Doing More with Less: Orchestrating Serverless Applications without an Orchestrator

David H. Liu
Princeton University

Amit Levy
Princeton University

Shadi Noghabi
Microsoft Research

Sebastian Burckhardt
Microsoft Research

Abstract

Standalone orchestrators simplify the development of serverless applications by providing higher-level programming interfaces, coordinating function interactions and ensuring exactly-once execution. However, they limit application flexibility and are expensive to use. We show that these specialized orchestration services are unnecessary. Instead, application-level orchestration, deployed as a library, can support the same programming interfaces, complex interactions and execution guarantees, utilizing only basic serverless components that are already universally supported and billed at a fine-grained per-use basis. Furthermore, application-level orchestration affords applications more flexibility and reduces costs for both providers and users.

To demonstrate this, we present Unum, an application-level serverless orchestration system. Unum introduces an intermediate representation that partitions higher-level application definitions at compile-time and provides orchestration as a runtime library that executes in-situ with user-defined FaaS functions. On unmodified serverless infrastructures, Unum functions coordinate and ensure correctness in a decentralized manner by leveraging strongly consistent data stores.

Compared with AWS Step Functions, a state-of-the-art standalone orchestrator, our evaluation shows that Unum performs well, costs significantly less and grants applications greater flexibility to employ application-specific patterns and optimizations. For a representative set of applications, Unum runs as much as 2x faster and costs 9x cheaper.

1 Introduction

Serverless computing offers a simple but powerful abstraction with two essential components: a stateless compute engine (Functions as a Service, or FaaS) and a scalable, multi-tenant data store [27]. Developers build applications using stateless, event-driven "functions" which persist states in shared data stores. This abstraction allows users to leverage scalable data center resources with fine-grained per-invocation billing and frees them from server administration.

While serverless platforms originally targeted simple applications with one or a few functions, this paradigm has increasingly proven useful for more complex applications composed of many functions with rich and often stateful interaction patterns [19, 20, 25, 26, 40]. Unfortunately, building such applications using the basic FaaS is challenging. Event-driven execution makes depending on the results of multiple previous functions and therefore fan-in patterns difficult. At-least-once execution guarantee that is typical for FaaS functions complicates end-to-end application correctness as non-deterministic functions may pass inconsistent results downstream. Finally, the lack of higher-level programming interfaces for expressing inter-function patterns hinders application development.

Standalone orchestrators are recently introduced into the serverless infrastructure to support such complex applications (§2.1). Cloud providers commonly offer serverless orchestrators as a service [3, 6, 22, 23], though users may build custom orchestrators and deploy them in separate VMs or containers alongside their functions [19, 20, 38]. These orchestration services provide higher-level programming interfaces, support complex interactions and ensure exactly-once execution.

Though often internally distributed, standalone orchestrators operate as *logically centralized* controllers. Developers provide a description of an execution graph—nodes in the graph represent FaaS functions and edges represent invocations of a function with the output of one or more functions—and the orchestrator drives the execution of this graph by invoking functions, receiving function results and storing application states (e.g., outstanding invocations and function results) centrally.

Centralization simplifies supporting stateful interactions—e.g., an orchestrator can run fan-in patterns by simply waiting for all branches to complete before invoking an aggregation function. Similarly, an orchestrator can ensure that applications appear to execute exactly-once by choosing a single result from multiple executions for each function invocation.

However, standalone orchestrators have important drawbacks for both serverless providers and serverless users. As an additional service that is critical to application performance and correctness, a standalone orchestrator is expensive to host and use. User-deployed orchestrators risk under-utilization and do not benefit from serverless' per-use billing. Provider-hosted orchestrators are multi-tenant and can thus multiplex over many users to improve resource utilization and amortize the cost. However, they still incur the costs of hardware resources and on-call engineering teams. These costs may be affordable for large platforms but can be a significant burden for smaller providers.

Furthermore, standalone orchestrators preclude users from making application-specific trade-offs and optimizations. While the interface and implementation of an orchestrator might efficiently support the needs of many applications, it cannot meet all applications' needs, resulting in a compromise familiar from operating systems [9,13], networks [17,39], and storage systems [21,28].

For example, applications that need orchestration patterns not supported by the provider-hosted orchestrator have to either compromise performance by using less-efficient patterns or first repeat the hard work of building, deploying and managing their own custom orchestrator. A video processing application that encodes video chunks and aggregates results of adjacent branches in parallel has to compromise performance if the orchestrator only supports aggregating results of all branches.

Similarly, applications that consist entirely of deterministic functions, such as an image resize application for creating thumbnails or an IoT data processing pipeline for aggregating sensor readings, can tolerate duplicate executions without weakening correctness. However, with a standalone orchestrator that always persists states to ensure exactly-once execution, this application would incur the overheads of strong guarantees regardless.

In this paper, we show that additional standalone orchestrators for serverless applications are unnecessary. Furthermore, we argue that application-level orchestration is better for both serverless providers and developers. It is better for developers as it affords applications more flexibility to implement custom patterns as needed and apply application-specific optimizations. It is better for providers as it obviates the need to host an additional complex service and frees up resources such that providers can focus on fewer, core services in their serverless infrastructure. Moreover, application-level orchestration built on top of existing storage and FaaS services in the serverless infrastructure can benefit automatically from improvements to cost and performance to these services.

To support these arguments, we present Unum, an application-level serverless orchestration system. Unum provides orchestration as a library that runs *in-situ* with user-defined FaaS functions, rather than as a standalone service. The library relies on a minimal set of existing serverless APIs—function invocation and a few basic data store operations—that are common across cloud platforms. Unum introduces an intermediate representation (IR) language to express execution graphs using only node-local information and supports front-end compilers that transform higher-level application definitions into the IR.

A key challenge in Unum is to support complex stateful orchestration patterns and strong execution guarantees in a *decentralized* manner. Our insight is that, scalable and strongly consistent data stores, already an essential building block of serverless applications, address the hardest challenge of orchestration: coordination. Using such data stores, we show that an application-level library running in-situ with user functions can orchestrate complex execution graphs efficiently with strong execution guarantees.

At a high level, Unum relies on the FaaS scheduler to run each function invocation *at least* once and consistent data store operations to coordinate interactions and de-duplicate extra executions of the same invocation. Unum uses checkpoints to commit to exactly one result for a function invocation and ensures workflow correctness despite duplicate executions of non-deterministic functions. Unum fan-ins use objects in a consistent data store as coordination points for aggregating branches. Both require generating globally unique names for nodes and edges in the execution graph *locally* (using only information available at each node) as well as cleaning up intermediate data store objects in a timely manner.

Our implementation of Unum (§4) includes a compiler for AWS Step Functions' description language, enabling Unum to run arbitrary Step Function workflows. We show that Step Function workflows compiled to Unum execute with the same execution guarantees as running natively using the Step Functions orchestrator.

Moreover, while performance and cost are difficult to compare objectively with existing black-box production orchestrators—both are influenced by deployment and pricing decisions that may not reflect the underlying efficiency or cost of the system—Unum performs well in practice (§5). We find that a representative set of applications run faster and cost significantly less with Unum than Step Functions (Table 2). We also demonstrate that Unum's IR allows applications to run faster by using application-specific optimizations and supporting a richer set of interaction patterns.

