, Beldi seamlessly enables transactions within SSFs and across workflows with opacity, whereas AFT targets the much weaker (but more efficient) read atomic isolation level [4]. Due to the weaker isolation, it would be more difficult to implement our travel reservation system on AFT. Finally, Beldi allows SSFs to be managed independently and to keep their data private from each other, while AFT's servers manage all SSF data, handle failures and garbage collection, and serve as a central point of coordination for transactions.

10 Conclusion

Beldi makes it possible for developers to build transactional and fault-tolerant workflows of SSFs on existing serverless platforms. To do so, Beldi introduces novel refinements to an existing log-based approach to fault tolerance, including a new data structure and algorithms that operate on this data structure (§4.1), support for invocations of other SSFs with a novel callback mechanism (§4.5), and a collaborative distributed transaction protocol (§6). With these refinements, Beldi extracts the fault tolerance already available in today's NoSQL databases, and extends it to workflows of SSFs at low cost with minimal effort from application developers.

Acknowledgments

We thank the OSDI reviewers for their feedback and our shepherd, Jay Lorch, for going above and beyond and providing suggestions that dramatically improved the content and presentation of our work. We also thank Srinath Setty for many invaluable discussions and his help with Olive. This work was funded in part by VMWare, NSF grants CNS-1845749 and CCF-1910565, and DARPA contract HR0011-17-C0047.

References

[1] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[2] AWS Lambda. `https://aws.amazon.com/lambda/`.

[3] Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the ACM SIGMOD Conference*, June 2014.

[5] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[6] Cloud Bigtable overview of replication. `https://cloud.google.com/bigtable/docs/replication-overview`.

[7] Quotas and limits for Cloud BigTable. `https://cloud.google.com/bigtable/quotas`.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[9] Consistency levels in Azure Cosmos DB. `https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels`.

[10] Azure Cosmos DB service quotas. `https://docs.microsoft.com/en-us/azure/cosmos-db/concepts-limits`.

[11] C. de la Torre, B. Wagner, and M. Rousos. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, .NET and Visual Studio product teams, v3.1 edition, Jan. 2020. `https://docs.microsoft.com/en-us/dotnet/architecture/microservices/`.

[12] DeathStarBench. `https://github.com/delimitrou/DeathStarBench/`.

[13] Amazon DynamoDB read consistency. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html`.

[14] Limits in DynamoDB. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html`.

[15] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[16] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2019.

[17] Google Cloud Functions. Retrying background functions. `https://cloud.google.com/functions/docs/bestpractices/retries`.

[18] Google Cloud Functions. `https://cloud.google.com/functions`.

[19] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

[20] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2008.

[21] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Oct. 2001.

[22] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2019.

[23] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[24] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 2019.

[25] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2), June 1981.

[27] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2013.

[29] .Net Microservices Sample Reference Application. `https://github.com/dotnet-architecture/eShopOnContainers`.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1), 1992.

[31] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts Institute of Technology, 1981.

[32] S. Mu, S. Angel, and D. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[33] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating

concurrency control and consensus for commits under conflicts. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[34] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), Oct. 1979.

[35] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[36] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[37] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the ACM SIGMOD Conference*, 2019.

[38] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Sept. 2006.

[39] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2020.

[40] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. arXiv:2001/04592, Jan. 2020. `https://arxiv.org/abs/2001.04592`.

[41] AWS Step Functions. `https://aws.amazon.com/step-functions/`.

[42] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, Aug. 1995.

[43] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[44] wrk2: A constant throughput, correct latency recording variant of wrk. `https://github.com/giltene/wrk2`.

[45] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows (extended version). arXiv:2010/06706, 2020. `https://arxiv.org/abs/2010.06706`.

[46] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[47] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear. Practical non-blocking unordered lists. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Oct. 2013.

A Artifact Appendix

A.1 Abstract

Our artifact runs on Amazon AWS Lambda without additional requirements or dependencies. Deploying the code, performing the measurements, generating the plots, and running the benchmarks depend on some third-party frameworks including serverless, gnuplot and wrk2.

A.2 Artifact check-list

- **Program:** Golang
- **Run-time environment:** AWS Lambda
- **Metrics:** Throughput and latency
- **Experiments:** Our serverless port of DeathStarBench
- **Expected experiment run time:** Around 20 hours
- **Public link:** `https://github.com/eniac/Beldi`
- **Code licenses:** MIT License

A.3 Description

A.3.1 How to access

`https://github.com/eniac/Beldi`

A.4 Installation

A.4.1 Set up docker container

1. login to a registry

   ```
   $ docker login
   ```

2. pull the docker image

   for github packages users:

   ```
   $ docker run -it \
   > docker.pkg.github.com/eniac/beldi/beldi:latest
   /bin/bash
   ```

   for docker hub users:

   ```
   $ docker run -it tauta/beldi:latest /bin/bash
   ```

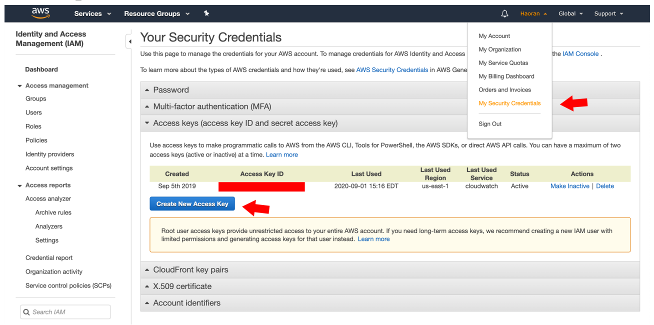
The purpose of this container is to setup the environment needed to run our configuration, deployment, and graph plotting scripts. The actual code of Beldi runs on AWS lambda.

