A Serverless Architecture for Application-Level Orchestration

Hao Liu

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Sciences
Adviser: Amit Levy

January 2023

Abstract

This thesis examines the problem of building large-scale applications using the serverless computing model and proposes decentralized, application-level orchestration for serverless workloads. We demonstrate that application-level orchestration is possible and practical using just the basic APIs of existing serverless infrastructures, and benefits both cloud users and cloud providers, compared with standalone orchestrators, the state-of-the-art solution to building large-scale serverless applications. It empowers cloud users with the flexibility of application-specific optimizations. It frees cloud providers from hosting and maintaining yet another performance-critical service. Furthermore, the performance and efficiency of application-level orchestration improve as the underlying systems develop. Thus, cloud providers can direct freed-up resources to core services in their serverless infrastructure and automatically reaps the benefits of a better orchestrator.

This thesis describes mechanisms and implementations that help realize the goal of application-level orchestration. In particular, we explain the necessity and challenges of decentralizing orchestration and present a system for decentralized orchestration named Unum. Unum introduces an intermediate representation (IR) language to express execution graphs using only node-local information to decentralize the orchestration logic of applications. Unum implements 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 datastore operations—that are common across cloud platforms. Unum ensures workflow correctness despite multiple executions of non-deterministic functions by using checkpoints to commit to exactly one output for a function invocation.

Our results show that a representative set of applications scale better, run faster, and cost significantly less with Unum than a state-of-the-art centralized orchestrator. We also show that Unum's IR allows hand-tuned applications to run faster by using application-specific optimizations and supporting a richer set of application patterns.

We hope the results of this thesis inspire cloud practitioners to reconsider the approach of supporting new functionalities by simply adding more services to the cloud infrastructure. And we hope to encourage the building of other application-level orchestration systems from the serverless community.

Acknowledgements

I would like to thank my advisor, Amit Levy, for his feedback, advice, and support throughout my graduate study. I learned a great deal from him over the many projects we worked on together. I am particularly grateful for his flexibility and accommodation during the challenging time of COVID.

This thesis builds on research I have conducted in collaboration with Shadi Noghabi and Sebastian Burckhardt at MSR. I want to thank both Shadi and Sebastian for their invaluable contributions and for being wonderful colleagues. Additionally, I would like to thank Landon Cox for guiding me in my early days of research at Duke, for hosting me at MSR, and for giving me the support and freedom to explore what eventually became the Unum project and this thesis.

I also want to thank my other committee members, Mike Freedman, Jennifer Rexford, Wyatt Lloyd, and Ravi Netravali. They all provided valuable guidance and feedback during my study at Princeton. Special thanks to Mike Freedman for taking me on at the beginning of my Ph.D. study. I learned and benefited greatly from his emphasis on being precise about the research questions.

Over my time at (and away from) Princeton, I am fortunate to be around (in-person or virtually) amazing peers at the SNS group. Ashwini, Haonan, Zhenyu, Yue, Logan, Harris, Nan, Andrew, Chris, Jeff, Khiem, Sam, and so many others have all given me valuable feedback and help on my research. I am thankful that I get to stay in touch with many of them despite the interruption from COVID. Graduate school would have been much less stimulating and satisfying without them.

Finally, I wish to give wholehearted thanks to my wife, Rowena. None of this would have been possible without her love and support.

To my wife Rowena Gan,

for her abiding love and suppport

Contents

List of Figures

List of Tables

Chapter 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 [31]. Developers build applications using stateless, event-driven "functions" which persist states in shared data stores.

Ease of use and fine-grained usage-based billing are key appeals of serverless computing over traditional "serverful" options (e.g., EC2 virtual machines). A serverless developer is relieved from virtually all system administration tasks such as provisioning, managing configurations, fault tolerance, and planning for peak demands. Instead, serverless platforms automatically scale applications in response to request loads and charge users in small compute and storage increments (e.g., per millisecond of CPU time or per stored object or data store operations).

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 [22, 23, 29, 30, 45]. Unfortunately, building such applications using 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 hinder application development.

Standalone orchestrators are recently introduced into the serverless infrastructure to support such complex applications. Cloud providers commonly offer serverless orchestrators as a service [6, 9, 25, 27], though users may build custom orchestrators and deploy them in separate VMs or containers

alongside their functions [22, 23, 43]. 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 underutilization 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 to operating systems [12, 16], networks [20, 44], and storage systems [24, 32].

For example, applications that need orchestration patterns not supported by the provider-hosted orchestrators have to either compromise performance by using less-efficient patterns or first repeat the hard work of building, deploying, and managing their custom orchestrators. 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.

This dissertation shows 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 the cost and performance of 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 promptly.

Our implementation of Unum 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. We find that a representative set of applications run faster and cost significantly less with Unum than with Step Functions (Table 4.2). 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.

We hope the results of this thesis inspire cloud practitioners to reconsider the approach of supporting new functionalities by simply adding more services to the cloud infrastructure. Instead, we hope to encourage the community to further explore the potential of the serverless paradigm and alternative designs of application-level orchestration frameworks.

Chapter 2

Background

2.1 Serverless Computing and Simple Serverless Applications

Popularized by Amazon's release of the AWS Lambda service in 2015, serverless computing has since seen increasingly greater adoption—every major cloud platform now has its own serverless offering—and the transition to serverless has been compared to the evolution from assembly language to high-level programming languages in the field of cloud computing [31].

Ease of use and fine-grained usage-based billing are key appeals of serverless computing over traditional "serverful" options (e.g., EC2 virtual machines). A serverless developer is relieved from virtually all system administration tasks such as provisioning, managing configurations, fault tolerance, and planning for peak demands. Instead, serverless platforms automatically scale applications in response to request loads and charge users in small compute and storage increments (e.g., per millisecond of CPU time or per stored object or data store operations). For cloud users, this new paradigm directly accelerates application development and lowers costs. Table 2.1 lists a few main differences between serverless computing and traditional "serverful" computing.

In the serverless paradigm, developers build applications in the form of *stateless* functions that run in response to platform events and persist data in shared data stores. Figure 2.1 depicts an example serverless application that consists of a function that creates thumbnails of large images uploaded to an object store. The developer simply needs to upload the function to the serverless platform to make it available to use. The serverless platform takes care of provisioning resources to execute the application and scales the application automatically in response to varying request loads.

Characteristic	*Serverless*	*Serverful*
Scaling	Cloud provider responsible	Cloud user responsible
Deployment	Cloud provider responsible	Cloud user responsible
Fault Tolerance	Cloud provider responsible	Cloud user responsible
Server Instance	Cloud provider selects	Cloud user selects
When the program is run	On event selected by user	Continuously until explicitly stopped
Minimum Accounting Unit	1 millisecond	1 minute

Table 2.1: Difference of serverless cloud functions vs. serverful cloud VMs. Adapted from Table 2 in [31]

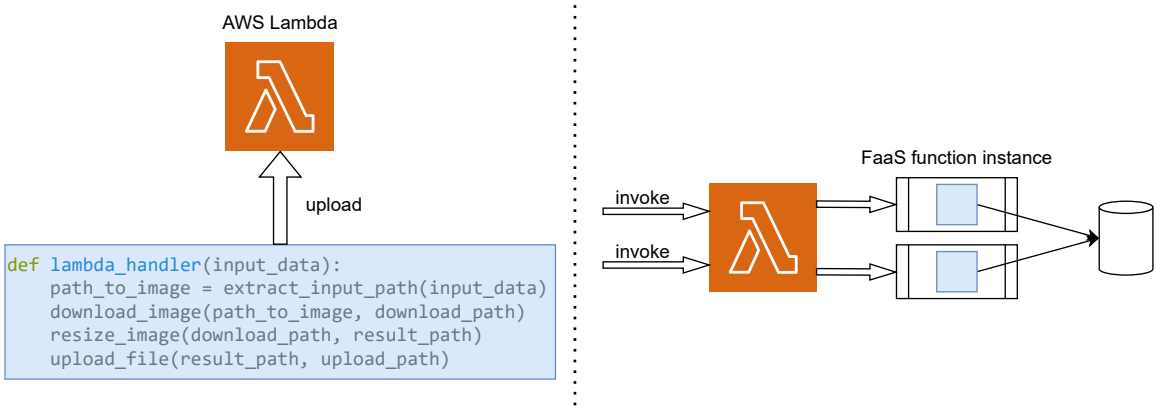


Figure 2.1: Serverless developers do not need to manage servers. Instead, applications are deployed by simply uploading a zip of the application code to serverless platforms. Serverless platforms automatically scale applications in response to request loads and charge users in small compute and storage increments.

2.2 The Serverless Abstraction

While design and implementation details differ across platforms, all FaaS systems adhere to the same abstraction of serverless computing with two defining characteristics. First, all serverless platforms adhere to an architecture that decouples computation and storage. Computation and storage are implemented by different services that are provisioned and scale separately and priced independently. Figure 2.2 illustrates this architecture.

Computation is provided in the form of Function-as-a-Service (FaaS) systems and is *stateless*. Developers build their applications as one or more "functions" that are typically written in a high-level language and packaged as OS containers or virtual machines. FaaS systems instantiate functions by creating containers loaded with application code and executing them. FaaS functions are stateless because any data written by applications to their container's local storage (e.g., virtual disk) does not persist.

Storage is provided via a variety of cloud data stores such as object stores, databases, and key-value stores. To write data persistently, applications need to explicitly use cloud data stores of

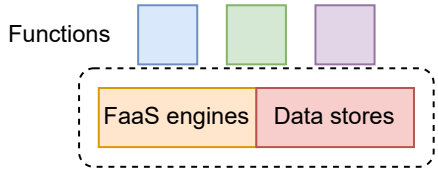


Figure 2.2: All serverless platforms adhere to an architecture that decouples computation and storage. Computation and storage are implemented by different services that are provisioned and scaled separately and priced independently.