2 Background & Motivation

The basic serverless abstraction is simple and quite powerful. Developers build "functions", typically written in a high-level language and packaged as OS containers or virtual machines, which run short computations in response to a platform event. Events include storage events (e.g., the creation of an object) or HTTP requests. The platform can scale resources for each function to respond to instantaneous bursts in events and developers are absolved from capacity planning and resource management tasks.

This simple abstraction can be used to compose many simple applications with one or a few functions. For example, developers can chain functions for data pipelines using triggers. In trigger-based composition [10] each function in a chain invokes the next asynchronously or writes to a data store configured to invoke the next function in response to storage events. Alternatively, developers might use a "driver-function" [40] to drive more intricate control-flow logic. A driver function acts as a centralized controller that invokes other functions, waits for their outputs, and invokes subsequent functions with their outputs.

Such ad-hoc approaches work "out-of-the-box", that is, they require no additional platform provided infrastructure. However neither is well suited to complex applications with 10s or 100s of functions [20, 35]. Trigger-based composition can only support chaining of individual functions or fan-out from one function to multiple, but cannot, for example, fan-in from multiple functions to one. Moreover, trigger-based composition scatters control-flow logic across each function or in configured storage events, making development unwieldy when application complexity grows.

On the other hand, driver functions concentrate control flow in a single function and support arbitrary composition. However, most serverless platforms impose modest runtime limits on individual functions, and thus driver functions restrict the total runtime of applications. Furthermore, driver functions suffer "double billing" since they are billed for the entire call-graph execution despite spending most time idly waiting for callees to return.

Finally, both ad-hoc approaches require developers to handle function crashes, retries and duplicate invocations gracefully [1, 7, 8, 14]. Application typically want to ensure "exactly once" semantics [10, 11, 24, 25, 40] for an entire call-graph, but failures and multiple invocations of individual functions can subvert this goal without careful consideration.

2.1 Standalone Orchestrators

A common solution to address the needs of complex serverless applications is to introduce a workflow orchestrator that provides a high-level programming interface with support for a rich set of patterns (e.g., branching, chaining, fan-out and fan-in) [3, 6, 19, 20, 22, 23, 38]. Many cloud providers offer serverless orchestrators as a service [3, 6, 22, 23] or users can build custom orchestrators [19, 20, 38] and deploy them in VMs alongside their functions.

Similar to driver functions, orchestrators operate as *logically centralized* controllers. They drive a workflow by invoking its functions and hosting application states such as function outputs and outstanding invocations.

However, different from driver-functions, orchestrators are standalone services. Orchestrators are not limited by function timeouts and can be arbitrarily long-running [4]. Moreover, as standalone services, orchestrators are often internally distributed and employ techniques such as replication and sharding to provide strong execution guarantees, fault-tolerance and scalability. For example, orchestrators can ensure that workflows appear to execute exactly-once by choosing one result for each function invocation, even if FaaS engines only guarantee at-least once execution. Orchestrators can also persist or replicate states during execution so that in face of orchestrator failures, applications do not lose executions or retry from the beginning.

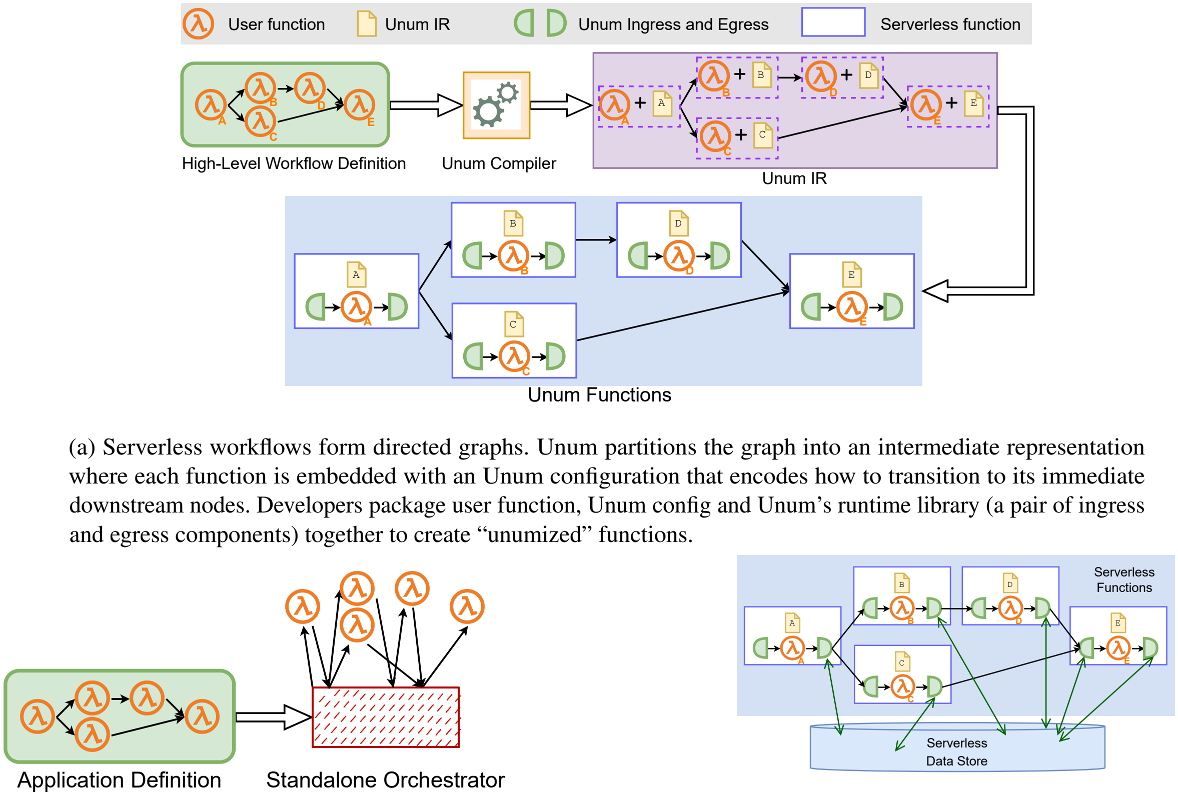
While orchestrators are able to address the needs of complex serverless applications, introducing a new standalone service has significant drawbacks. Building performant, scalable and fault-tolerant multi-tenant systems is hard and orchestrators introduce yet-another potential performance and scalability bottleneck. Indeed, we find that, in practice, production systems limit end-to-end performance for highly-parallel applications (§ 5).

Moreover, hosting such services is expensive. Deploying a custom orchestrator per user risks under-utilization as it cannot multiplex over many users and users pay even when the orchestrator is not actively in use, breaking the fine-grained billing benefit of serverless. Provider-hosted orchestrators are multi-tenant and can amortize this cost. But they still incur engineering expenses as they require teams on-call. Indeed, we find that provider-hosted orchestrators cost developers significantly and dominate the total cost of running applications(§ 5).

Lastly, provider-hosted orchestrators preclude users from making application-specific optimizations. Each provider typically offers just a single orchestrator service option. While the interface and implementation of the orchestrator might efficiently support many applications, it cannot meet all applications' needs, resulting in a compromise familiar from operating systems [9, 13], networks [17, 39], and storage systems [21, 28]. Indeed, we find that provider-hosted orchestrators force applications to compromise performance by using less-efficient patterns (§ 5).