A.4.2 Set AWS Credentials

Inside the container run

```
$ aws configure
```

It will ask you for an access key ID, a secret access key, region and output format. The first two can be found/created at:



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access the IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS Gene

Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

Use access keys to make programmatic calls to AWS from the AWS CLI, Tools for PowerShell, the AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more

Created	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Sep 5th 2019	[redacted]	2020-09-01 15:16 EDT	us-east-1	cloudwatch	Active	Make Inactive | Delete

Root user access keys provide unrestricted access to your entire AWS account. If you need long-term access keys, we recommend creating a new IAM user with limited permissions and generating access keys for that user instead. Learn more

CloudFront key pairs

X.509 certificate

Account identifiers

Set the region to us-east-1 and the output format to json.

A.5 Evaluation and expected result

A.5.1 Primitives (Figure 13)

To run the experiment

```
$ ./scripts/singleop/run.sh
```

The script has two modes

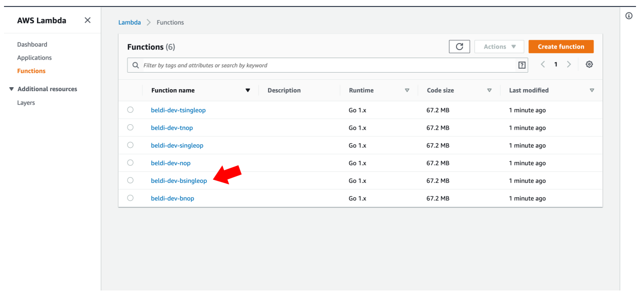
1. fast mode: less time, approximate result (around 5 min)
2. full mode: full experiment (around 30 min)

The script will ask you which mode to run when it starts.

Figure 13 includes three experiments, baseline (without beldi), beldi and beldi-txn. Their function names are bsingleop, singleop and tsingleop respectively. After deployment, the script will ask for the HTTP endpoint for these three lambdas, which needs manual setup at AWS.

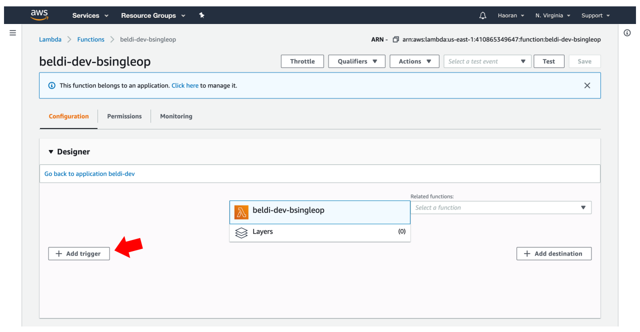
Take bsingleop as an example:

1. Go to the lambda console, click the function



Function name	Description	Runtime	Code size	Last modified
beldi-dev-tsingleop		Go 1.x	67.2 MB	1 minute ago
beldi-dev-tnop		Go 1.x	67.2 MB	1 minute ago
beldi-dev-singleop		Go 1.x	67.2 MB	1 minute ago
beldi-dev-nop		Go 1.x	67.2 MB	1 minute ago
beldi-dev-bsingleop		Go 1.x	67.2 MB	1 minute ago
beldi-dev-bnop		Go 1.x	67.2 MB	1 minute ago

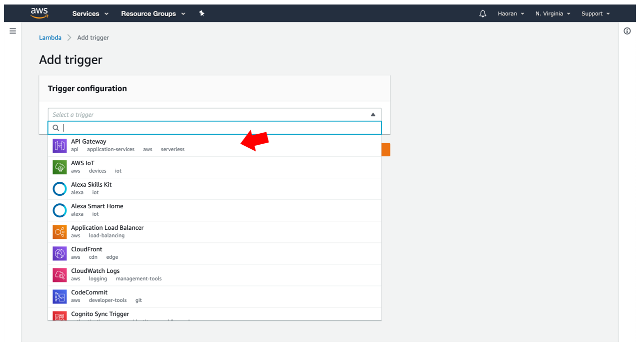
2. Click add trigger



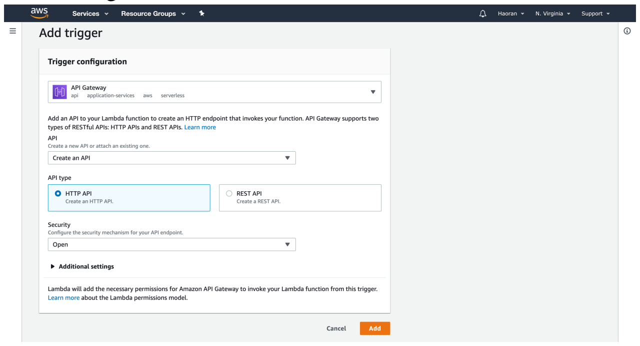
ARN - arn:aws:lambda:us-east-1:410865349647:function:beldi-dev-bsingleop

This function belongs to an application. Click here to manage it.