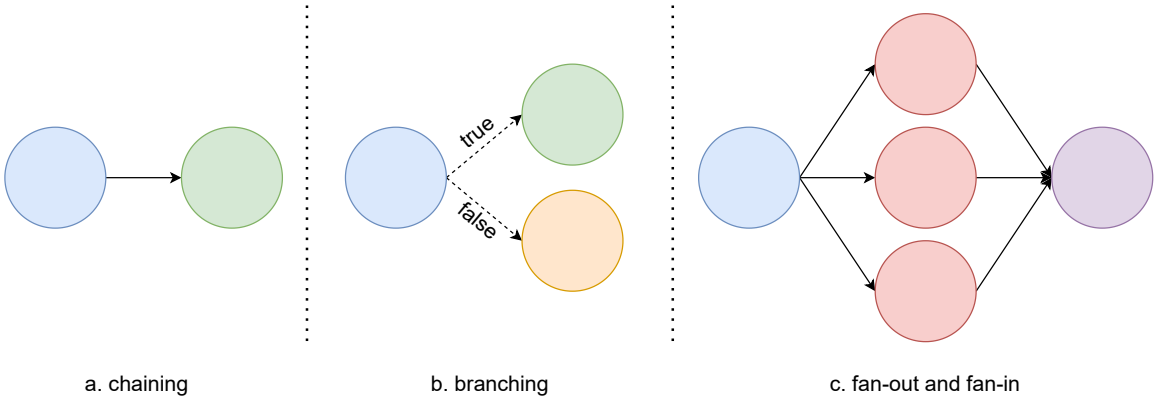


Figure 2.3: Common inter-function interaction patterns include chaining, branching, fan-out, and fan-in.

their choice in their function code. Cloud providers have been building serverless data stores and also making many existing data stores serverless [2, 3, 26] such that users do not need to manage provisioning and are charged based on usage rather than provisioned capacities.

Secondly, FaaS systems all have a single API for invoking function execution. The semantics of this invocation API can be generalized as `invoke(function_name, input_data)`, similar to that of a regular language-level function call. Calling `invoke` causes FaaS systems to allocate computation resources, in the form of containers or virtual machines, load the application, and execute it. This API allows FaaS systems to autoscale applications on a fine, per-request granularity and, with the help of lightweight containers and virtual machines [1], at a much faster speed and higher efficiency.

There are many options on how to call `invoke`. Many platforms support events from storage services. For example, creating a new object in S3 can trigger an event that calls the `invoke` API. In this case, S3 is the caller of `invoke`. Moreover, the caller can choose to wait synchronously for the response from the callee function or not which makes the call asynchronous.

2.3 Complex Serverless Applications

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 [22,23,29,30,45]. Common inter-function interaction patterns include chaining, branching, fan-out and fan-in.

For instance, an Internet-of-Things (IoT) application that collects sensor data about the environment and adjusts an actuator based on the data might consist of a chain of two functions. The first function cleans and processes the data to extract needed aggregate statistics, and the second function decides how to adjust the actuator based on the first function's output. For another example, the user registration service of a social network site might need a branching pattern. It has a function that processes user account registration. If the registration is successful, it invokes another function to render the user's homepage. If the registration fails, it invokes a different function to ask the user to try again, for instance by changing the desired username.

Highly-parallel applications are particularly good fits for serverless and many new applications emerged to take advantage of serverless' high burst scalability. These applications often use fan-out and fan-in patterns that process data in many parallel functions and aggregate results at the end. For instance, ExCamera [23] is a new video encoder that parallelizes the traditionally sequential work of video encoding. It relies on a fan-out pattern where a large raw video is broken into thousands of small chunks, each of which is encoded by a separate FaaS function instance. Outputs from those parallel FaaS function instances are then aggregated (fan-in) to create the final encoded video. Such an application utilizes FaaS' burst scalability advantage that can start thousands of parallel functions quickly, without any provision work from the developer.

Unfortunately, building such applications using 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.

People have tried to use the basic FaaS system to compose some simple applications with one or a few functions in ad-hoc manners. For example, developers can chain functions for data pipelines using triggers. In trigger-based composition [13] each function in a chain invokes the next asynchronously or each function writes to a data store configured to invoke the next function in response to the

storage event. Alternatively, developers might use a "driver function" [45] to drive more intricate control-flow logic. A driver function acts as a centralized controller that invokes other functions, waits for their responses, and invokes subsequent functions with their responses.

Such ad-hoc approaches work "out-of-the-box", i.e. they require no additional platform-provided infrastructure. However neither is well suited to complex applications with 10s or 100s of functions [23, 40]. The 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 [4, 10, 11, 17]. Applications typically want to ensure "exactly once" semantics [13, 14, 28, 29, 45] for an entire call graph, but failures and multiple invocations of individual functions can subvert this goal without careful consideration.

2.4 Serverless Orchestrators

Complex serverless applications call for a high-level programming interface that can easily express a rich set of inter-function interactions and a system that can execute applications with strong end-to-end correctness guarantees.

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) [6,9,22,23,25,27,43]. Many cloud providers offer serverless orchestrators as a service [6,9,25,27] or users can build custom orchestrators [22,23,43] 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

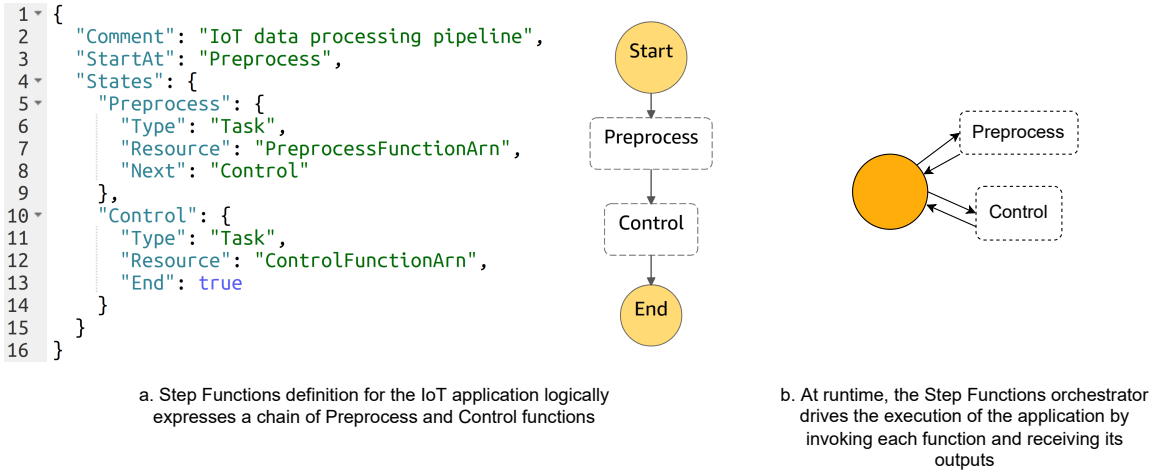


Figure 2.4: Standalone orchestrators provide a high-level programming interface that express inter-function interactions and runs as a logically centralized controller to drive the execution of applications at runtime.

not limited by function timeouts and can be arbitrarily long-running [7]. 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.

Take AWS Step Functions [6], a state-the-of-art standalone orchestrator, as an example. It uses a JSON-based declarative interface to express inter-function interactions. Applications are represented as state machines where states are FaaS functions and transitions between states are a set of pre-determined patterns, such as Map, Parallel, and Next. To run a chain application such as the IoT controller app in Section 2.3, the developer would write two states and the transition from the first state to the second using the `Next` construct. The `Next` field instructs the Step Functions to run the second state only after the first state completes and to pass the first state's output to the second state as input. At runtime, Step Functions invokes the first `preprocess` function, waits for `preprocess` to complete, receives its output, invokes the second `adjust` function with `preprocess`'s output, and finally receives `adjust`'s output as the final result of the application.

For fan-out, Step Functions provides two patterns—Map and Parallel. The Map state expects an array as input and, for each element in the array, concurrently invokes an instance of the branch, which is one or a chain of many functions. Parallel treats the input as one entity and invokes each

branch with its input. Both Map and Parallel output an array that contains the branches' results in order. At runtime, Step Functions invokes each fan-out branch and waits for all of them to complete before returning their results. Once all branches return their results, Step Functions groups them into an ordered array and returns it as the output of the state.

Note that the standalone orchestrator decides which patterns applications can program with and how the patterns are implemented. For instance, Step Functions provides Map and Parallel as the only two interfaces for fan-out patterns. Moreover, the Step Functions imposes a concurrency limit on Map when the array size is greater than 40 [34]. Applications that use Step Functions need to consider those limitations.

Chapter 3

Application-Level Orchestration: Design Goals, Benefits and Challenges

In this section, we discuss the drawbacks of relying on standalone orchestrators to build and run complex serverless applications, motivate an alternative architecture we call application-level orchestration, and describe key design principles that application-level orchestration should follow.

3.1 Drawbacks of Standalone Orchestrators

As explained in §2.4, standalone orchestrators work as centralized controllers that drive the execution of applications. Such centralization makes supporting complex interactions simple—e.g. an orchestrator can support fan-in patterns by simply waiting for all branches to complete before invoking an aggregation function. Similarly, a centralized orchestrator can ensure that workflow results appear to be the result of executing each constituent function exactly once by choosing one result for each function invocation.

However, standalone orchestrators have important drawbacks for both serverless providers and serverless users. They are a potential performance bottleneck, they are expensive to host and use, and they preclude users from making application-specific optimizations.