3 Design

Unum is an application-level orchestration system that supports complex serverless applications without a standalone orchestrator. It does so by decentralizing orchestration logic in a library that runs in-situ with user-defined FaaS functions and leverages a scalable consistent data store for coordination and execution correctness. By removing standalone orchestrators, Unum improves application flexibility and reduces costs. Importantly, Unum does this while retaining the expressiveness and execution guarantees (§3.3) of standalone orchestrators.



(a) Serverless workflows form directed graphs. Unum partitions the graph into an intermediate representation where each function is embedded with an Unum configuration that encodes how to transition to its immediate downstream nodes. Developers package user function, Unum config and Unum's runtime library (a pair of ingress and egress components) together to create "unumized" functions.

(b) A typical standalone orchestrator operates as a logically centralized controller that drives the execution of applications by invoking functions, receiving function results and storing application states

(c) At runtime, Unum orchestration logic is decentralized and runs in-situ with the user functions on an unmodified serverless platform. For coordination and checkpointing, Unum relies exclusively on a standard data store of choice, such as DynamoDB or Cosmos DB.

Figure 1: Unum's Decentralized Orchestration. Unum partitions orchestration logic at compile time and a Unum runtime runs in-situ with user functions to perform only the orchestration logic local to its node.

3.1 Architecture

Figure 1a depicts how developers run serverless workflows using Unum. Developers write individual functions and describe the workflow using a high-level workflow language, such as Step Functions' expression language. An Unum frontend compiler uses these to extract portable Unum IR for each node in the graph and "attaches" it to the function (e.g. by placing a file containing the IR alongside the function code). A platform-specific Unum linker "links" each function with a platform-specific Unum runtime library. [1] Developers deploy each linked function along with its IR to the FaaS platform.

Each Unum workflow begins with an "entry" function. Invoking this function (e.g. using an HTTP or storage trigger) starts a workflow. Moreover, admission control rules for the workflow, such as access control and rate limiting, are implemented by setting appropriate rules on this entry function. For example, a workflow can be invoked by a particular principal if the entry function is exposed to that principal.

The runtime library is composed of an ingress and egress component that run before and after the user-defined function and unwrap and wrap the results of the function in Unum execution state, respectively (Figure 2). The ingress component coalesces input data from each incoming edge (e.g. in a fan-in), resolves input data if passed by name rather than by value, and passes the input value to the function. The egress component uses the function's result to invoke the next function(s), enforces execution semantics using checkpoints, performs coordination with sibling branches in fan-in, and deletes intermediate states no longer needed for the workflow, executing the workflow in-situ with the functions, in lieu of a centralized orchestrator (Figure 1c).

3.2 Unum Intermediate Representation

Similar to many standalone orchestrators, Unum applications are modeled as directed execution graphs where nodes represent user-defined FaaS functions and edges represent function invocations (incoming edges) with the output of one

[1] Since functions are typically written in dynamic languages, the Unum library source code is placed alongside the function and dynamically imported, rather than statically linking an object file

`Invoke(Fn)`	Queue a single invocation of a function.
`Map(Fn)`	Queue one function invocation for each element in the current function's result.
`FanIn(Set, Size, Fn)`	Queue a fan-in to a function using the provided coordination set object and size.
`Pop`	Pops the top frame of the execution state stack (passed via Unum requests).
`Next`	Increments the current execution state frame's iteration counter.
`CreateSet(Name)`	Creates a coordination set object in the data store with the provided name.

Table 1: Unum intermediate representation instructions.

or more other functions (outgoing edges).

An Unum graph may include fan-outs, where a node's output is used to invoke several functions or split up and "mapped" multiple times on the same function. Each such branch may be taken conditionally, based on the output value or dynamic states of the graph. Execution graphs may also contain fan-ins, where the outputs of multiple nodes are used to invoke a single aggregate function. Cycles are also supported and each iteration through a cycle is a different invocation of the target function.

The Unum intermediate representation (IR) is designed to encode directed execution graphs in a way that both allows decentralization of orchestration and is low-level enough to support application-specific patterns.

Each function's IR includes the function's name and a sequence of instructions (Table 1). Instructions direct the runtime to invoke functions and operate on Unum metadata passed between functions (Figure 2).

The egress component, which receives the function's user-code output, executes the IR and uses it to determine which next steps to take. An invocation can be protected by a conditional—a boolean expression that operates on the invocation request and the current function's output. Unum's IR provides three kinds of invocations:

- **Invoke** simply invokes the named function using the current functions output.
- **Map** treats the current function's output as iterable data (e.g. a list) and invokes the named function once for each item in the output.
- **FanIn** invokes the named function using the current function's output along with the outputs of all other functions fanning into the same node. Fan-in requires coordination among multiple functions and is described in detail in §3.4.

```
struct InvocationRequest {
  data: Vec<DatastoreObjectName>,
  workflowId: String,
  fanOut: Stack<FanOut>,
}

struct RequestData {
  reference: DatastoreObjectName,
  value: Option<Value>
}

struct FanOut {
  index: usize,
  size: usize,
  iteration: usize,
}
```

Figure 2: An Unum request wraps function outputs with metadata that allows function invocations to be named uniquely and assists in coordinating fan-ins. Unum IR instructions can reference and modify this metadata.

When multiple invocations occur, either using multiple instructions or a single `Map` invocation, each of the invocations adds a fan-out frame to the invocation request's fan-out stack. This allows different invocations of the same function to be differentiated for naming (§3.6) and to coordinate fan-in (§3.4).

The IR also includes instructions for manipulating the Unum request data and an instruction that creates a new coordination set, typically for use in later nodes to coordinate fan-in (§3.4) or garbage collection (§3.5).

This IR is sufficient to represent basic patterns, as well as more complex fan-in patterns (described in §3.4).

Chain & Fan-out. Unum encodes passing the output of a function to one (chain) or multiple (fan-out) subsequent functions, simply, with one or more calls to the `Invoke` instructions.

Map. Applications may perform the same operation on each component of a function's output. For example, an application may unpack an archive of high-resolution images in one function and perform compression on each of the images. Unum's `Map` instruction invokes the same `Fn` for each element of a function's output.

Branching. Applications may need to invoke different functions based on runtime conditions (e.g., the output of a function). For instance, an application may first validate that a user-uploaded photo is a valid JPEG. If it is, it invokes, e.g., one of the patterns above, otherwise it notifies the user of the error. Unum's invocation instructions are optionally protected by a conditional expression that has access to the function output and execution metadata (Figure 2).

3.3 Execution Guarantees Using Checkpoints

FaaS functions only provide weak execution guarantees. Functions can fail mid-execution and be retried. Even in the absence of failures, one function invocation may result in more than one execution because most FaaS engines only ensure at-least-once execution. This is problematic for applications whose functions are non-deterministic because a single workflow invocation can produce multiple *diverging* outputs.

An important benefit of orchestrators is strong execution semantics such that applications appear to execute *exactly-once* even if individual functions in the application run multiple times. Because standalone orchestrators are logically centralized, guaranteeing exactly-once is conceptually straightforward: the orchestrator can choose a single result from executions of the same invocation and use it as input for all downstream functions. At the end of the workflow, the result is consistent with an execution of the workflow where each function invocation executed exactly-once.

A key challenge for Unum is to provide the same semantics without centralizing orchestration. Moreover, because failures and, thus, retries are the exception, not the rule, Unum should provide these semantics without expensive coordination—function instances should be able to proceed without blocking in the common, fault-free case.