3. Choose API Gateway



4. Configure as below



Add trigger

Trigger configuration

API Gateway
api application-services aws serverless

Add an API to your Lambda function to create an HTTP endpoint that invokes your function. API Gateway supports two types of RESTful APIs: HTTP APIs and REST APIs. Learn more

API
Create a new API or attach an existing one.
Create an API

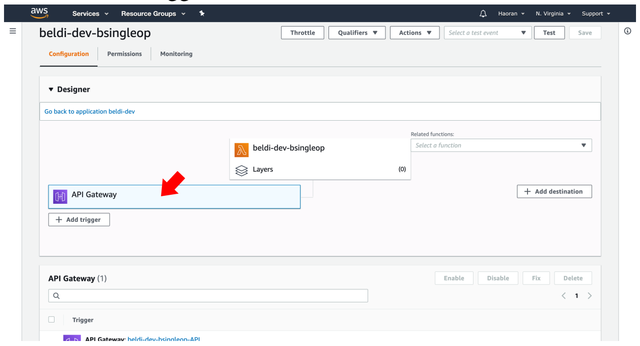
API type
[✓ HTTP API Create an HTTP API.] REST API Create a REST API.

Security
Configure the security mechanism for your API endpoint.
Open

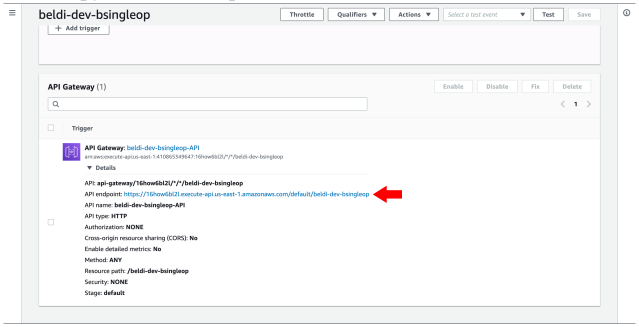
Additional settings

Lambda will add the necessary permissions for Amazon API Gateway to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

5. Click the trigger created



6. Copy the link and paste in terminal



API Gateway: beldi-dev-bsingleop-API
arn:aws:execute-api:us-east-1:410865349647:16how6bl2l/*/*/beldi-dev-bsingleop

API: **api-gateway/16how6bl2l/*/*/beldi-dev-bsingleop**
API endpoint: https://16how6bl2l.execute-api.us-east-1.amazonaws.com/default/beldi-dev-bsingleop
API name: **beldi-dev-bsingleop-API**
API type: **HTTP**
Authorization: **NONE**
Cross-origin resource sharing (CORS): **No**
Enable detailed metrics: **No**
Method: **ANY**
Resource path: **/beldi-dev-bsingleop**
Security: **NONE**
Stage: **default**

After all three endpoints get set, the experiment will start running. The result will be saved at `beldi/result/singleop/singleop`, which can be loaded by gnuplot

```
$ gnuplot < scripts/singleop/singleop.pg
```

The figure will show up as `beldi/result/singleop/res.png`. You can use docker cp to copy it to your host.

A.5.2 Garbage Collection (Figure 10)

To run the experiment,

```
$ ./scripts/gctest/run.sh
```

The script has two modes

1. fast mode: less time, prefix of Figure 10 (around 30 min)
2. full mode: full experiment (around 150 min)

The script will ask you which mode to run when it starts.

The script compiles the code and deploys the binary to AWS. After that, it will ask for the HTTP endpoint for **beldi-dev-gctest**. The result will be saved as `beldi/result/gctest/gc`.

To generate the figure,

```
$ gnuplot < scripts/gctest/gc.pg
```

The figure will show up as `beldi/result/gctest/res.png`.

A.5.3 Movie review service (Figure 14)

Baseline.

```
$ ./scripts/media/run-baseline.sh
```

Each data point takes around 20 min.

The script will first ask you for a request rate (the default is 100). After deployment, it will ask for the HTTP endpoint for **beldi-dev-bFrontend**. When it finishes, it will print to the terminal the median and p99 latency. This result will also be saved to `result/media/baseline.json`. Alternatively, you can view the metrics on AWS CloudWatch.

Beldi.

```
$ ./scripts/media/run.sh
```

A.5.4 Travel Reservation (Figure 15)

Baseline.

```
$ ./scripts/hotel/run-baseline.sh
```

Each data point takes around 20 min.

It will ask for the HTTP endpoint for **beldi-dev-bgateway**. When it finishes, it will print to terminal the median and p99 latency. It will also save the result to `result/hotel/baseline.json`.

Beldi.

```
$ ./scripts/hotel/run.sh
```