First, building performant, scalable, and fault-tolerant multi-tenant services is hard. As an additional such service that is critical to application performance and correctness, orchestrators introduce

yet-another potential performance and scalability bottleneck. Indeed, we find that, in practice, production orchestrators limit end-to-end performance for highly-parallel applications. For example, AWS Step Functions limits the number of concurrent branches for its Map pattern [34]. When the number of concurrent branches exceeds 40, Step Functions may cease starting new executions until previous branches are complete. For applications that need higher levels of parallelism (such as a video encoder that tries to process more than 40 video chunks in parallel), Step Functions restricts their scale and performance. Importantly, applications have no way to circumvent these limitations and thus have to wait for the providers to improve the service's performance.

Furthermore, hosting orchestrator services are expensive as they require dedicated resources that are often replicated as well for performance and fault-tolerance. 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, losing 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. AWS Step Functions charges users based on the number of state transitions at around $27.4 per 1 million transitions. In comparison, AWS Lambda charges $0.2 per 1 million functions invoked.

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 to operating systems [12,16], networks [20,44], and storage systems [24, 32].

For example, applications that need orchestration patterns not supported by the provider-hosted orchestrators have to either compromise performance by using less-efficient patterns or first repeat the hard work of building, deploying, and managing their custom orchestrators. Similarly, an application with deterministic functions, which requires weaker execution guarantees, may not be able to reap performance benefits if the orchestrator service only supports strong exactly-once execution. Indeed, we find that provider-hosted orchestrators force applications to compromise performance by using less-efficient patterns (§ 4.8). For example, in our implementation of ExCamera with AWS Step Functions, we found that 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.

In summary, relying on an additional standalone service to orchestrate complex serverless ap-

plications costs both cloud providers and cloud users, risks performance bottlenecks, and prevents applications from adding custom patterns or optimizing implementations for application-specific needs. It benefits both cloud providers and users if an alternative solution can allow applications to design and implement their own orchestration logic without adding an additional service that providers or users have to provision and manage.

3.2 Design Goals and Challenges

This thesis proposes application-level orchestration, namely moving orchestration into a library that runs in-situ with user-defined FaaS functions. Our vision is to design a system that can fully support at least the same set of applications achievable with standalone orchestrators, with equally strong execution guarantees, while removing orchestrator services from the architecture. Specifically, orchestration-level orchestration should satisfy the following design requirements to be an adequate alternative to standalone orchestrators.

First, application-level orchestration should not require adding additional services to existing serverless infrastructures. Every additional service added introduces the same drawbacks as standalone orchestrators. Namely, the additional service risks being a performance bottleneck, requires dedicated hardware and engineering personnel resources, and precludes users from implementing application-specific optimizations.

Without the option to introduce additional services, application-level orchestration must be able to support complex serverless applications using only the existing systems on unmodified serverless infrastructures. Specifically, the library must build the ability to coordinate stateful inter-function interactions and ensure strong execution correctness upon FaaS functions and serverless data stores.

Second, application-level orchestration design should work with the existing serverless APIs without requiring modified or specialized interfaces. While implementation details differ, all serverless platforms support the same abstraction described in Chapter 2.2. The orchestration library should build on top of this universally-supported abstraction without assuming or requiring specialized APIs that may only exist on certain platforms. While some platform-specific APIs might improve the functionality or performance of an application-level orchestration library, being able to run on any infrastructure that implements the basic serverless abstraction makes the library, and thus any applications that use the library for orchestration, platform-independent and portable. If a platform-specific API can improve the performance of the library, it should be used in the implementation. But the system design should not rely on such APIs to be functional.

Third, application-level orchestration should support high-level programming interfaces. An important contribution of standalone orchestrators is the introduction of high-level programming interfaces that allow developers to easily express inter-function interactions(Chapter 2.4). Application-level orchestration should preserve this advantage.

Fourth, application-level orchestration should support strong, exactly-once execution guarantees. Exactly-once execution is important for serverless workflows because FaaS functions can be non-deterministic and execute multiple times. For example, 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 particularly 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. Therefore, application-level orchestration must be able to ensure exactly-once execution guarantee.

Last but not least, application-level orchestration should be able to support other key functionalities such as access control and rate limiting. Standalone orchestrators often integrate with other cloud services to support access control. For instance, AWS IAM controls permissions to access Step Functions workflows. Individual function permissions within workflows are also controlled via IAM. Such access control is important for protecting applications from malicious access as well as for users to control the request rates and thus monetary costs of running the application. Application-level orchestration should still support these functionalities.

Achieving application-level orchestration under the goals and constraints above presents several challenges. The key challenge comes from the elimination of the standalone orchestrator service and, as a result, the need to decentralize orchestration logic.

First, without a standalone service that functions as a centralized controller, it is difficult to coordinate stateful interactions such as fan-out and fan-in. Recall that 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. Thus, to perform fan-out and fan-in, an orchestrator can simply invoke all branches, wait for them to complete and then pass the outputs of all branches to the fan-in function.

However, supporting such patterns is challenging for application-level orchestration without a standalone service, because FaaS functions are short-lived and cannot communicate directly. Branches cannot wait for all other branches to complete because it risks timeout in the presence

of stragglers and they lack a way to communicate the completion with each other. Invoking the fan-in function is also challenging due to the lack of communication. Each function only has access to its own states, including outputs. Thus, if we designate one of the branches to invoke the fan-in function, that branch cannot access the outputs of branches. Having branches "send" their outputs to the fan-in function is also not possible as functions can no longer receive inputs after it is invoked.

Second, without a standalone service, it is difficult to ensure exactly-once execution. FaaS engines only guarantee at-least-once execution for functions. Orchestrators can ensure that workflows appear to execute exactly once by choosing one result for each function invocation because all invocations return their outputs to the orchestrator. Moreover, 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. Application-level orchestration cannot rely on a centralized process to deduplicate extra executions.

Third, without a standalone service, it is difficult to support high-level application definitions. Standalone orchestrators support high-level application definitions by interpreting the definitions and centrally driving the execution of applications. There will not be such a process in application-level orchestration. Using a FaaS function to execute the high-level definition creates the same drawbacks as driver functions.

3.3 Towards Application-level Orchestration

In this thesis, we demonstrate that application-level orchestration is achievable and feasible despite the difficulties in Chapter 3.2. Our design of application-level orchestration relies on two key insights. First, serverless data stores can solve the challenge of coordinating stateful interactions as well as ensuring strong execution guarantees. Second, high-level application definition can be broken up at compile-time and executed in a decentralized manner in-situ with user code.

For stateful patterns such as fan-in and fan-out, instead of having all functions return outputs to a centralized orchestrator, an application-level orchestration library writes function outputs to a shared data store. Storing such intermediate data in a shared data store not only allows each branch to complete without waiting for others, but it also empowers any one of the branches to invoke the fan-in function because all branches can access other branches' outputs once they are written into the shared data store. Moreover, it solves the difficulty where the fan-in function cannot receive inputs from multiple upstream functions. The invoker of the fan-in function simply passes in the pointers to all branches' outputs, and the orchestration library in the fan-in function can read the data from

the shared data store before passing it to the user code. The success of such an algorithm naturally depends on a unique naming scheme such that different executions' outputs are distinguished. We discuss the details of one such naming scheme in the next Chapter.

Serverless data stores can also help ensure exactly-once semantics in application-level orchestration. Using checkpoints, each function execution can record its output to the shared data store. The existence of a checkpoint represents the successful completion of a prior execution. Thus, in the face of multiple executions, checking the existence of a checkpoint before writing ensures only one execution's output is the "committed" result of that step. Many data stores support such an atomic "check-a-condition-and-then-write" operation natively, or via transactions. Therefore, the library can ensure exactly one execution's result is taken as the final result. All other executions simply discard their outputs. The correctness of the checkpointing algorithm requires a unique naming scheme as well as careful consideration of retry executions to ensure the orchestration library code is idempotent. We discuss the details of one such algorithm in the next Chapter.

Last but not least, most high-level programming languages express applications as directed graphs. Nodes in the directed graph represent FaaS functions and edges transitions between functions. For each transition, the tail node's output is passed to the head as input. We can decentralize such a directed graph where each tail node executes its outgoing edges by invoking the head nodes and passing its output. A head node with multiple tails constitutes a fan-in and tails would coordinate via the data store so that the head node is invoked only once. A high-level application definition can be decentralized at compile-time by assigning the edges to the tail node. We present one possible encoding of edges in the next Chapter.

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.

With orchestration in a library, applications have full control over what patterns to implement and what functionality to include. Applications that need orchestration patterns not supported by the provider-hosted orchestrator can add that pattern into the library without having to build their own custom orchestrators. Similarly, an application with deterministic functions, which requires weaker execution guarantees, can simply turn off the part of the library that ensures exactly-once guarantee to save on latency and resources.

Moreover, as cloud providers improve the performance and efficiency of essential serverless com-

ponents, application-level orchestration built on top of those services benefits automatically. For example, when a serverless data store improves its write throughput, an orchestration library that uses the data store for coordination also becomes faster without any changes to the library's code. Additionally, when a FaaS engine becomes more efficient and cheaper to use, application-level orchestration automatically reduces its costs.