Unum leverages two key insights to achieve these semantics. First, it is correct for different executions of the same function invocation to return different results as long as Unum ensures downstream functions are always invoked with exactly one of those results. Second, a workflow's output is *correct* even if a function is invoked more than once, as long as the invocations uses the same input, since additional, but identical, invocations are indistinguishable from additional executions.

The Unum library employs an atomic `create_if_not_exists` operation in the serverless data store to *checkpoint* exactly one execution of each function invocation. The egress component of the Unum library attempts to write the result of the function to a checkpoint object in the data store. If such a checkpoint already exists, a concurrent or previous execution of the invocation must have already completed and the operation will fail. To invoke downstream functions, the egress component *always* uses the value stored in the checkpoint, rather than the result of the recently completed function. Essentially, Unum "commits" to result of the first successful executions of invocations.

Data stores need to be strongly consistent to support `create_if_not_exists`. It is important that a later attempt to create an existing checkpoint fails and the slower execution can read the existing checkpoint.

As a further optimization, the ingress component in the Unum library checks for the checkpoint object before executing the user-defined function. If the object exists, it bypasses the user-defined function and passes the checkpoint value directly to the egress component to invoke downstream functions. This is not necessary for correctness but helps reduce computation that we know will go unused.

```
def ingress(self, function):
    ...
    result = datastore_get(self.checkpoint_name):
    if result:
        self._egress(result)
    else:
        self.egress(function.handle())

def egress(self, result):
    ...
    if not datastore_atomic_add(self.checkpoint_name, result):
      result = datastore_get(self.checkpoint_name)
    self._egress(result)
    ...

def _egress(self, result)
    for f in next_functions:
        faas.async_invoke(f, result)
```

Figure 3: Pseudo-code showing Unum's checkpointing mechanism. As different executions of a function may return different results, Unum's egress component checkpoints the first successful execution using an atomic add data store operation. All subsequent executions will uses this committed value rather than the result their own execution returned.

Note that the exactly-once guarantee does not automatically extend to applications with external side effects, i.e. functions that directly call external services. In such cases, retries can lead to unexpected results if the effects are not idempotent. This issue is well known, and independent of the orchestrator architecture (centralized vs. decentralized). Thus, we consider the question of how to control such side effects to be orthogonal and beyond the scope of this paper. However, Unum does not preclude applications from using libraries, such as Beldi [40], that can solve this problem.

3.3.1 Fault Tolerance

Another source of multiple executions is retrying failed functions. Retries in Unum rely on FaaS engines' error handling support. All popular FaaS engines provide error handling so that applications do not just crash silently without a way to react to failures. Common mechanisms include "automatic retry" that re-executes the same function [7, 14, 32, 34] or failure redirection that triggers a pre-configured error-handler function [29, 33]. Unum can work with either mechanism.

The Unum error handler is part of Unum's standard library and is triggered in a separate FaaS function after an application function crashes. The error handler simply retries the crashed function by invoking it again. As part of the orchestration library, the error handler is assumed to be bug-free and relies on the FaaS scheduler to execute at least once.

Unum's checkpointing mechanism ensures that while faults may occur at any point during the execution of a function's user code or the Unum library, and while downstream functions may be invoked multiple times by different executions of the same invocation, a single value is always used to invoke downstream functions.

If there is a fault after the user code completes but before creating the checkpoint, user code result is ignored (indeed, never seen) by other executions and another execution's value will be used to invoke downstream functions. If the "winning" function crashes after creating a checkpoint, and before invoking some or all downstream functions, other executions will use the checkpoint value to invoke downstream functions. Finally, even if multiple executions invoke some or all downstream functions, execution guarantees are still satisfied as these invocations will have identical inputs.

3.4 Fan-in Patterns

In fan-in patterns, the results of multiple nodes are used to invoke a single head node. Such patterns are a particular challenge for decentralized orchestration because invoking the target function cannot happen until all branches complete, but there is no standalone orchestrator to wait for this condition. Designating one of the tail nodes as the coordinator would address this directly. However, there is no guarantee that branches for a fan-in complete soon after each other, incurring a potentially large resource cost to do virtually no work, and risk exceeding platform-enforced function timeouts. Moreover, functions typically cannot communicate with each other directly, so it is not obvious how other branches would notify this coordinator of their completion.

Unum, instead, leverages the same insight as checkpoints—the data store provides strong consistency that can serve as a coordination point. Rather than designating a single branch function as the coordinator, all branches are empowered to invoke the fan-in function once all other branches have completed. To determine this condition, branches in a fan-in add the name of their checkpoint object to a shared "Set" in the data store. Any branch that reads the set with size equal to the total number of branches invokes the target function using all the branches' checkpoints as input.

Importantly, functions do not wait for any other to complete. As long as all functions complete eventually (in other words, they run at-least once), *some* function will read a full set and invoke the fan-in target function. More than one function may observe this condition, resulting in multiple invocations, but these invocations will be identical and are handled as spurious executions of the same invocation (§3.3).

In order to perform this coordination, branches must know the branching factor—the size of the set. The `FanIn` instruction includes this size, which is either specified explicitly, or using a variable from the invocation request, commonly the fan-out size.

Similar to checkpoints, the set data structure for coordination requires the data store to be strongly consistent. Updates to a set must be immediately visible to other branches otherwise the downstream fan-in function may ever be invoked. Moreover, the data store must support data structures that can implement a "set" abstraction.

Fan-in supports enable more patterns that commonly arise in applications:

Aggregation. After processing data with many parallel branches, applications commonly want to aggregate results. For example, to build an index of a large corpus, the application might process chunks in parallel and then aggregate the results. Aggregation is a common pattern to join back multiple parallel functions, by invoking a single "sink" function with the outputs from a vector of functions.

Fold. `fold` sequentially applies the same function on the outputs of a vector of source functions, while aggregating with the intermediate results of running the function so far. For example, a video encoding application might encode chunks in parallel and then concatenate the results in order: concatenating chunk 1 and 2, then concatenating chunk 3 to chunk [1–2], and so on. `fold` is an advanced pattern that is not supported by all existing systems (e.g., AWS Step Functions do not support `fold`) but is expressible in Unum.

3.5 Garbage Collection

Both checkpointing and fan-in require storing intermediate data (e.g., checkpoints and coordination sets) in the data store. These intermediate data is only temporally useful and grows with each invocation. This poses a garbage collection challenge. Deleting them too early can compromise execution guarantees while deleting too late incurs storage costs.

Checkpoint Collection. A checkpointing node does not know when *its* checkpoint is no longer necessary. If it deletes its checkpoint immediately after invoking subsequent functions, it may crash and the FaaS platform may re-execute it, yielding a potentially inconsistent result. However, downstream nodes know that once they have committed to a value by checkpointing, previous checkpoints are no longer necessary to ensure their own correctness. Once a node has committed to some particular output, future invocations, even with *different* inputs will produce the same output, as the node will *always* use the checkpointed value.

Note that a duplicate execution that checkpoints after the previous checkpoint is garbage collected has the same semantics as a separate invocation. It may result in multiple outputs from the workflow, though each output is still consistent with an execution of the workflow where each function was invoked exactly-once. Any GC policy, no matter how conservative, might lead to multiple executions if the FaaS platform could execute duplicates of a function invocation after an arbitrarily long time in the future.