Chapter 4

Application-Level Orchestration in Unum

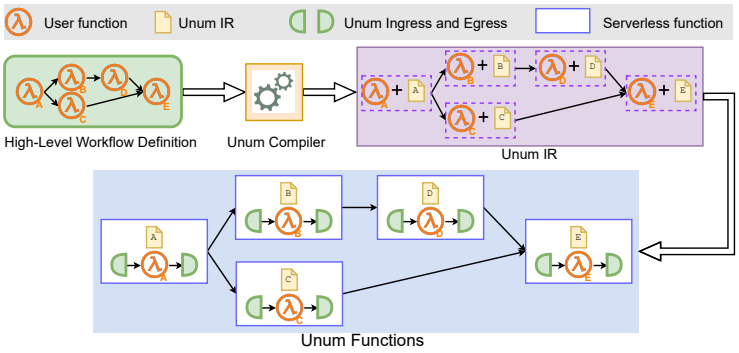
Unum is an application-level orchestration system that supports complex serverless applications without a standalone orchestrator. It does so by decentralizing orchestration logic in a library that runs in-situ with user-defined FaaS functions and leverages a scalable consistent data store for coordination 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 (§4.3) of standalone orchestrators.

4.1 Architecture

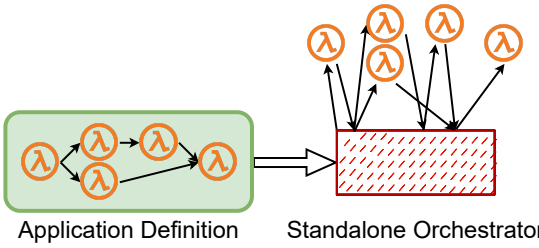
Figure 4.1a depicts how developers run serverless workflows using Unum. Developers write individual functions and describe the workflow using a high-level workflow language, such as Step Functions' expression language. An Unum front-end compiler uses these to extract portable Unum IR for each node in 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,

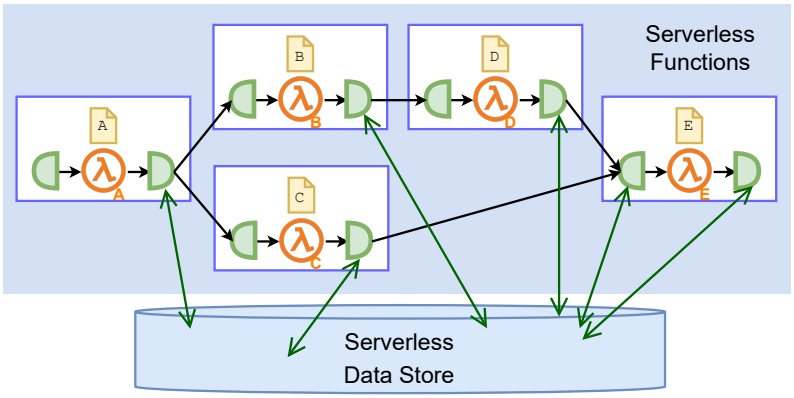
[1] Since functions are typically written in dynamic languages, the Unum library source code is placed alongside the function and dynamically imported, rather than statically linking an object file



(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 4.1: 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.

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.

`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 4.1: Unum intermediate representation instructions.

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 the Unum execution state, respectively (Listing 1). 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 4.1c).

4.2 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 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 4.1).

```
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,
}
```

Listing 1: An Unum request wraps the outputs of parent nodes with execution state that allows function invocations to be named uniquely and assists in coordinating fan-ins. Unum IR instructions can reference this metadata and modify it for subsequent functions.

Instructions direct the runtime to invoke functions and operate on state metadata passed between functions in Unum-wrapped function inputs (Listing 1).

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 output of the current function.
- **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 §4.4.

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 (§4.6) and to coordinate fan-in (§4.4).

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 (§4.4) or garbage collection (§4.5).

This IR is sufficient to represent basic patterns, as well as more complex fan-in patterns (described in §4.4).

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

Map. Applications may also 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 resulting images. Unum's `Map` invocation passes each element from the function's output to a different invocation of the same function.

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 (Listing 1).

4.3 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 pro-

vide 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 (and it is, of course, possible for the checkpoint to be added after a concurrent execution checks for its existence in the ingress component) but helps reduce computation that we know will go unused.

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 dissertation. However, Unum does not preclude applications from using libraries, such as Beldi [45], that can solve this problem.

```
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)
```

Listing 2: 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.

4.3.1 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 [10, 17, 36, 39] or failure redirection that triggers a pre-configured error-handler function [33, 37]. 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.

4.4 Fan-in Patterns

In fan-in patterns, the results of multiple nodes are used to invoke a single target 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 dependent functions as a coordinator for the fan-in 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 (§4.3).

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.

4.5 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.

As an application-level library, Unum's design allows applications to customize their GC strategy. Simple policies such as deleting intermediate data older than a set limit are easily implementable via data stores or as a cron job. However, in this section, we demonstrate the algorithm in Unum's standard library that deletes checkpoints and coordination sets *as soon as* they are no longer needed for execution correctness, to show that Unum's decentralized design allows the most aggressive strategy.

4.5.1 Checkpoint Collection

A checkpointing node does not know when *its* checkpoint is no longer necessary. If it deletes its checkpoint after invoking subsequent functions but before completing, 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*

```
def egress_fan_out(self, result):
    result = _checkpoint(self, result)
    _do_next(result)

    completion_set_after =
        write_set_read_result(fan_out_gc_set, fan_out_index)
    if all_complete(completion_set_after):
        database_delete(prev_checkpoint)
        database_delete(fan_out_gc_set)
```

Listing 3: Pseudo-code showing Unum's GC mechanism.

use the checkpoint 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 might execute duplicates of a function invocation an arbitrary time in the future.

In particular, 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.

Unum runtime implementations delay garbage collection until after invoking next functions, sacrificing some storage overhead in favor of minimizing end-to-end latency for a workflow.

Fan-out cases 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.

4.5.2 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.

4.6 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.

4.7 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 IR instructions 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, and per-write DynamoDB costs to create checkpoints).

4.7.1 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 [33] 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 [18]) 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 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.

4.7.2 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 invokes a function by publishing the input data as an event to the function's queue.

GCF supports automatic retry for asynchronous functions [39]. 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.

4.8 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.

4.8.1 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 [22, 23]. DynamoDB uses the on-demand provisioning option that charges per-read and per-write [15]. 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 [42], which provides similarly strong execution guarantees as Unum [19]. 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 (§ 4.8.4).

4.8.2 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 consisting of `noop` functions as well as the end-to-end performance of real applications. Overall we find that Unum performs comparably or significantly better than Step

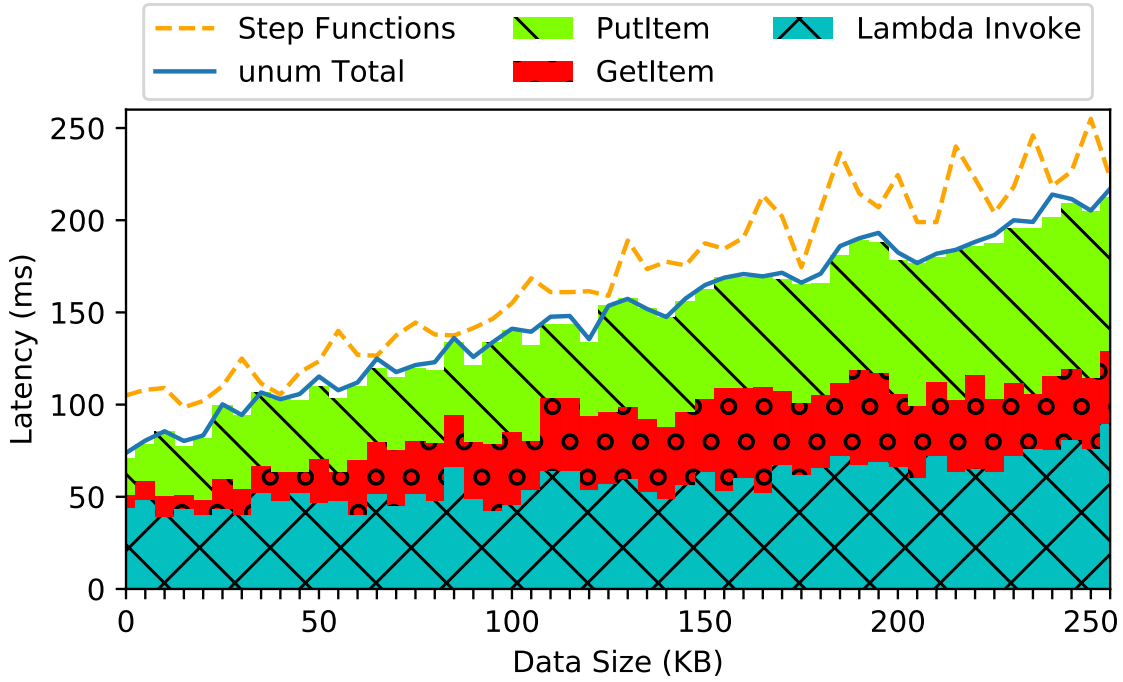


Figure 4.2: An orchestrator incurs a latency on each transition between functions. Unum's overhead is due to storage operations to ensure exactly-one-result semantics, Lambda invocation API overhead to enqueue the next function to run, and additional Unum runtime code in the function instance itself for the orchestration logic.

Functions in most cases owing to higher parallelism and a more expressive orchestration language, with modest slowdowns in the remaining cases due to implementation deficiencies.

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 4.2 shows the 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 4.2).

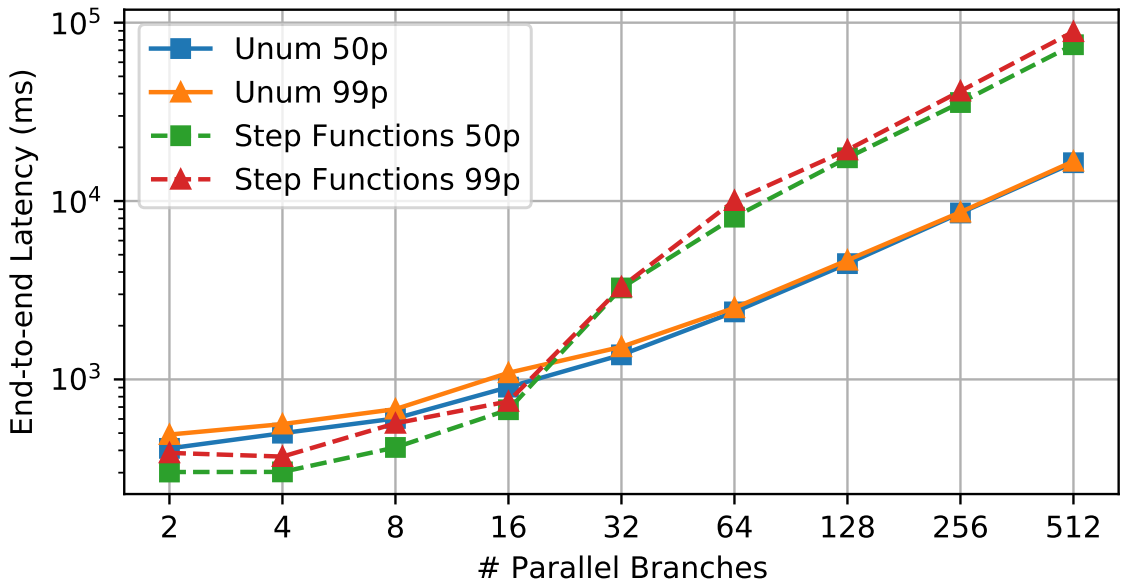


Figure 4.3: End-to-end latency of a fan-out and fan-in pattern with increasing branching degrees. Unum is slower at lower branching degrees but significantly outperform Step Functions at moderate and high branching degrees.

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 4.3 shows the latency of a fan-out followed by a fan-in at varying branching degrees for both Unum and Step Functions. At a 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 [34] while Unum is limited only by Lambda's scalability, resulting in over 4x lower latency with 512 branches.

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 4.2).

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

	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 4.2: 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.

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.

4.8.3 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 [8], 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 are charged per millisecond such as Lambda runtime cost [5], and storage for persisting states is charged per read and write such as DynamoDB reads and writes [15].

On AWS, Unum is much cheaper than Step Functions—AWS's native orchestrator. For a basic transition in a chaining 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 a checkpoint, (3) $0.279 for 1 DynamoDB read to check a checkpoint's existence, and (4) $2.79 for 1 DynamoDB write to garbage collect a 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 4.2 shows the cost to run each of the applications we implemented based on public pricing information for AWS in the `us-west-1` region using Unum and Step Functions. Unum is consistently,

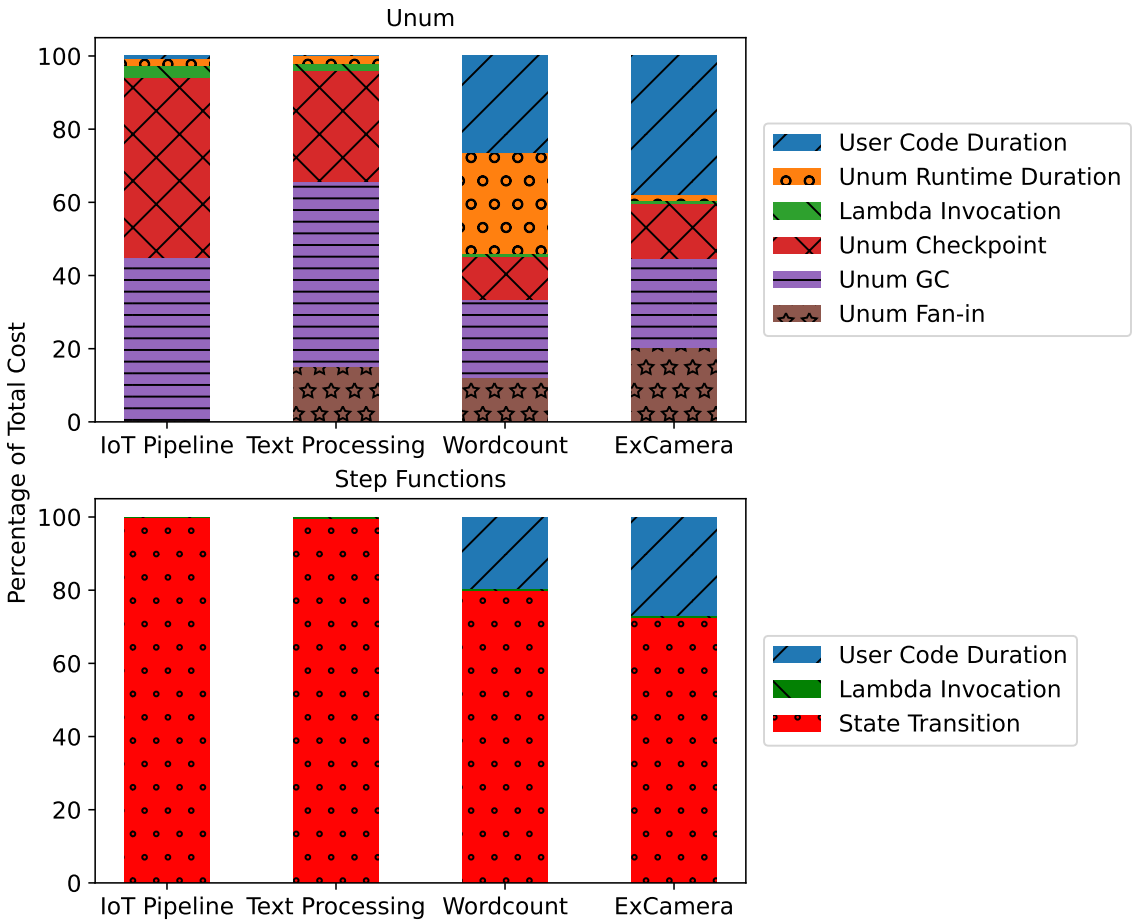


Figure 4.4: Step Functions state transitions dominate the total costs for all applications (99.5% in IoT Pipeline, 99.4% in Text Processing, 80.0% in Wordcount, 72.2% in ExCamera). While Unum runtime cost is also the majority, it accounts for a smaller portion of the overall costs (95.7% in IoT Pipeline, 97.8% in Text Processing, 72.5% in Wordcount and 61.0% in ExCamera).

and up to 9x, cheaper than Step Functions for the applications we tested.

Figure 4.4 shows the cost to run each application using Unum broken down into 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 are the majority[2]. This includes writing checkpoints, writing to coordination sets (either for fan-in for garbage collection), and deleting checkpoints for garbage collection.

Of course, developer-facing pricing is only a proxy for the *actual* costs of hardware and human resources. However, it is clear that, in practice, Unum's costs are reasonable and, in fact, often lower

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

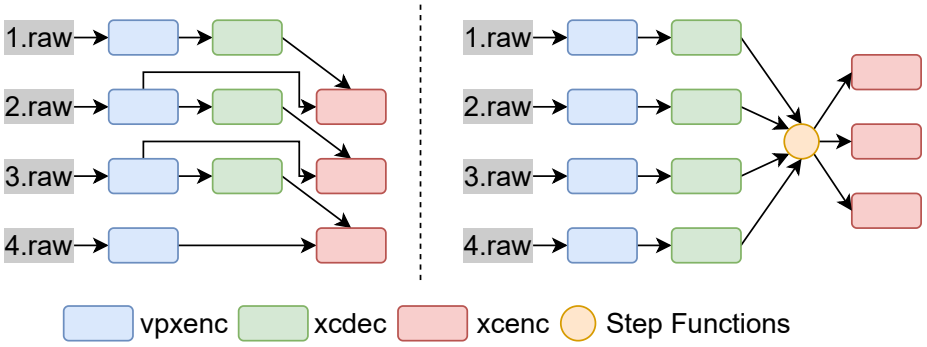


Figure 4.5: Unum ExCamera replicates the application logic from gg and mu where the re-encode stage (xcenc) of a branch can start immediately when the previous branch completes decoding (xcdec) and my own branch completes the initial encoding (vpxenc). Step Functions provides a Map pattern [34] for parallel 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 4.3: ExCamera performance. Unum is 7.1% faster than gg [22] and 10.5% slower than the hand-optimized implementation.

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 economy of scale. 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).

4.8.4 Case Study: ExCamera

ExCamera [23] 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 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 4.5).

Performance

Using the same experimental setup as the prior work (i.e., encoding the first 888 chunks of the `sintel-4k` [35] video using 16 chunks per batch and Lambdas configured with 3GB of memory), Unum is 7.1% faster than gg [22] and 10.5% slower than the original, hand-optimized ExCamera (Table 4.3). The original authors attribute 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.

Compared 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.

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 [23], while gg's only has a coordinator server [22]. 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.

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.

Chapter 5

Discussion

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 (orchestration-as-code) and allows the developer to run arbitrary logic to determine what the next workflow step should be at runtime. This is not currently possible with Unum. It is worth exploring whether orchestration-as-code interfaces can be translated into the Unum IR, and if not, whether the Unum IR is extensible to support orchestration-as-code applications.

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.

Support for more platforms. While Unum is designed to run on any serverless platform that meets our minimal criteria, our current implementation is only complete for AWS Lambda using DynamoDB and Google Cloud Functions using Firestore. It would be interesting to broaden Unum's implementation onto more platforms and a wider variety of data stores.

Chapter 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 [6] defines serverless workflows as state machines using a JSON schema. Google Workflows [27] uses a YAML-based interface to list steps in a workflow sequentially and allows jumps among steps. Azure Durable Functions [9] 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 [6] and Google Workflows [27], it is provided by a cloud service that is separately hosted and billed. In Azure Durable Functions [9], 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 [46] 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 [41] 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 [23] is parallel video encoder running on Lambda. It is built with 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 [22] proposes a thunk abstraction where each thunk executes as a lambda that has no nondeterministic or non-idempotent behavior, and a way to program data dependencies between

thunks. 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, but requires manual restart. 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 [45] and Boki [29] 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 [6], Google Workflows [27]). Others allow expressing workflow as code (e.g., Durable Functions [9], Kappa [46], Fn Flow [21]).

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.

Chapter 7

Conclusion

7.1 Summary of Contributions

This dissertation demonstrates the feasibility of application-level orchestration for complex serverless applications. Moreover, it contributes techniques to coordinate stateful inter-function patterns and ensure exactly-once execution guarantees in a decentralized manner without a logically centralized controller service. By leveraging existing FaaS schedulers and strongly consistent data stores, we showed the design and implementation of an application-level orchestration library that provides the same functionalities as standalone orchestrators, affords applications more flexibility, reduces cost , and achieves competitive performance.

More specifically, this dissertation:

- Motivates the need for application-level orchestration, shows the drawbacks of standalone orchestrators, and lays out challenges and design goals for achieving orchestration in the application-level.
- Designs and implements the Unum intermediate representation that expresses directed graphs of serverless functions in a decentralized manner where each node encodes only local information of its inter-function interactions, and shows how this intermediate representation can be automatically derived from existing higher-level programming languages of standalone orchestrators.
- Designs and implements Unum, an application-level orchestration system that supports complex stateful inter-function patterns and exactly-once execution guarantees using only existing

FaaS schedulers and strongly consistent data stores without requiring changes to the serverless infrastructures.

7.2 Open Questions and Future Work

Support for alternative programming interface An important contribution of Unum is supporting existing higher-level programming interfaces such as AWS Step Functions. However, an important assumption that Unum makes is that the interface expresses a directed graph where nodes are FaaS functions and edges are transitions between functions. Alternative programming interfaces such as orchestration-as-code used in Azure Durable Functions do not directly express serverless applications as directed graphs. Instead, orchestrators are similar to general-purpose processes that can execute any program—similar to driver functions 2.3. It is worth exploring whether orchestration-as-code interfaces can be translated into the Unum IR, and if not, whether the Unum IR is extensible to support orchestration-as-code applications.

Evaluate and support alternative data stores The current Unum design relies on strongly consistent data stores and the implementation uses NoSQL databases. While they fulfill the functional needs of Unum, alternative data stores might yield better performance or costs. Therefore, it is worth exploring other data store options, such as key-value stores and object stores, to understand the performance and cost characteristics of Unum's intermediate metadata.

7.3 Concluding Remarks

Serverless platforms allow developers to construct applications from modular programming units that can scale quickly and independently, promising burst-scalability and fine-grained billing. Workflow orchestrators make building complex applications out of these event-driven, asynchronous functions reasonable, but are inflexible and may introduce performance, cost, scalability, and deployment overhead to developers and platform providers. We designed and implemented Unum, a *decentralized* workflow system that requires no additional infrastructure to deploy, imposes no additional limits on scalability, and performs as well as or better than centralized solutions while providing similar expressiveness and execution guarantees.

This thesis demonstrates that application-level orchestration is possible and practical using just the basic APIs of existing serverless systems. Moreover, application-level orchestration benefits

both cloud providers and cloud users. It empowers cloud users with access to implementations of orchestration operations and the flexibility of employing application-specific optimizations. It frees cloud providers from the work of building, scaling, and maintaining yet another complex performance-critical service. Furthermore, the performance and efficiency of the application-level orchestration library improve as the underlying systems develop. Thus, cloud providers can direct freed-up resources to fundamental building blocks of their serverless infrastructure and automatically reaps the benefits of a better "orchestrator".

We hope the conclusions of this thesis inspire cloud practitioners to reconsider the approach of supporting new functionalities by simply adding more services and broadening the "kernel APIs" of their infrastructure. Moreover, we hope our results would encourage the serverless community to build and explore additional frameworks that streamline the process of developing and running large-scale serverless applications.

Bibliography

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI 2020*, 2020.

[2] Amazon Aurora Serverless. `https://aws.amazon.com/rds/aurora/serverless/`.

[3] Amazon DynamoDB On-Demand – No Capacity Planning and Pay-Per-Request Pricing. `https://aws.amazon.com/blogs/aws/amazon-dynamodb-on-demand-no-capacity-planning-and-pay-per-request-pricing/`.

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

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

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

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

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

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

[10] 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`.

[11] 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`.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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`.

[20] 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.

[21] Fn Flow. `https://fnproject.io/`.

[22] 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.

[23] 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.

[24] 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.

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

[26] Google Cloud Firestore. `https://cloud.google.com/firestore`.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

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

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

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

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

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

[40] 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.

[41] 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.

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

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

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

[45] 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.

[46] 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.

Appendix A

Unum Intermediate Representation

A.1 Overview

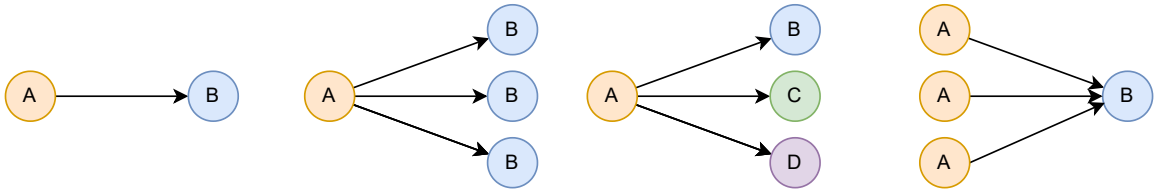
The Unum intermediate representation (IR) expresses serverless applications that consist of many FaaS functions. Applications are modeled as directed graphs where nodes are FaaS functions and edges are transitions between functions. The Unum IR expresses such a directed graph by encoding each node with its outgoing edges in a separate file. For example, an application that comprises of 5 functions would have 5 files in its Unum IR, one for each function that encodes the function's node in the direct graph as well as the node's outgoing edges.

Application developers can write the Unum IR directly for each function to build applications. Alternatively, they can provide an AWS Step Functions definition to the Unum frontend compiler, and the compiler can translate the state machine into a set of Unum IR files, one for each function in the application.

During execution, the Unum runtime library reads the Unum IR and performs orchestration operations based on the IR. Orchestration operations include invoking the next unum function with the current function's result, checkpointing the current function's output, signaling a branches completion by updating the Completion Set data structure, etc. For more details, see the chapter on Unum runtime (Appendix B).

A.1.1 A Note on Implementation and Customization

An important point of Unum is to demonstrate what you can achieve on the application-level for orchestrating serverless applications. The general idea of the Unum IR is to use a platform-agnostic



(a). One-to-one (b). One-to-many (c). Many-to-one

representation that encodes serverless applications as directed graphs. Moreover, the representation encodes in a decentralized manner where each node is encoded along with its outgoing edges. Such a design not only allows Unum to support existing higher-level programming interfaces such as AWS Step Functions, but also enables orchestration to execute with a logically-centralized controller.

Therefore, it is worth noting that the implementation of Unum IR described in this document is one possible implementation of the general idea. This implementation may not be the best or richest realization of the idea, but the point is that developers, without the help or control of cloud providers, can create other implementations of this idea to build and run complex serverless applications.

The goal of this document is to demonstrate this implmetation and show how one would create an IR that is capable of support a superset of applications possible with AWS Step Functions. We hope this work will inspire developers to create other custom-made IRs that are optimized for their applications' use cases.

A.1.2 Transitions

Transitions between nodes in the Unum IR can be one-to-one, one-to-many or many-to-one. A one-to-one transition chains two functions together where the head node function is invoked when the tail node function's result becomes available. The input to the head node function is the output of the tail node function.

A one-to-many transition represents a fan-out where the output of the tail node function is "broadcasted" to many "branches". The head node function of each branch is invoked when the tail node function's result becomes available.

There are two flavors of one-to-many transitions in Unum. The first flavor is similar to AWS Step Functions' Parallel state [38] where the input to every branch's head node function is the output of the tail node function. In other words, every branch is invoked with the same input. This flavor of one-to-many transition is useful when you need to process the same data in different ways, where each branch has a different head node function and all branches can run in parallel. The other

flavor of one-to-many transition is similar to AWS Step Functions' Map state [34] where the tail node function outputs an iterable (e.g., an array) and the input to each branch's head node function is one element of the iterable, in order. Thus, every branch is invoked with different input data. Moreover, each branch has the same head node function. In other words, you're applying the same computation on different data in parallel. This flavor of one-to-many transition is useful when you need to break up a large dataset into chunks and process each chunk in parallel.

A many-to-one transition represents a fan-in where a single head node function is invoked with the outputs of multiple tail node functions. In Unum, the tail nodes' outputs are grouped into an ordered array and the head node function is invoked with this array as its input. An important feature of many-to-one transitions is that the head node function is invoked only when all tail node functions' outputs become available.

A.2 The IR Language

In practice, each function itn an Unum application has an IR file that encodes the function's node in the directed graph as well as the node's outgoing edges. The Unum IR language uses YAML and has the following fields to encode nodes and edges,

```
Name: this function's name
Next:
    Name: next/head function name
    Type: Scalar | Map | Fan-in
    Values: an array of invocation names (When Type: Fan-in)
    Conditional: boolean expression (Optional. Default True)
    Payload Modifiers: an array of modifier instructions (Optional. Default None)
Start: boolean (Optional. Default False)
Checkpoint: boolean (Optional. Default True)
```

By default, the Unum runtime expect this file to be named `unum_config.yaml` and each function of an Unum application should have its own `unum_config.yaml` that is package together with user-defined FaaS function code and the Unum runtime library. For more details, see the chapter on Unum runtime (Appendix B).

A.2.1 Name

The `Name` field specifies the function's name which can be any valid ASCII strings. Each function must have a name that's unique within its application. The application's name is specified in the Unum template and not in each function's `unum_config.yaml`. When deploying applications, Unum by default names the deployed FaaS function `<application name>-<function name>`. That is if you deploy your application on AWS, your Lambda functions will have names of `<application name>-<function name>`.

A.2.2 Start

`Start` is set to True for entry functions of applications. Users invoke an application by invoking its entry function. At runtime, the Unum library checks if `Start` is true, and if yes, adds a `Session` field to the runtime payload that uniquely identifies each application invocation. See the chapter on Unum runtime (Appendix B).

A.2.3 Checkpoint

`Checkpoint` is a boolean field that controls whether a node's output is checkpointed into the data store. Checkpoints directly affect the execution guarantees of applications. When `Checkpoint` is set to false, application are executed at-least once; whereas when `Checkpoint` is true, applications are executed exactly-once.

A.2.4 Next

The `Next` field specifies the outgoing edges. If there is only one outgoing edge (i.e., a chain or a one-to-one transition), the `Next` field contains only a single object. For example,

```
Name: A
Next:
  Name: B
  Type: Scalar
Start: True
Checkpoint: True
```

If there are multiple outgoing edges (i.e., a fan-out or one-to-many transition), the `Next` field specifies an array of objects. For example,

```
Name: A
Next:
  - Name: B
    Type: Scalar
  - Name: C
    Type: Scalar
Start: True
Checkpoint: True
```

The object that encodes an outgoing edge has up to five fields:

- `Name`: the function name of the head node function of this outgoing edge
- `Type`: specifies the type of this transition. The standard IR supports 3 different values for `Type`:
  - `Scalar`: The output of the tail node is treated as a single scalar entity when passed as input to the tail node function of this edge.
  - `Map`: The output of the tail node is treated as a iterable, and each element of the output is passed to one invocation of the tail node function as input. That is the tail node function is invoked x number of times where x equals the size of the iterable output of the tail node.
  - `Fan-in`: The output of the tail node is grouped together with the outputs from other functions and passed as input to the tail node function of this edge in the form of an ordered array. All values needed to invoke the head node function is specified in the additional `Values` field which is only used when `Type: Fan-in`.
- `Values`: Only used when `Type` is `Fan-in`. `Values` lists, *in order*, the invocation names of all tail node functions whose outputs are needed to invoked the head node.
- `Conditional`: A boolean expression that controls whether or not this edge is taken at runtime. The boolean expression can contain runtime variables such as the invocation name. An edge

is executed, i.e., the head node function is invoked, only when its `Conditional` evaluates to true. By default, such as when `Conditional` is not even specified, `Conditional` is set to true.

- `Payload Modifiers`: A list of modifier instructions that can change the value of runtime variables and states of the execution. See below for more details.

The following examples illustrate how Unum uses the above object to encode and support a variety of transitions.

Chaining

To chain a B function to an A function, A's IR would look like,

```
Name: A
Next:
  Name: B
  Type: Scalar
Start: True
```

Branching

To branch on A's result and invoke B if the result is above some threshold (e.g., 50) or otherwise invoke C, A's IR would look like,

```
Name: A
Next:
  - Name: B
    Type: Scalar
    Conditional: "$out > 50"
  - Name: C
    Type: Scalar
    Conditional: "$out <= 50"
Start: True
```

`Conditional` is the field that specifies the branching logic. `$out` is a Unum runtime variable that refers to the output of the function.

Map

If A outputs a list and wants to further process each element of the list with B, A's IR would look like,

```
Name: A
Next:
  Name: B
  Type: Map
Start: True
```

The number of B invocations would equal to the size of A's output list. Moreover, each B invocation would be assigned a unique index at runtime to distinguish the B function invocations that are on different branches. For instance, the B invocation that processes the 1st element in A's output list would be assigned a branch index of 0. In general, it's important that every function invocation can be uniquely identified, and assigning branches unique indexes is one mechanism that Unum employs to guarantee unique naming. For more details, see the chapter on Unum runtime (Appendix B).

Fan-out

To fan-out A's output as a single scalar entity to multiple head node functions, A's IR would look like,

```
Name: A
Next:
  - Name: B
    Type: Scalar
  - Name: C
    Type: Scalar
Start: True
Checkpoint: True
```

Similar to the Map case, each branch of a fan-out is assigned a unique branch index at runtime based on the order a branch appears in the `Next` field. For instance, function B in this case would

be assigned index 0 and C index 1. Even though branches in fan-out have head node functions of different names, assigning branch index help distinguish invocations at runtime when fan-outs are nested. For more details, see the chapter on Unum runtime (Appendix B).

Fan-in

To fan-in the outputs of multiple tail nodes into a single head node, all tail nodes need to use the `Fan-in` type when specifying their outgoing edges. For example, if A and B fan-in to C, A and B would have IR that looks like,

```
Name: A
Next:
  Name: C
  Type: Fan-in
  Values: [A, B]
```

```
Name: B
Next:
  Name: C
  Type: Fan-in
  Values: [A, B]
```

Both A and B would specify C as the next node and the `Values` field lists the names of *invocations* whose outputs are required before the fan-in head node—C in this case—can be invoked. Note that the `Values` field must list the names in the same order in all tail node functions' IR, because the tail nodes' outputs are passed to the head node function in the order specified in the `Values` field.

Most likely, A and B are first created as branches of a fan-out. For instance, an S function first fan-out to A and B,

```
Name: S
Next:
  - Name: A
    Type: Scalar
```

```
  - Name: B
    Type: Scalar
Start: True
Checkpoint: True
```

In this case, the A and B invocations will have branch indexes 0 and 1. For A and B to fan-in to C, the names in `Values` should include the branch index. For instance, the default runtime library expects branch indexes to appear after a `-UnumIndex-` string,

```
Name: A
Next:
  Name: C
  Type: Fan-in
  Values: ["A-UnumIndex-0", "B-UnumIndex-1"]
```

```
Name: B
Next:
  Name: C
  Type: Fan-in
  Values: ["A-UnumIndex-0", "B-UnumIndex-1"]
```

The index is known a priori at compile time because the directed graph that the IR expresses is static.

For dynamic patterns such as Map, Unum supports wildcard characters and globbing in the IR which expands at runtime. For instance, if S maps to many branches of A and the A's fan-in to B, A's IR would look like,

```
Name: A
Next:
  Name: B
  Type: Fan-in
  Values: ["A-UnumIndex-*"]
```

If S' output list has size 3, there will be 3 A invocations and `[A-UnumIndex-*]` would expand to `[A-UnumIndex-0, A-UnumIndex-1, A-UnumIndex-2]` at runtime. The globbing process uses Unum's payload metadata at runtime. Specifically, in this example, the runtime library would perform globbing using the `Fan-out` field in the input payload to the A invocations. Each A invocation would receive an input that looks like,

```
{
    "Session": "8cef2097-f6fa-4fa2-bc70-7aa7bbbc23d9",
    "Fan-out": {
        "Size": 3,
        "Index": 0
    }
}
```

The `Fan-out` field is added by S when invoking the A functions. The runtime on A would expand `A-UnumIndex-*` using the `Size` field knowing that there are in total 3 A invocations. See the chapter on Unum runtime (Appendix B) for details on how the runtime interprets and executes the IR.

Additionally, Unum supports payload modifiers which are instructions that modifies the payload metadata. They enable further manipulating the payload metadata. For instance, to remove a fan-out field from the payload, the standard library supports a `Pop` instruction. To use the `Pop` instruction, A's IR would look like,

```
Name: A
Next:
  Name: B
  Type: Fan-in
  Values: ["A-UnumIndex-*"]
  Payload Modifiers: ["Pop"]
```

The `Pop` instruction would remove the `Fan-out` field from the input payload when A's fan-in to B such that B's input would look something like the following without a `Fan-out` field,

```
{
```

```
    "Session": "8cef2097-f6fa-4fa2-bc70-7aa7bbbc23d9"
}
```

`Pop` is useful when you have nested fan-outs and maps.

A.2.5 Unum Runtime Variables

Many complex interactions require additional control over the runtime metadata. To provide the necessary programmability, Unum IR supports a set of runtime variables. Through the runtime variables, developers can read and write runtime metadata. To learn more about the runtime metadata supported in the Unum standard library, see the chapter on Unum runtime (Appendix B).

Unum runtime variables can appear in `Conditional` as part of the boolean expression, in `Values` as part of the invocation names, or in `Payload Modifiers` as part of the modifier instructions. The following is a list of Unum variables currently supported in the standard library and their example use cases:

- `$out` refers to the output of the function.
  - You can branch on the function output by using `$out` in the `Conditional` field of the IR as shown in the branching example previously
  - You can manually set the output of a function by writing to `$out` in the `Payload Modifier`, for example `$out="Hello World"`.
- `$n` where n is a non-negative integer refers to the branch index of the nth fan-out. For instance, `$0` is the index of the most recent fan-out (i.e., the inner-most loop), `$1` is the index of the 1st outer loop, so on and so forth.
  - You can control which branch in a Map moves forward to execute the edge and invoke the head node. For example, to only have branches with even indexes (or every other branch) invokes the head node, use `Conditional: "$0 % 2 == 0`.
  - You can have each branch fan-in with its next branch by specifying the `Values` as `[A-UnumIndex-$0, A-UnumIndex-($0+1)]` and setting the `Conditional` as `$0<$size-1` so that the last branch do not try to fan-in with its next branch which does not exist and whose index is out of bound.
  - You can modify the index in the `Payload Modifier`, for example `$0=0` to set it to 0, or `$0=$0+1` to increment it by 1.

- `$size` refers to the number of branches in the most recent fan-out (Unum supports nested fan-outs. The most recent fan-out is the inner-most loop).
  - We've seen an example of using `$size` in the `Conditional: $0<$size-1` to prevent the last branch from fan-in with a non-existent branch.
  - You can modify the size in the `Payload Modifier`, for example `$size=3`, or `$size=$size-1`.

A.2.6 Payload Modifiers

`Pop` is the only modifier currently supported in the standard library. It is designed to support nested fan-out and fan-in by removing the outer-most `Fan-out` object in the payload metadata.

Users can add any payload modifier instructions they deem necessary. Unum's design allows developers to fully custom the orchestration logic.

Appendix B

Unum Runtime

B.1 Overview

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.

In practice, the Unum library wraps around user code and interposes on user input and output. In addition to input and output data to and from user code, Unum adds runtime metadata into the input payload. The metadata helps Unum uniquely identify function invocations and enable correct execution of patterns. Moreover, the library interprets the IR to perform orchestrations, including executing the outgoing edges, checkpointing and modifying hte input payload metadata.

B.2 Input Payload Format

Every unum function is invoked with a JSON input of the following structure.

```
{
    "Data": {
        "Source": "http | dynamodb",
        "Value": "data value as a JSON object | [data store pointers]"
    },
    "Session": "uuid4 string",
    "Fan-out": {
```

```
        "Index": 1,
        "Size": 3,
        "OuterLoop": {
            "Index": 2,
            "Size": 5,
        "OuterLoop": {
            "Index": 0,
        "Size": 3
        }
        }
    }
}
```

The input contains two types of information

1. Input data for the user-defined function
2. Unum runtime metadata

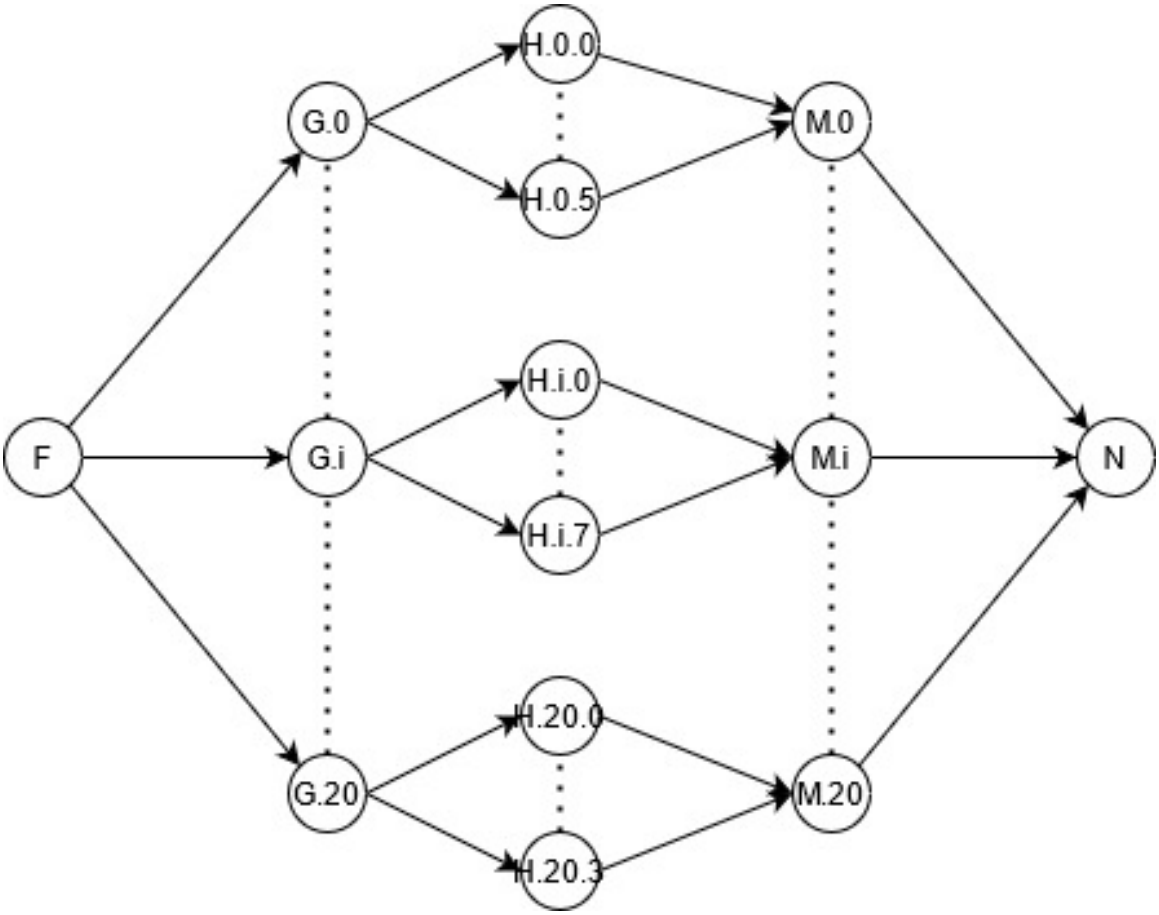
The `Data` field contains the input data to the user-defined function. Data is either passed directly in the payload or passed by reference as pointers to an intermediate data store. The current implementation supports DynamoDB as the intermediate data store. However, users can extend the runtime library to support other data stores that better suits their applications' needs. For instance, applications that process large binary data might benefit from using object stores.

The rest of the fields are Unum's runtime metadata. The main purpose of the runtime metadata is to uniquely identify each function invocation. Unum names each invocation using a combination of the function name, session ID and fan-out indexes. Details are discussed below.

B.2.1 Naming

Uniquely naming each function invocation is critical for execution correctness. Unique names not only ensure correctness for fan-in where branches are different invocations of the same function (e.g., branches of a map pattern), but also work with checkpoints to guarantee exactly-once execution.

First, to distinguish different, and likely concurrent, invocations of the same application, each application invocation has a unique name that is the session ID. The session ID is passed through the execution graph to every function at runtime in the `Session` field. The current implementation



uses a UUID4 string generated by the entry function (the function whose IR's `Start` field is set to true) of the application. Alternatively, one can use the FaaS platform's invocation identifier for the entry function.

Within an application, each function invocation is identified by its user-defined name and runtime branch indexes. The standard library implementation uses a `<function name>-UnumIndex-<a>.<b>.....<z>` format, where the `<a>...<z>` refers to the branch index at each fan-out loop, starting with the outer-most loop and delimited by `.`.

For instance, in the application below, F maps to 20 G invocations and each G invocation further maps to a varying number of H invocations. Each H invocation would be named `H-UnumIndex-<x>.<y>` where `x` is the index of its tail node G invocation and `y` is its branch index from its tail node G's map.

B.2.2 Fields

Data

[REQUIRED]

The `Data` field contains the input data to the user-defined function. Data is either passed directly in the payload or passed by reference as pointers to an intermediate data store.

- `Source` specifies where the data is coming from. It can be `http`, `dynamodb`.
  - If `Source` is `http`, data in the `Value` field should be a JSON object that the Unum runtime passes directly to the user function as input.
  - If `Source` is not `http`, data in the `Value` field is one or more pointers to a data store.
- `Value`
  - If `Source: http`, data in the `Value` field is a JSON object that the Unum runtime passes directly to the user function as input. The Unum runtime does *not* interpret what's in the `Value` field.
  - If `Source` is not `http`, the `Value` field is one or more pointers to an Unum data store. The runtime reads the data via the pointers *in order* and then pass the ordered list to the user function.
  - Pointers are *function invocation names* such as `[A]`, `[A-UnumIndex-0]`, `[A-UnumIndex-0, A-UnumIndex-1,A-UnumIndex-2]`
  - Pointer names do not support wildcards and globbing patterns. Tail nodes always pass fully-resolved names.

Session

[REQUIRED]

The `Session` field is created by the entry function. All downstream functions' input have this field with the same value. The current implementation uses UUID4 strings as session IDs.

Checkpoints names are prefixed by the session ID, for example `743d9ef6-5e89-4d18-a64d/A`.

Fan-out

[OPTIONAL]

`Fan-out` is a recursive field where outer loops are nested inside `OuterLoop`. Every time a fan-out happens, the existing `Fan-out` field from the input is moved to a nested `OuterLoop` field. Each "loop" specifies an `Index` which the branch index of this function and `Size` which is the total number of branches in the fan-out.

The `Pop` modifier removes the most recent loop and makes the first `OuterLoop` the `Fan-out` field in the input payload.

- `Index`:
  - For Map fan-out, each function instance is assigned an index that is the same as its input's index in the array.
  - For Parallel fan-out, each function is assigned an index that is the same as its index in the `Next` array.
  - Index starts at 0.
- `Size`:
  - For Map fan-out, `Size` is the input data array length.
  - For Parallel fan-out, `Size` is the array size of the `Next` field in the tail function's IR

B.3 Invoke an Application

To invoke a Unum application, clients invoke the entry function.

`Value` can be any valid JSON objects and it is passed as is to to the user function. The Unum runtime does not inspect or interpret the `Value` field content. For instance,

```
{
    "Data": {
        "Source": "http",
        "Value": "Hello!"
    }
}
```

Alternatively, one can pass input data as pointers to a data store.