Therefore, Unum collects checkpoints by relaxing the constraint that nodes always output the same value. Instead, they must only output the same value until all subsequent nodes have committed to their own outputs. This means that, in non-fan-out cases, once a node checkpoints its result, it can delete the previous node's checkpoint.

Fan-outs are more complicated because deleting the checkpoint must wait until all branches have committed to an output. Unum repurposes the same set-based technique from fan-in to collect checkpoints in fan-out cases as well. The originating node of a fan-out creates a set for branches to coordinate when to delete its checkpoint. Branches add themselves to the set after checkpointing their own value. Any node that reads a full set deletes the parent's checkpoint as well as the set. This guarantees that the parent's checkpoint is deleted and ensures that all branches have first checkpointed.

Note that it is possible for one of the branches to re-execute *after* the set has been deleted. This is safe because it is the origin of the fan-out that creates the set, so a branch's attempt to add itself to a, now, non-existent set will simply fail.

Fan-in Set collection. Deleting sets used for fan-in works much like removing checkpoints—the target node of a fan-in deletes the set once it has generated a checkpoint. However, who *creates* the set?

If each branch in the fan-in creates the set if it doesn't already exist, a spurious execution of one of the branches *after* the fan-in target removes the original set will create a new one that is never deleted (because it never fills, and thus the target function is never invoked again). To avoid this, Unum places the responsibility to create the set on the node that originates the *fan-out* at the same level as the target node.

3.6 Naming

Much of Unum's functionality relies on unique naming. A workflow invocation must be named to differentiate it from other concurrent invocations of the workflow; functions must be named to invoke them; different invocations of functions must have different names to uniquely name invocation checkpoints and coordination sets for fan-in.

Each workflow invocation has a unique name that is passed through the execution graph. The name is either generated in the ingress to the first function using, e.g., a UUID library or, when available, is taken from the FaaS platform's invocation identifier for the first function. This enables functions to have different names when invoked as part of a specific workflow invocations. The function's name is either user-defined or determined by the FaaS platform (e.g. the ARN on AWS Lambda) and determined at "compile-time" (i.e. when generating Unum IR).

However, this is not sufficient as functions may be invoked multiple times in the same workflow due to map patterns—which invoke the same function multiple times over an iterable output—and cycles. Moreover, invocation names must be determined using local information only. Once running, each function only has access to it's own code (including the IR) and metadata passed in its input. Nonetheless a particular invocation must be able to determine its own name for checkpointing as well as, if it is part of a fan-in, the name of downstream invocations to coordinate with other branches.

As a result, Unum names function invocations using a combination of the global function name, a vector of branch indexes and iteration numbers (taken from the Unum request fan-out stack) leading to the invocation, and the workflow invocation name. Function names are global and the remaining items are propagated by Unum in invocation arguments.

During a fan-out pattern (multiple scalar invocations or a map invocation), a branch index is added to a list in the next functions' input. If the next function is an ancestor of the current function (a cycle), an iteration field in the input is incremented. Note that a single iteration field is sufficient even if there are nested cycles since it is only important that different invocations of the same function have *different* names, not that the iteration field is sequential. Thus, a monotonically increasing iteration field is sufficient.

We note that the format of this name is not significant and, importantly, it need not be interpretable. It must only be deterministic and unique for its inputs. For example, a reasonable implementation could serialize the inputs and take a cryptographic hash over the result, guaranteeing uniqueness (with very very high probably) while preventing names from growing too large to use as object names.

4 Implementation

We implement a prototype Unum runtime that supports AWS and Google Cloud. We also implement a front-end compiler that transforms AWS Step Function definitions to Unum IR. Currently our runtime only supports Python functions and is itself written in 1,119 lines of code. The Step Functions compiler is 549 lines-of-code.

Implementing the runtime primarily requires specializing high-level functionality the IR depends on for a particular FaaS platform and data store. The FaaS platform must support asynchronous invocation and the data store must be strongly consistent with support for atomic creation and set operations.

Importantly, we choose data stores and primitives that only incur per-use costs and scale on-demand. For example, we use DynamoDB in on-demand capacity mode, rather than provisioned capacity mode, and avoid long-running services such as a hosted Redis or cache. As a result, Unum incurs fine-grained costs only when performing orchestration (e.g., per-millisecond Lambda runtime costs to execute the Unum library, per-write DynamoDB costs to create checkpoints).

4.1 AWS Lambda & DynamoDB

Asynchronous invocation in Lambda is natively supported. In particular, the Lambda `Invoke` API is asynchronous when passed `InvocationType=Event`. In the event of a crash, we use Lambda's Failure Destination [29] to redirect the fault to an error handler function which runs just the Unum runtime. The error handler checks if the failed function should be retried (e.g., based on the Step Function definition [15]) and if so, retries the function by explicitly invoking it again.

DynamoDB organizes data into tables, with each item in a table named by a key. Within tables, items are unstructured by default. Our implementation of Unum uses a single table for each workflow. Each item in the table corresponds to a checkpoint, or a coordination set for fan-in or garbage collection.

DynamoDB supports atomic item creation by passing the conditional flag `attribute_not_exists` to the `put_item` API call. We use this for creating both checkpoint blobs and coordination sets. DynamoDB supports set addition natively using the `Map` field type. In particular, we use update expressions to atomically set a named map element to true. As an optimization, we use the `ALL_NEW` flag when adding to a set to atomically get the new value after a set in a single operation.

4.2 Google Cloud Functions & Firestore

Google Cloud Functions (GCF) do not have an asynchronous invocation API. Instead, we allocate function-specific pub-sub queues and subscribe each function to its respective queue. Unum then asynchronously invoke a function by publishing the input data as an event to the function's queue.

GCF supports automatic retry for asynchronous functions [34]. In the event of a crash, the Unum runtime in the retry execution checks if the failed function should be retried and if so, retries the function by explicitly invoking it again.

Firestore organizes data into logical collections (which are created and deleted implicitly) containing unstructured items, named by a unique key. Similar to DynamoDB, we use a separate collection for each workflow. Atomic item creation is supported using a special `create` API call, which only succeeds if the key does not already exist. Firestore supports an `Array` field type which can act as a set by using the `ArrayUnion` and `update` operation, which atomically sets the field to the union of its existing elements and the provided elements. The `update` operation always returns the new value data.

5 Evaluation

Unum argues for eschewing standalone orchestrators and, instead, building application-level orchestration on unmodified serverless infrastructure using FaaS schedulers and consistent data stores. In this section, we evaluate how well application-level orchestration performs, reduces costs, and improves application flexibility. In particular, we focus our performance evaluation on whether decentralization comes at a reasonable overhead compared with standalone orchestrators.

Specifically, we answer the following questions:

1. What overhead does Unum incur in end-to-end latency and what are the sources of Unum's overheads?

2. How much does it cost to run applications with Unum and what are the sources of costs compared with Step Functions?

3. How well does Unum support applications that Step Functions cannot support well?

Though we evaluate the applications running on both AWS and Google Cloud, we focus our discussion on our AWS implementation with Lambda and DynamoDB, because it runs on the same serverless infrastructure as Step Functions.

5.1 Experimental setup

We run all experiments on AWS with Lambda and DynamoDB, and on Google Cloud with Cloud Functions and Firestore. All services are in the same region (`us-west-1` on AWS and `us-central-1` on Google Cloud). All functions are configured with 128MB of memory except for ExCamera where we use 3GB of memory to replicate the setup in the original paper [19, 20]. DynamoDB uses the on-demand provisioning option that charges per-read and per-write [12]. To avoid performance artifacts related to cold-starts, we ensure functions are warm by running each experiment several times before taking measurements.

All but one application were originally written as Step Function state machines. For Step Function experiments, we ran them directly with the "Standard" configuration [37], which provides similarly strong execution guarantees as Unum [16]. For Unum experiments, we first compiled the Step Functions definitions to Unum IR, linked the functions with the Unum runtime library and finally executed them as lambdas or Google Cloud functions. The notable exception is our Unum and Step Functions implementations of ExCamera, which differ due to a limitation in the Amazon State Language. As a result, the more efficient Unum implementation is written directly in Unum IR instead of compiled from the Step Functions definition (§ 5.4).

5.2 Performance

Unum's performance overhead results from the Unum runtime logic run in each function as well as API calls to data stores and FaaS engines. We characterize these overheads by measuring the latency to execute various patterns consists of `noop` functions as well as end-to-end performance of real applications. Overall we find that Unum performs comparably or significantly better than Step Functions in most cases owing to higher parallelism and a more expressive orchestration language, with modest slow downs in the remaining cases due to implementation deficiencies.

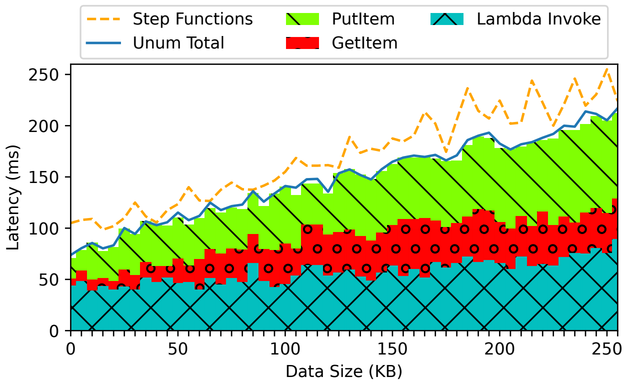


Figure 4: An orchestrator incurs a latency on each transition between functions. Unum's overhead is due to storage operations to ensure exactly-one-result semantics, Lambda invocation API overhead to enqueue the next function to run, and additional Unum runtime code in the function instance itself for the orchestration logic.

5.2.1 Chaining

For the simple chaining pattern, the Unum runtime performs a storage read to check whether a checkpoint already exists, a storage write to checkpoint the function's result, and an asynchronous function invocation to initiate the next function in the chain.

Figure 4 shows time to perform each of these operations for different result sizes. As expected, storage operations are slower when checkpointed results are larger, but the total overhead from the Unum runtime operations is consistently lower than an equivalent Step Function transition.

The Unum implementation of the IoT pipeline application benefits from this difference, with the Unum version running 1.9x faster than the Step Functions version (Table 2).

5.2.2 Fan-out and fan-in

Fan-out requires the same number of storage operations as chaining and similar orchestration logic, but the Unum runtime performs an additional asynchronous invocation at the source function for each branch. For fan-in patterns, each source branch performs an additional storage read to determine if it is the final branch to execute, and only the final branch performs the asynchronous invocation of the target function.

Figure 5 shows the latency of a fan-out followed by a fan-in at varying branching degrees for both Unum and Step Functions. At low branching degree, Unum incurs a modest overhead (up to 200ms) relative to Step Functions. We believe this is mostly due to our implementation initiating each branch invocation sequentially. However, at higher branching degrees (as low as 20 branches), Step Functions limits the number of outstanding fan-out branches [30] while Unum is limited only by Lambda's scalability, resulting in over 4x lower latency with 512 branches.

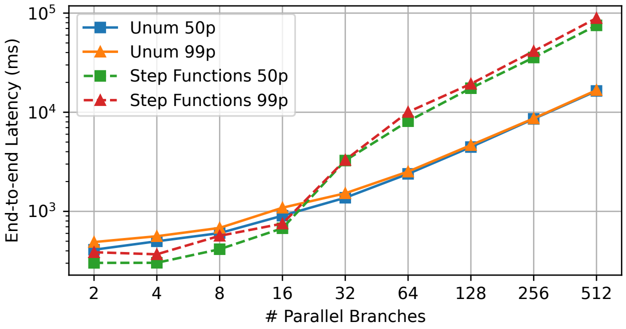


Figure 5: End-to-end latency of a fan-out and fan-in pattern with increasing branching degree. Unum is slower at lower branching degrees but significantly outperform Step Functions at moderate and high branching degree.

These differences manifest in real workloads as well. Wordcount is highly parallel (with 262 parallel mappers and 250 reducers) and performs over 2x faster on Unum than on Step Functions (Table 2).

Although it may not be the case that standalone orchestrators fundamentally have to impose limits on the number of outstanding function invocations, this example shows that it is at least not trivial to ameliorate the constraint. On the other hand, as a library, Unum is free from the need to design and implement yet-another service that supports parallel applications well, but can instead provide as much parallelism as FaaS schedulers and data stores permit. FaaS schedulers and data stores already support highly-parallel applications well, and Unum's performance will improve automatically when these underlying services further improve.

5.3 Cost

One of the main attractions of building applications on serverless platforms is fine-grained and often lower cost. In particular, because resources are easy to reclaim, applications are charged only for resources used to respond to actual events. Thus, the *cost* of orchestration matters as well as performance.

The source of costs for Unum and Step Functions is quite different. Step Functions imposes a cost to developers for each workflow transition [5], such as each branch in a fan-out. This abstracts the underlying, likely shared, costs to run the Step Functions servers, persist states and checkpoint data. Conversely, Unum incurs costs directly from those services. In particular, compute resources for executing orchestration logic is charged per-millisecond such as Lambda runtime cost [2] and storage for persisting states is charged per read and write such as DynamoDB reads and writes [12].

On AWS, Unum is much cheaper than Step Functions—AWS's native orchestrator. For a basic transition in a chaining

	Latency (seconds)			**Costs ($ per 1 mil. executions)**		
App	*Unum-aws*	*Unum-gcloud*	*Step Functions*	*Unum-aws*	*Unum-gcloud*	*Step Functions*
IoT Pipeline	0.12	0.81	0.23	$12.38	$6.3	$112.02
Text Processing	0.52	3.56	0.55	$60.42	$31.7	$225.29
Wordcount	408.88	484.12	898.56	$13,433.67	$11,727.3	$18,141.19
ExCamera	84.52	122.63	98.42	$62,684.29	$51,617.2	$114,633.13

Table 2: Application latency and costs comparison between Unum and Step Functions. Running applications on Unum is 1.35x to 9x cheaper than on Step Functions. Furthermore, Unum is faster than Step Functions especially for workflows with high degrees of parallelism.

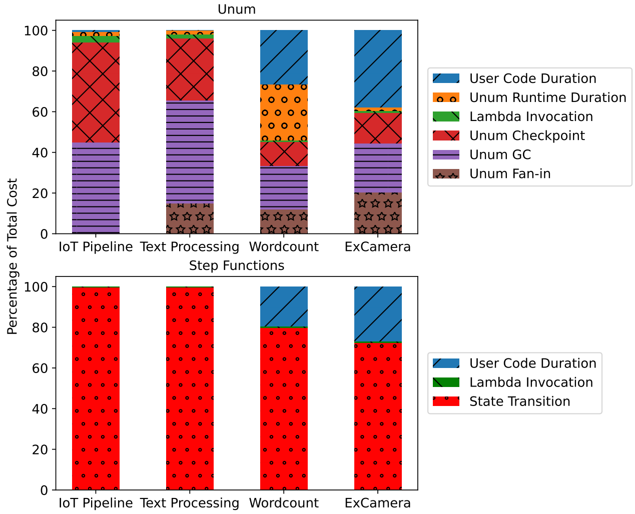


Figure 6: Step Functions state transitions dominate the total costs for all applications (99.5% in IoT Pipeline, 99.4% in Text Processing, 80.0% in Wordcount, 72.2% in ExCamera). While Unum runtime cost is also the majority, it accounts for a smaller portion of the overall costs (95.7% in IoT Pipeline, 97.8% in Text Processing, 72.5% in Wordcount and 61.0% in ExCamera).

pattern, Step Functions charges $27.9 per 1 million such transitions. On the other hand, Unum costs, for 1 million transitions, (1) $0.42 for ∼200ms extra Lambda runtime to execute orchestration library code, (2) $2.79 for 1 DynamoDB write to checkpoint, (3) $0.279 for 1 DynamoDB read to check checkpoint existence, and (4) $2.79 for 1 DynamoDB write to garbage collect the checkpoint. In total, a basic transition in Unum is about 4.4x cheaper than the provider-hosted orchestrator on the same platform ($27.9 vs $6.279).

Table 2 shows the cost to run each of the applications we implemented in the `us-west-1` region. Unum is consistently, and up to 9x, cheaper than Step Functions for the applications we tested.

Figure 6 shows the cost to run each application using Unum broken down to each component: data store costs for writing and reading checkpoints, data store costs for writing coordination sets, data store costs for deleting checkpoints and writing coordination sets for garbage collection, Lambda invocation, and Lambda CPU-time for both the Unum runtime and user function. Storage costs, using DynamoDB, are the largest portion of overall cost and costs for writing to DynamoDB is the majority[2]. This includes writing checkpoints, writing to coordination set (either for fan-in for garbage collection) and deleting checkpoints for garbage collection.

Of course, developer-facing pricing is only a proxy for *actual* costs of hardware and human resources. However, it is clear that, in practice, Unum's costs are reasonable and, in fact, often lower than Step Functions. This suggests that at least applications that currently run on Step Functions could afford to run using Unum instead.

Furthermore, services that Unum builds on—FaaS schedulers and data stores—are core multi-tenant services that likely multiplex over a larger audience of applications than orchestrators for greater economies of scales. These services typically have enjoyed long periods of improvement already to make them efficient. Unum's design obviates the need to host yet-another service which frees up resources such that providers can focus on fewer core services in their serverless infrastructure.

Moreover, Unum automatically benefits from improvements to the underlying infrastructure and pricing schemes. For example, Azure's Cosmos DB provides similar performance and consistency guarantees to DynamoDB but charges 5x less to perform a write operation (the dominant cost of Unum's data store operations).

5.4 Case Study: ExCamera

ExCamera [20] is a video-processing application designed to take advantage of high burst-scalability on Lambda using custom orchestration. We compare our Unum implementation with three others: (1) the original hand-optimized ExCamera using the mu framework, (2) an implementation using a generalized orchestrator (gg) by the same authors, and (3) an optimized Step Functions implementation we wrote.

Both gg and mu employ standalone orchestrators to proxy inter-function communications, store application states and

[2] Writes in DynamoDB cost about an order-of-magnitude more than reads

invoke lambdas. However, mu uses a fleet of long-running identical lambdas where all application code is co-located and raw video chunks are pre-loaded, whereas gg lambdas are event-driven, task-specific and cannot leverage pre-loading. The application logic, though, is identical for gg ExCamera and mu ExCamera. Unum's ExCamera replicates the application logic from gg and mu. However, the Step Functions ExCamera implementation must serialize the encode and re-encode stages because Step Function's Map pattern requires all concurrent branches to complete before any fan-in starts (Figure 7).

5.4.1 Performance

Using the same experimental setup as the prior work (i.e., encoding the first 888 chunks of the `sintel-4k` [31] video using 16 chunks per batch and Lambdas configured with 3GB of memory), Unum is 7.1% faster than gg [19] and 10.5% slower than the original, hand-optimized ExCamera (Table 3). The original authors attributes the slower performance of gg ExCamera to the lack of pre-loading which is likely also the reason for Unum's slower performance.

But different from gg, Unum executes orchestration in a decentralized manner while gg has a standalone coordinator on EC2. The reduced number of network communications likely explains why Unum is slightly faster.

Comparing with Step Functions, Unum's design allows the flexibility to implement ExCamera's original application pattern where tasks start as soon as their input data becomes available, whereas the Step Functions implementation had to use the less-efficient Map pattern without the flexibility to add new orchestration patterns easily. As a result, the Unum ExCamera enables more parallelism between branches and is 16.7% faster than Step Functions.

5.4.2 Cost

Unlike Unum, neither gg nor mu aimed to reduce the cost of running serverless applications and neither discussed costs in detail. Nevertheless, there are several important factors in comparing Unum with gg and mu in relation to costs.

First, similar to Step Functions, gg and mu both rely on standalone orchestrators. Thus, the fundamental costs difference is also similar, namely Unum's use of storage vs gg's and mu's use of VMs. mu's orchestrator consists of a coordinator server as well as a rendezvous server [20], while gg's only has a coordinator server [19]. In the mu authors' experiments, they used a 64-core VM (`m4.16xlarge`) as the rendezvous server. Neither mu nor gg specified the instance type of its coordinator server. However, the cost of the rendezvous server, at the time of writing, is $3.20 per hour, or approximately $2352 per month.

Furthermore, standalone orchestrators must separately consider fault-tolerance in case of orchestrator failures. Most commonly, fault-tolerance is achieved by running multiple coordinating instances (replicas) of the service. As a result, production deployments of mu and gg would likely cost more.

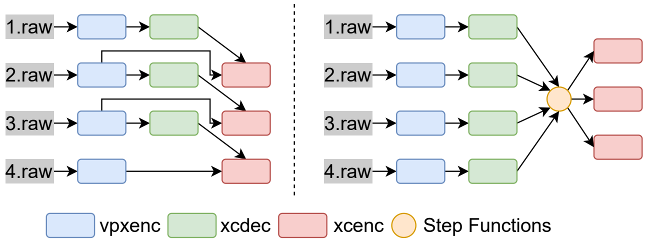


Figure 7: Unum ExCamera replicates the application logic from gg and mu where the re-encode stage (xcenc) of a branch can start immediately when the previous branch completes decoding (xcdec) and my own branch completes the initial encoding (vpxenc). Step Functions provides a Map pattern [30] for parallel workloads. However, branches in Map must be identical and Map does not support data dependencies between branches. As a result, to ensure previous branches' xcdec have completed, all branches must first finish and fan-in to Step Functions before starting the xcenc step, essentially serializing the stage.

ExCamera Implementation	Latency (seconds)
Original	76
Unum-aws	84
gg	90
Step Functions	98

Table 3: ExCamera performance. Unum is 7.1% faster than gg [19] and 10.5% slower than the hand-optimized implementation.

Lastly, deploying an orchestrator per application or per user limits the ability to amortize costs through multi-tenancy. A provider-hosted orchestrator, such as Step Functions, can achieve larger economies of scale by serving many users concurrently with a single deployment.

6 Related Work

Serverless Workflows. Many systems have recognized the need to augment serverless computing with support for composing functions to build larger and more complex applications. AWS Step Functions [3] defines serverless workflows as state machines using a JSON schema. Google Workflows [23] uses a YAML-based interface to list steps in a workflow sequentially and allows jumps among steps. Azure Durable Functions [6] uses a "workflow-as-code" approach, similar to driver functions, where the workflow logic is written in a programming language (e.g., C#, Python).

In all of these systems, orchestration is performed by a

standalone orchestrator. The nature and location of this component varies: in AWS Step Functions [3] and Google Workflows [23], it is provided by a cloud service that is separately hosted and billed. In Azure Durable Functions [6], it is an extension of the serverless runtime, and uses the same billing. In contrast to all of these, Unum proposes a novel **decentralized orchestration strategy** and runs entirely on unmodified serverless infrastructure without adding any new services or new components.

Kappa [41] addresses the lack of coordination between function and function timeout limits when executing large applications. Similar to Durable Functions, it also exposes a high-level programming language interface. Cloudburst [36] uses a specialized key-value store to enable low-latency execution of serverless functions. Users can express workflows as static DAGs and an executor program runs the DAG by passing data and coordinate via the key-value store. ExCamera [20] proposes the mu framework which uses a long-running coordinator to command a fleet of lambdas, each of which executes a state machines where user functions are the states. gg [19] proposes a thunk abstraction where each thunk executes as a deterministic lambda and expresses data dependencies between thunks as DAGs. gg uses a standalone coordinator to receive thunk updates and lazily launch thunks when their inputs become available.

Similarly, the above systems rely on a standalone orchestrator program. As the orchestrator program is not itself executing in a hosted environment, progress is not guaranteed when its host crashes. Also, progress is not checkpointed (except in Kappa), so workflows must restart from the beginning in that situation. In contrast, Unum relies only on a basic, highly available serverless platform. Thus, it guarantees progress under all faults, including the orchestrator. And Unum checkpoints each function result to minimize redundant computations when handling faults.

Beldi [40] and Boki [25] are two recent systems that provide exactly-once execution and transactions to stateful serverless applications. Both extend transactional features to specific application side effects supported by the system (e.g., DynamoDB writes). Developers use Beldi or Boki's library in user code when writing to a supported data store (e.g., DynamoDB) such that writes are executed only once. In comparison, Unum does not change how developers write user code and does not extend exactly-once guarantee to side effects in user code. Instead, Unum treats user code as a black box and ensures exactly-once semantics on a workflow-level. However, Unum users who want to ensure exactly-once when writing to DynamoDB can additionally use Beldi or Boki in their user code.

Programming Interface. Most serverless workflow systems require developers to write workflows with specialized interfaces. Some uses a declarative approach that defines workflows using JSON or YAML schemas (e.g., AWS Step Functions [3], Google Workflows [23]). Others allow expressing workflow as code (e.g., Durable Functions [6], Kappa [41], Fn Flow [18]).

Unum does not propose a new frontend for defining workflow. Instead, Unum aims to support any existing frontend that explicitly or implicitly expresses a directed graph where nodes are functions and edges are transitions between functions. Developers using Unum can choose the frontend that they prefer.

7 Discussion & Limitations

Unsupported applications. Unum supports a superset of applications that can be expressed using Step Functions, but there are applications that do not fit Unum's constraints. In particular, Unum only supports statically defined control structures. For example, Durable Functions expresses workflows dynamically as code and allows the developer to use arbitrary logic to determine what the next workflow step should be at runtime. This is not currently possible with Unum.

Measurement error. Due to the opaque design, implementation and pricing of production workflow systems, such as Step Functions, comparisons in our evaluations are limited in their explanatory power. In particular, we use the current *price* of Lambda, DynamoDB, and Step Functions as a proxy for the *cost* of providing these services. Of course, prices may be either lower or higher for a particular service than the underlying cost.

Code Complexity. While Unum affords users more flexibility, application-level orchestration increases code complexity for developers. Coordination and exactly-once execution require careful design and implementation to function correctly in a decentralized manner. Introducing application-specific optimization also needs additional developer efforts than using off-the-shelf patterns from provider-hosted orchestrators.

8 Conclusion

We designed and implemented Unum, an application-level, decentralized orchestration system that runs as a library on unmodified serverless infrastructure without requiring additional services. Our results show that basic serverless components—function schedulers and consistent data stores—are sufficient abstractions for building complex and fault-tolerant serverless applications. Moreover, Unum affords applications more flexibility, reduces costs and performs well compared with standalone orchestrators with similar execution guarantees.

Acknowledgments We thank the anonymous reviewers and our shepherd, Douglas Terry, for their insightful comments. We thank Landon Cox for his support and feedback during the early stage of this project. This work was funded in part by NSF Grant 2028869.

References

[1] Asynchronous invocation, AWS Lambda Developer Guide. https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html.

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

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

[4] AWS Step Functions Quotas. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html.

[5] AWS Step Functions Pricing. https://aws.amazon.com/step-functions/pricing/.

[6] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[7] Azure Functions error handling and retries, Azure Functions Developers Guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages?tabs=csharp.

[8] Azure Functions reliable event processing, Azure Functions Developers Guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reliable-event-processing#how-azure-functions-consumes-event-hubs-events.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995.

[10] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, apr 2022.

[11] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[12] DynamoDB Pricing for On-Demand Capacity. https://aws.amazon.com/dynamodb/pricing/on-demand/.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, dec 1995.

[14] Error handling and automatic retries in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html.

[15] Error handling in Step Functions, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html.

[16] Execution guarantees, Standard vs. Express Workflows, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/express-at-least-once-execution.html.

[17] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[18] Fn Flow. https://fnproject.io/.

[19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[20] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[21] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 323–336, USA, 2010. USENIX Association.

[22] Google Cloud Composer (GCC). https://cloud.google.com/composer.

[23] Google Workflows. https://cloud.google.com/workflows.

[24] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[25] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems*

Principles, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[28] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, October 2018. USENIX Association.

[29] Introducing AWS Lambda Destinations. https://aws.amazon.com/blogs/compute/introducing-aws-lambda-destinations/.

[30] Map State, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html.

[31] MPI Sintel Flow Dataset. https://paperswithcode.com/dataset/mpi-sintel.

[32] OpenFaaS Retries for functions. https://docs.openfaas.com/openfaas-pro/retries/.

[33] OpenWhisk Actions, Error Handling. https://github.com/ibm-cloud-docs/openwhisk/blob/master/error-handling.md.

[34] Retrying Event-Driven Functions, Google Cloud Functions. https://cloud.google.com/functions/docs/bestpractices/retries.

[35] Arnav Sankaran, Pubali Datta, and Adam Bates. Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference*, ACSAC '20, page 496–509, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[37] Standard vs. Express Workflows, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html.

[38] Temporal Platform. https://docs.temporal.io/.

[39] David Tennenhouse. Active networks. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[40] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[41] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery.