

# Netherite: Efficient Execution of Serverless Workflows

Sebastian Burckhardt  
Microsoft Research  
sburckha@microsoft.com

David Justo  
Microsoft Azure  
dajusto@microsoft.com

Badrish Chandramouli  
Microsoft Research  
badrishc@microsoft.com

Konstantinos Kallas  
University of Pennsylvania  
kallas@seas.upenn.edu

Chris Gillum  
Microsoft Azure  
cgillum@microsoft.com

Connor McMahon  
Microsoft Azure  
comcmaho@microsoft.com

Christopher S. Meiklejohn  
Carnegie Mellon University  
cmeiklej@cs.cmu.edu

Xiangfeng Zhu  
University of Washington  
xfzhu@cs.washington.edu

## ABSTRACT

Serverless is a popular choice for cloud service architects because it can provide scalability and load-based billing with minimal developer effort. Functions-as-a-service (FaaS) are originally stateless, but emerging frameworks add stateful abstractions. For instance, the widely used Durable Functions (DF) allow developers to write advanced serverless applications, including reliable workflows and actors, in a programming language of choice. DF implicitly and continuously persists the state and progress of applications, which greatly simplifies development, but can create an I/O bottleneck.

To improve efficiency, we introduce Netherite, a novel architecture for executing serverless workflows on an elastic cluster. Netherite groups the numerous application objects into a smaller number of partitions, and pipelines the state persistence of each partition. This improves latency and throughput, as it enables workflow steps to group commit, even if causally dependent. Moreover, Netherite leverages FASTER’s hybrid log approach to support larger-than-memory application state, and to enable efficient partition movement between compute hosts.

Our evaluation shows that (a) Netherite achieves lower latency and higher throughput than the original DF engine, by more than an order of magnitude in some cases, and (b) that Netherite has lower latency than some commonly used alternatives, like AWS Step Functions or cloud storage triggers.

## PVLDB Reference Format:

Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient Execution of Serverless Workflows. PVLDB, 15(8): 1591 - 1604, 2022.  
doi:10.14778/3529337.3529344

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/microsoft/durabletask-netherite/tree/vldb-oct-2021>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.  
doi:10.14778/3529337.3529344

## 1 INTRODUCTION

The term *serverless* is often considered as a synonym for *Functions-as-a-Service* (FaaS), which was pioneered by Amazon [1] and is now ubiquitous [4, 6, 15, 18]. In FaaS, a function is a piece of application code designed to respond to an individual event, called the *trigger*. Compared to a virtual machine or a compute instance, a function is significantly more fine-grained, which allows for faster scheduling and better load balancing given a pool of compute resources. Furthermore, FaaS platforms support per-invocation billing. This means that a service built on FaaS is not only highly available, but is both (1) very cheap to operate under low load, and yet (2) can scale automatically to a high load, at a proportional cost. Given the potential developer productivity boost that the serverless paradigm provides, it is anticipated to become prominent for cloud applications [45, 55].

Originally, serverless functions were used primarily for small, stateless applications. However, both practitioners and researchers quickly discovered the appeal of connecting event-driven functions with stateful services, such as cloud storage, to build fully serverless applications with persistent state, elastic scaling and load-based billing. To meet this demand, cloud platform providers now support many types of triggers, and other services that help to orchestrate stateless functions into stateful workflows [5, 17, 19, 23].

Stateful serverless applications use storage not just for persistent application data, but also for checkpointing communication channels or intermediate computation results. As they can provide failure, availability, and scalability isolation between service components, persistent queues (such as Kafka [3]) have become a standard architectural pattern for cloud services.

The practical significance of storage-connected triggers is borne out by the data: in Azure Functions, for instance, stateless HTTP triggers make up less than 50% of invocations [59]. The rise in stateful serverless offerings is also met by an recent increase of research in this area (e.g., [51, 62, 65–67]). A notable example is Starling [54], a scalable database query engine on top of AWS Lambda.

**Implicit Persistence.** Using pure FaaS to develop nontrivial, stateful applications is not straightforward, as it provides limited execution guarantees [43]. For example, a function may be restarted many times before completing, possibly concurrently, and must complete within strict time limits. Furthermore, all state management and synchronization must be performed via explicit calls or triggers connected to external storage services. This can create

significant challenges for developers. For that reason, a new generation of frameworks either wrap storage services to provide APIs that can deal with partial or duplicate executions [61, 62, 66], or provide serverless abstractions that can implicitly persist application progress and/or application state and fully automate recovery [5, 9, 19, 23, 67].

While implicit and frequent persistence is highly desirable from a developer perspective, it creates significant challenges from a systems efficiency perspective. Run-of-the-mill implementations or ad-hoc composition techniques<sup>1</sup> can easily generate an excessive number of storage accesses that negatively impact latency, throughput, and cost.

**Durable Functions.** In this paper, we focus on the Durable Functions (DF) programming model. DF is part of Azure Functions, Microsoft’s FaaS platform [6]. It allows programmers to compose tasks into *orchestrations*, which are task-parallel workflows written from within mainstream programming languages. It also lets them conveniently store application state in shared objects called *entities*, which process operations reliably and serially, and supports synchronization and concurrency control via *critical sections*. DF is widely used in practice; at the time of writing, about 6% of all function apps deployed on Azure used DF.

**Serverless Message-Passing Model.** DF applications are internally translated to an intermediate representation. This representation, which we call the *serverless message-passing model*, is an extension of stateless FaaS that introduces addressable, stateful *instances*. The serverless message-passing model, like the original FaaS, is fine-grained and offers ample opportunity for parallelism and distribution. Thus, like FaaS, it enables scalability and load-based billing.

**Original DF implementation.** The original DF implementation uses individual storage operations to update instance states and to enqueue or dequeue messages. Under load, this creates a throughput bottleneck due to the limited number of I/O operations that nodes and storage permit per second (IOPS).

## 1.1 Netherite

The main contribution of this paper is *Netherite*, a distributed elastic architecture that implements the serverless message-passing model. Netherite groups fine-grained state and computation into coarse-grained partitions (shards). Then it load-balances these partitions over a variable number of hosts to achieve elasticity. Partitions communicate exclusively through asynchronous channels.

**Communication.** Each stateful instance is assigned to a fixed partition, by hashing its key. To deliver messages between partitions, Netherite uses a reliable ordered queue service, with one queue per partition.

**No two-phase commit.** As messages, not transactions, are the fundamental synchronization primitive, Netherite does not require costly distributed two-phase commit: a partition can commit messages to a send channel without having to coordinate with the receiving partition. This makes Netherite well suited for efficient execution of workflow and actor workloads.

**Local Partition Recovery and Mobility.** An individual partition can be moved without requiring coordination with other partitions, by simply persisting and then recovering its state and receive position on a different host. This partition mobility is important, as it enables Netherite to run in an autoscaled context where hosts are dynamically added and removed depending on load, which requires re-balancing the partitions.

**Commit Log and Checkpoints.** Netherite persists the state of each partition (including its receive position) both continuously using a commit log, and occasionally using checkpointing. The main benefit of the commit log is that partitions can quickly commit a batch of transitions using a single storage write, which addresses the throughput bottleneck mentioned above. This **group commit** significantly improves performance compared to the original DF implementation.

**FASTER** Netherite uses the FASTER [14, 34] open-source framework to help manage memory, logging, and interactions with storage. For example, partition persistence and recovery are handled by FasterKV, a hybrid (spans memory and storage) key-value store that uses a log-structured storage interface on top of Azure Page-Blob Storage. FASTER provides significant benefits; for example, it can handle over-subscribed partitions whose state does not fit into memory, and it can support instant recovery, which loads instance states lazily as needed.

**Persistence Pipelining.** Consider two transitions A, B that are causally dependent.<sup>2</sup> A conservative workflow engine waits for A to be persisted before starting execution of B. This is not necessary, as execution and persistence are successive stages of a pipeline: we can start executing B immediately after A completes execution, even before A is persisted; as long as we respect causal dependencies during persistence, i.e. do not commit B before committing A.

Netherite exploits that independence using local pipelining, that is, directly processing work items without waiting for their local dependencies to be persisted. Local pipelining improves latency significantly for workflows that execute within a single partition. We stuck with local (as opposed to global) pipelining because it enables simple local partition recovery, which is important in a context where partitions move frequently.

**Strong Execution Guarantees.** Netherite guarantees that each individual workflow step commits like a serializable transaction. This implies that messages appear to be processed exactly once.

## 1.2 Evaluation

Our evaluation is organized into two parts, focusing on throughput and latency, respectively. First, we compare the throughput of Netherite and the original DF implementation. To this end, we run a series of experiments where both implementations are run on the same fixed-size cluster of machines. Second, we compare the latency of executing an individual workflow for a number of typical solutions, including both DF and common DF alternatives, such as storage-based triggers and AWS step functions.

<sup>1</sup>For example, workflows are often expressed by representing each intermediate state using a file in storage, or a message in a persistent queue.

<sup>2</sup>We define causality of two transitions A, B as follows: B is causally dependent on A if B consumes a message produced by A, if B reads an instance state written by A, or if there is a transitive chain of such dependencies.

**Results.** Our results show that Netherite significantly outperforms the original DF implementation and the most common alternatives. The experiments demonstrate that Netherite successfully addresses the IOPS bottleneck, a common challenge for stateful serverless models, by drastically reducing the number of storage accesses. Therefore, Netherite scales much better than the original DF implementation, improving throughput by over an order of magnitude in some cases. Netherite also has better latency than some commonly used alternatives. For example, Netherite outperforms trigger-based composition by orders of magnitude, both on AWS and Azure. Also, Netherite achieves lower latency than AWS Step Functions: a workflow composing AWS lambdas completes faster on Netherite (even though deployed in Azure and invoking lambdas in AWS via HTTP) than the same workflow in Step Functions (deployed in AWS and invoking lambdas directly).

### 1.3 Contributions

We make the following contributions:

- (1) We propose a message-passing model for stateful serverless. It extends FaaS with stateful instances that can exchange messages, and allows fine-grained parallelism, distribution, and load-based billing. We also show how this model serves as an intermediate representation for DF, a stateful serverless programming model with support for multiple languages and paradigms (§3).
- (2) We show how the original DF engine, which is widely used in production, implements the message-passing model, and explain how its excessive number of storage accesses can limit the efficiency of the system (§4).
- (3) We propose Netherite, an alternative engine that divides the application state into a fixed number of *partitions*. Optimizing the continuous persistence of each partition, Netherite exploits locality, supports pipelining, and reduces the number of storage accesses (§5).
- (4) We evaluate the performance of Netherite and the optimizations on several applications, demonstrating performance improvements compared to the original DF implementation and alternative solutions (§6).

Our work shows that a major efficiency challenges for serverless workflows, namely the excessive number of storage accesses caused by the compute-storage separation, can be mitigated by a better engine architecture. Lowering this cost widens the range of applications that benefit from stateful serverless programming models.

We anticipate that serverless will become the default for the majority of cloud service development, while serverful programming remains relevant for lower layers of the stack only. Strong reliability plays an important role in this future; but to get there, we must adapt the classic state management techniques so they stay relevant even as the application development paradigms undergo a major transformation.

## 2 DURABLE FUNCTIONS

We start with a quick overview of Durable Functions (DF) and the programming abstractions that it offers for building stateful serverless applications. DF lets programmers write applications that combine (a) *orchestrations*, which reliably compose tasks to

```

1 [FunctionName( SimpleSequence )]
2 public static async Task<int> Run(
3     [OrchestrationTrigger] IDurableOrchestrationContext c)
4 {
5     try
6     {
7         var x = c.GetInput<int>();
8         var y = await c.CallActivityAsync<int>( F1 , x);
9         var z = await c.CallActivityAsync<int>( F2 , y);
10        return z;
11    }
12    catch (Exception) {
13        // Error handling or compensation can go here.
14    }
15 }

```

Figure 1: Sequencing two functions F1 and F2 using a durable functions orchestration in C#.

perform sequential or parallel composition and iteration, (b) *entities*, which are persistent shared objects that reliably process operations serially, and (c) *critical sections*, which provide mutual exclusion and concurrency control. Its implementation is open-source [8, 11–13], and is built on top of the Azure Functions framework [7]. The currently supported languages are JavaScript, Python, C#, and PowerShell. For a in-depth description of the DF programming model, including a formal semantics, we refer to [30].

**Orchestrations** are reliable workflows written in a task-parallel style. An example illustrating a simple orchestration, a sequential composition of two functions, is shown in Fig. 1. Lines 1–3 declare that this is an orchestration function named *SimpleSequence*. When invoked, this orchestration reads its input (line 7) and then calls an activity with name F1. The term “activity” is DF terminology for a stateless serverless function, that can take an input and return an output. We have omitted the code for the activities in our examples. The `await` on line 8 indicates that the orchestration should resume execution only after F1 is complete. The returned result is then passed to the next function F2 (line 9). When the latter finishes, the orchestration returns the final result (line 10). If anything goes wrong, the exception handler (line 13) can take appropriate action.

A slightly more interesting example containing parallel iteration is shown in Fig. 2. It shows a JavaScript example of an orchestration that creates thumbnails for all pictures in a directory. It receives a directory name as input (line 4), and then calls an activity “GetImageList” (line 6) to obtain the list of files. The `yield` on line 6 serves as a JavaScript equivalent of `await`. Next, to create the thumbnails in parallel, the orchestration starts an activity for each of them, without `yield`, thus not waiting for the result, but storing the tasks in an array (line 12). Next, it calls `yield` to indicate that the orchestration should resume after all the parallel tasks are complete (line 16). Finally, it aggregates (sums) all the returned numbers (sizes) and returns the result (line 18).

**Entities** are addressable units that can receive operation requests and execute them sequentially and reliably. Fig. 3 shows a C# example of an entity representing a bank account. The state of the entity is an integer (line 3) that is read by an operation `Get` (line 4) and updated by an operation `Modify` (line 5). An entity ID consists of two strings, the entity name and the entity key. For

```

1 const df = require( durable-functions );
2 module.exports = df.orchestrator(function*(context) {
3   // Get the directory input argument
4   const directory = context.df.getInput();
5   // Call an activity and wait for the result
6   const files = yield context.df.callActivity(
7     GetImageList , directory);
8   // For each image, call activity without waiting
9   // and store the task in a list
10  const tasks = [];
11  for (const file of files) {
12    tasks.push(context.df.callActivity(
13      CreateThumbnail , file));
14  }
15  // wait for all the tasks to complete
16  const results = yield context.df.Task.all(tasks);
17  // return sum of all sizes
18  return results.reduce((prev, curr) => prev + curr, 0);
19 });

```

Figure 2: Example orchestration using the Durable Functions JavaScript API. It calls an activity `GetImageList`, and then, in parallel, `CreateThumbnail` for each image. It then waits for all to complete and returns the aggregated size.

```

1 public class Account
2 {
3   public int Balance { get; set; }
4   public int Get() => Balance;
5   public void Modify(int Amount) { Balance += Amount; }
6
7   // boilerplate for Azure Functions (feel free to ignore)
8   [FunctionName(nameof(Account))]
9   public static Task Run([EntityTrigger]
10    IDurableEntityContext ctx)
11     => ctx.DispatchAsync<Account>();
12 }

```

Figure 3: Example entity using the Durable Functions C# API. Its state is an integer `Balance`, and it has operations `Get` and `Modify` to read or update it.

example, an account entity may be identified by ("Account", "000-7-17-12-0-14-26"). All entity operations are serialized, that is, their execution does not overlap, which provides a simple solution to basic synchronization challenges. The concept of entities is similar to virtual actors, or *grains*, as introduced by Orleans [29, 31].

**Critical sections** help to address synchronization challenges involving durable state stored in more than one place, such as in multiple entities and/or in external services. For example, consider an orchestration that intends to transfer money between accounts. Fig. 4 shows such an orchestration, using the C# API. First, we obtain the input parameters (source, destination, and amount) on line 5. Then, we construct entity IDs for the two accounts (line 7, 8). The `LockAsync` call on line 10 locks both account entities for the duration of the critical section (lines 11 through 23), enforcing exclusive access. On line 12, we read the current balance of the source account by calling the `Get` operation.<sup>3</sup> If the balance does not cover the amount (line 13) we return false (line 15), otherwise we modify both accounts by calling the two account entities in parallel (lines 20, 21). After both entities finish the operation, the

<sup>3</sup>Our .NET interface also offers an interface-and-proxy-based syntax for calling entities that provides type checking for operations and arguments. It takes a bit more space so we chose the untyped syntax for this small demonstration example.

```

1 [FunctionName( Transfer )]
2 public static async Task<bool> Transfer(
3   [OrchestrationTrigger] IDurableOrchestrationContext ctx)
4 {
5   (string source, string dest, int amount) =
6     ctx.GetInput<string, string, int>();
7   EntityId sourceId = new EntityId( Account , source);
8   EntityId destId = new EntityId( Account , dest);
9
10  using (await ctx.LockAsync(sourceId, destId))
11  {
12    int bal = await ctx.CallEntityAsync<int>(sourceId, Get );
13    if (bal < amount)
14    {
15      return false;
16    }
17    else
18    {
19      await Task.WhenAll(
20        ctx.CallEntityAsync(sourceId, Modify , -amount),
21        ctx.CallEntityAsync(destId, Modify , +amount));
22      return true;
23    }
24  }
25 }

```

Figure 4: Example of an orchestration with a critical section that reliably transfers money between account entities.

`await` (line 19) completes and we return true, exiting the critical section, and releasing both locks.

**Transactional Guarantees** The individual processing steps of orchestrations and entities commit like serializable transactions; but the orchestration as a whole does not, as it does not fail or roll back, and because the steps of multiple orchestrations are allowed to interleave observably. This is the desired default semantics for workflows. However, as mentioned above, stronger isolation is sometimes still desirable. Critical sections fill this gap, as they guarantee serializability. Yet, unlike transactions, they do not have an implicit fail-and-roll-back behavior.

### 3 SERVERLESS MESSAGE-PASSING MODEL

To aid understanding and to enable the development of different back-end engines, we propose a reliable message-passing layer. This layer acts as an intermediate representation that can decouple the feature-rich high-level stateful serverless programming models (like DF, step functions, or other workflow definition languages) from the execution engines that operate at a lower level of abstraction.

This separation is important to reduce complexity and allow each side to evolve independently. For example, when considering engine optimizations, we do not have to reason about orchestrations, entities, or critical sections, but can focus just on how to process messages correctly and efficiently. Similarly, advanced features can be added easily to the high-level programming model without requiring revisions of the engine, as long as we express those features using message-passing. To the best of our knowledge, we are the first to propose the use of a serverless, reliable message passing layer for this purpose.

In the following, we first give a precise definition of how the serverless message passing model combines stateless functions with stateful *instances* that communicate via messages (§3.1). Then we explain how arbitrary Durable Functions code can be compiled to it (§3.2), and finally discuss the execution guarantees that can be achieved by implementations of this model (§3.3).

application state	work items
	<p>stateless task  <math>\langle \cdot \rangle^0</math></p> <p>stateful instance  <math>\langle \cdot \rangle^1 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0</math>  <math>\langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0</math></p>

**Figure 5: The serverless message-passing model.** Messages in the task queue represent stateless functions scheduled for execution. The key-queue-value stores the current state and message queue of each instance. The application progresses by fetching, executing, and committing work items. Work items can consume and produce messages and update instance states.

### 3.1 Model Definition

**3.1.1 State.** We show how the application state is structured on the left of Figure 5. All of the state of an application is persisted in storage (different strategies are used by different implementations §4.5). At the top is a queue for stateless tasks; its messages represent stateless functions scheduled for executions. The bottom half is a "key-queue-value" store that contains the stateful instances. Each instance is identified by a unique key, the *instance id*, which can be used as a routing destination for messages. For each instance, the store contains a queue of incoming messages, and the current state.

**3.1.2 Transitions.** The application makes progress at the granularity of transitions called *work items*, of which there are two types (see Figure 5 on the right):

- (1) Stateless *task work items* consume a message  $\langle \cdot \rangle$  from the task queue, compute a result, and then enqueue this result in the form of a response message  $\langle \cdot \rangle^0$  to the queue of the instance that produced  $\langle \cdot \rangle$ .
- (2) Stateful *instance work items*, for an instance with *id*  $\langle \cdot \rangle$  and current state  $E$ , consume a batch of incoming messages  $\langle \cdot \rangle^1 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0$  ( $= 1$ ) from the instance queue, update the instance state to  $E^0$ , and produce outgoing messages  $\langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0 \langle \cdot \rangle^0$  ( $\geq 0$ ) that are enqueued in the task queue and relevant instance queues.

Note that the queues are not required to be entirely FIFO: that is, a transition may dequeue messages from positions other than the head of the queue. However, we do guarantee that messages from the same source are processed in order.

The system continuously identifies work items, schedules them for execution, and commits their results; the precise execution details vary by implementation (§4.5). All stateless work items

can execute in parallel; for stateful work items there is parallelism across keys but only one work item per key is executing at a time.

Work items can be seen as a generalization of stateless FaaS functions, since they execute arbitrary application code, are individually billed, and are also subject to timeouts.

### 3.2 Compiling DF to Message-Passing

All DF applications are internally compiled into instances, tasks, and messages. Instances represent the state of orchestrations and entities, tasks represent activities, and messages represent calls and responses.

**Orchestrations.** A significant challenge for work flows-as-code approaches like DF is that orchestrations are mostly written in languages that do not provide easy checkpointing of intermediate states. To solve this, DF represents the state of an orchestration as a *partial history* of events, which can be replayed to reconstruct the state. The history replay is transparent; it does not re-execute completed tasks but reuses the results recorded in the history.

**Entities.** Entities are directly translated to instances by serializing their state fields as instance state, and by translating each of their methods as a work-item that processes a call or signal to the entity, update the state, and possibly send signals to other entities.

**Critical Sections.** Mutual exclusion is achieved using a distributed two-phase locking protocol, which is expressed in the message passing model as follows. When entering the critical section, a single lock message is passed around to all the entities to be locked, i.e., each entity that processes it passes it on to the next, respecting a fixed global order to avoid deadlocks. When exiting the critical section, release messages are sent in parallel to all locked entities. See Figure 6 for an illustration of the two-phase locking, on the example of the bank transfer critical section from Figure 4.

The two-phase locking protocol is layered on top of the message model, which already handles persistence and recovery. This is an interesting benefit of using reliable messages as the fundamental synchronization primitive.

### 3.3 Execution Guarantees

In this section we first describe *serializable commit* (§3.3.1), a simple but strong requirement for implementations of the serverless message passing model. Then we discuss what serializable commit means at a higher level, in terms of the execution guarantees for the application state (§3.3.2) and for external effects (§3.3.3).

**3.3.1 Serializable Commit.** We say the engine satisfies serializable commit if the transitions (1) and (2) defined in §3.1.2, which represent the committing of work items, behave like serializable transactions. For example, committing a stateful work item means (1) removing the consumed messages from the instance queue, (2) updating the instance state, and (3) enqueueing the produced messages. Serializable commit implies that all of these effects must appear to execute atomically and in isolation.

**3.3.2 Exactly-once Execution.** Serializable commit means that if committing a work item does not succeed for some reason, the application remains in its original state. In particular, no messages are consumed or produced, and the runtime can retry the execution and the commit of the work item. Since failed commits do not have

**Figure 6: Illustration of the distributed two-phase locking protocol, on the example of the bank transfer critical section from Figure 4.**

any visible effect on the application state, all messages appear to be processed exactly once. This is a much stronger guarantee than the "at-least-once" or "at-most-once" execution that some workflow systems settle for.

Note that automatic retries of message delivery happen only for transient *runtime-internal faults*, which unavoidably happen in a distributed system but should be hidden from the application. In contrast, *application-level errors*, including function timeouts, are not automatically retried, but are handled in a manner defined by the programming model; in the case of DF, there are multiple error handling mechanisms available, such as exception handling, and limited retries. Importantly, these are implemented on top of the message-passing layer, and do not involve the execution engine.

For a formal treatment of how serializable commit guarantees exactly-once processing of DF, we refer to [30].

**3.3.3 External Effects.** Since there can be multiple execution attempts of a work item before it commits successfully, calls to external services may be duplicated. This issue is an unavoidable property of fault-tolerant workflow systems permitting external calls, and sometimes requires the application code to take extra precautions. In practice, common solutions (other than simply ignoring the problem) are to test whether a prior execution has already performed the effect, or to rely on deduplication via unique request identifiers. It is also possible to use libraries that wrap calls to storage services, e.g., with logging [66], to provide exactly-once execution transparently to the user.

Note that in many cases one can avoid these problems by using DF entities to store data, instead of an external service. In that case, the serializable commit already guarantees exactly-once.

## 4 ORIGINAL IMPLEMENTATION

We now give a brief description of the original backend architecture. Note that this implementation is not just a strawman prototype; it is currently used by nearly all DF applications in production; at the time of writing, these account for about 6% of all serverless function apps deployed on Azure.

**Elastic Environment.** The DF engine supports invocation-based billing by running on an elastic *host* abstraction provided by the Azure Functions system. At any time, a function app contains some number of hosts. The number of hosts is scaled automatically

based on load, and goes to zero when the app is idle. The runtime load-balances functions over hosts, and can run multiple functions on the same host concurrently. It also detects failures and replaces failed hosts. Like FaaS functions, hosts are unreliable, and cannot connect to each other directly.

**Architecture.** The application state from Figure 5 is persisted in storage, and the work items are executed by task and instance workers that are distributed over the hosts, as shown in Figure 7a:

Stateless tasks are stored in an Azure Queue. Each host runs a task worker that fetches and executes work items from this queue. This is also known as the "competing consumers" pattern.

Instance queues are partitioned statically by computing a hash of the key. For each partition, there is a single Azure Queue, and a single instance worker to process work items. This is enforced by using storage leases.

Instance states are stored in Azure Storage Tables. The instance workers read and update these tables.

By design, Azure Queues guarantee at-least-once delivery only. To counter that, the workers deduplicate messages where necessary.

**Performance and Cost.** Under load, the efficiency of this execution engine suffers from having to perform an *excessive number of storage accesses*. For example, the execution of a simple two-task sequence involves 5 enqueues, 5 dequeues, 3 state reads, and 3 state updates. This causes high latency. Also, it limits throughput, as the number of storage accesses cannot exceed the IOPS limits imposed per host and per storage account. Finally, it drives up cost, because storage accesses are directly billed to the user.

## 5 NETHERITE

In this section, we introduce Netherite, an execution engine that implements the serverless message passing model (§3) and can execute efficiently and reliably in the context of a distributed elastic runtime environment. We start with an overview of the architecture (§5.1) and then we zoom in on a single partition (§5.2). We then describe how Netherite implements the message-passing model state (§5.2.1). Finally, we describe how Netherite improves latency by pipelining the execution and persistence stages (§5.3).

### 5.1 Architecture

Figure 7(b) shows the Netherite architecture. Like the original DF implementation, Netherite is designed to execute on an auto-scaling collection of hosts. The particulars of the auto-scaling and failure-detection components are orthogonal, and beyond the scope of this work. What is significant is that compute hosts may appear or disappear; this means that Netherite must continuously save progress to storage, and must rebalance across the available hosts.

As a fundamental design choice, Netherite packs *all* of the application state (Figure 5) into **partitions**. The rationale is that because the number of partitions (typically 12–32) is much smaller than the number of tasks and instances (multiple thousands), it is more efficient to implement load-balancing, persistence, and reliable communication at the granularity of partitions. Also, partitions can improve locality, as we can optimize intra-partition communication.

A load balancer places partitions on the available hosts, and moves them if there is an imbalance (i.e. if the number of partitions

(a)

(b)

(c)

Figure 7: (a) Illustration of the original AzureStorage engine architecture, showing 6 hosts. A single task queue delivers task work items to all hosts. Instance work items are partitioned over 3 control queues, each connected to one a nitized worker. The instance states are stored in tables. (b) Illustration of the Netherite engine architecture, showing 5 partitions distributed over 2 hosts. Workers do not connect to storage directly, but to locally hosted partitions. Each partition has its own state and uses an optimized persistence mechanism. Partitions communicate with each other via ordered persistent queues (EventHubs). (c) Partition-internal event processing, state management, and persistence with FASTER.

on two hosts differs by more than one). To ensure that the same partition is running on at most one host, we use storage leases.

Each partition has an ordered persistent **input queue**, provided by EventHubs [24] (an Azure-hosted queue service analogous to Apache Kafka [3]). Input queues are used both for inter-partition communication, and to receive requests from clients. An input queue can receive messages even when its corresponding partition is not loaded on any host, for example, while it is being moved, or if the cluster has been scaled to zero.

Each partition persists its internal state continuously to storage using an incremental commit log and incremental asynchronous checkpoints. We describe these below (§5.2). We chose to provide continuous persistence (not just sporadic checkpointing) because it minimizes re-execution of tasks on recovery. This is desirable for workflow applications because tasks can have external effects, such as sending an e-mail, and applications may not always de-duplicate such effects. Continuous persistence also allows easy movement of a partition to a different host, as it can be quickly shut down and then recovered on the destination host. The partition state stores the last processed input queue position. This means that on recovery, a partition continues processing input messages at the correct position.

## 5.2 Partition State Persistence

To achieve efficient continuous persistence, Netherite employs *event sourcing*, a dual persistence model using a combination of a commit log and checkpoints. With event sourcing, the partition state is a

deterministic function of the sequence of events that were processed. We persist the partition state both continuously as an event log, and occasionally as a checkpoint—limiting the number of events that have to be replayed on recovery.

Figure 7(c) shows the internal architecture of a partition. At the center is the event queue, which orders all events into a linear sequence. This sequence of events is then duplicated into two streams that are processed **in parallel** and independently, decoupling persistence and processing. The stream on the left is persisted to storage as-is, creating a commit log. The stream on the right is applied to the current in-memory state of the partition (described below in §5.2.1). That state is also saved to storage periodically, to create checkpoints.

Events are generated by multiple sources, such as messages received from the input queue, completed work items from the task and instance workers, and persistence acknowledgments from storage. Each event can represent an atomic update to multiple internal components of the partition (we describe events and the partition state in detail in §5.2.1). Since each event’s effect on the partition state must be deterministic, it is often necessary to decompose processes into chains of multiple events. Because the log persistence is decoupled from the processing of events on the partition state, such event chains can be executed without having to wait for I/O in between. This enables the pipelining optimization (§5.3).

**Group commit.** As explained earlier, performing a large number of storage accesses can easily become a throughput bottleneck of a workflow processing system under load. Netherite solves this

problem because, when under load, a partition can *group commit*, i.e. commit multiple events at once using a single storage access. This is possible because the commit log improves storage locality: in contrast to storing updated instance states in a table, consecutive events are stored to consecutive locations.

**FASTER.** Netherite uses FASTER [14, 34] to manage memory, logging, and interactions with storage. FASTER offers two abstractions: FasterLog is a scalable persistent log that uses epoch protection to perform fast and reliable log group-commits and reads directly to and from storage. FasterKV is a larger-than-memory key-value store backed by a hybrid log across memory and storage. FASTER is built on a simple and general storage abstraction called *Interface* – this allows us to use the low-level, low-cost page-blob APIs of Azure Storage directly, and affords us the flexibility to experiment with other storage alternatives (e.g., Amazon S3) in future.

As shown at the bottom of Figure 7c, each Netherite partition uses FasterLog to persist (and recover) the commit log and FasterKV to host the partition state. Using FasterKV for persisting the partition state has several additional benefits:

- It allows the partition state to be larger than memory.
- It acts as a LRU cache for instance states.
- It supports asynchronous, incremental checkpointing.
- It allows instant recovery (lazy loading).
- It can index the instances, to support analytical DF queries over state.

Instant recovery is particularly important in the elastic context where we run Netherite; because unlike most databases, partitions must frequently move between hosts.

**5.2.1 Partition State.** Figure 8 contains a visual representation of the partition state and its correspondence with the application state. The most important components are:

- I. A map from instance IDs to instance states.
- P. The queue position of the last processed input, and a deduplication vector.
- S. Buffers for incoming messages, by instance ID.
- O. A buffer for outgoing messages.
- T. A list of pending tasks.

An event updates the partition state deterministically, and can modify multiple components atomically. Partition state update events are more coarse-grained than instance transitions in the sense that they usually contain a lot of instance transitions batched together. The 4 most important event types are:

*MessagesReceived.* Updates P (advances position and deduplication vector) and S (enqueues messages).

*MessagesSent.* This updates O (removes messages).

*TaskCompleted.* This updates S (enqueues response) and T (removes completed task).

*StepCompleted.* This updates I (updates instance state), S (removes consumed messages), O (adds produced messages), and T (adds produced tasks).

Note that all the components of the state of partition as well as the update events are serializable and therefore can be persisted as checkpoints or in the commit log respectively.

**Figure 8: Illustration of the partition-internal state.**

**5.2.2 Recovery.** On recovery, a partition first recovers its latest persisted state. It does so by retrieving the latest checkpoint (if any), and then replaying the commit log (if the commit log has persisted events beyond what is in the checkpoint). After this step, the states of P, S, O, I, and T have been reestablished. Next, we restart tasks:

- (1) Start a stateful work item for each session in S.
- (2) Start a stateless work item for each task in T.
- (3) Start a sender loop, which (re-)sends all messages in O.
- (4) Start a receiver loop, which starts receiving from the position stored in P.

**5.2.3 Correctness.** Netherite satisfies serializable commit because it commits work items as a single *StepCompleted* event in the commit log. Writes to the commit log are atomic, thus the transition commits at the moment this event is persisted in the log. All local effects of the transition (such as updating the instance state, or sending messages) are also committed by this single event.

Within each partition, causality is guaranteed because causally dependent events always appear in correct order in the totally ordered event queue. This queue is processed and persisted in order (Figure 7(c)). Dependent events may thus be committed simultaneously, but never out of order.

Messages destined for local instances or the local task queue are enqueued instantly. As for messages headed for other partitions, they are first stored in the outbox. Only after the producing event has been persisted in the commit log are they actually sent, and only after the send has been acknowledged are they removed from the outbox. This ensures causally consistent commit across partitions, and that messages are never lost in transit.

If a partition crashes after sending a message, but before recording the ack, it will resend the message upon recovery. To avoid duplicate processing in such cases, receivers use a deduplication vector, which records the position of the last-processed incoming message from each remote partition. Since EventHubs guarantees in-order delivery of messages, and since we can checkpoint the receive position along with the partition state, this is sufficient to guarantee reliable, ordered delivery of messages.



### 5.3 Pipelining

The original DF implementation is conservative: when a work item completes, its effects are first persisted to storage before executing dependent work items. This slows work flows down considerably. For example, consider the simple sequence shown in Figure 1 which is internally represented as a sequence of 5 work items. If each work item is persisted before the next work item is started, then the entire sequence takes at least as long as 5 storage round-trips.

To improve this, the Netherite architecture supports *pipelining* within each partition. Since the commit log persistence happens in parallel to the event processing on the partition state (Figure 7c), Netherite can start executing the next work item before the previous work item has been persisted. Causal dependencies within each partition are handled correctly because the commit log preserves the order, and always recovers to a consistent prefix. To ensure causally consistent commit across partitions, Netherite simply holds back all messages to remote partitions in the outbox  $\$$  until the work item that produced the message has been persisted.

**Correctness.** Pipelining is a sound optimization that does not compromise serializable commit. The important bit is that the pipeline never commits a transition that has a dependency on an uncommitted transition. In fact, a case can be made that the pipelining optimization is analogous to a combination of group commit and early lock release, two well-known optimizations in transaction processing.

In particular, the pipelining optimization does not compromise the exactly-once guarantee for internal state (§3.3.2) or the at-least-once guarantee for external effects (§3.3.3). The only minor difference is that without pipelining, the scope of reexecution of external effects is limited to those belonging to a single transition, while pipelining can lead to a reexecution of external effects belonging to more than one transition.

**Global Pipelining.** In a previous version, Netherite also supported pipelining across partition, i.e., propagating messages across partitions before persisting them. However, we decided to disable it and limit pipelining to a single partition because global pipelining requires significant coordination during recoveries: when one partition crashes, multiple other partitions may have to roll back to a previous state which can disrupt service of the whole system. We leave a further investigation of how to improve recovery for global pipelining, and whether the benefits outweigh the costs, for future work.

## 6 EVALUATION

We start by formulating the research questions and then describe the experiments and results (§6.1–§6.3). We use the original DF implementation that runs in production as the baseline for most of the evaluation since it can be configured and deployed in a similar way with Netherite making measurements directly comparable.

- Q1** Does Netherite improve throughput compared to the original DF implementation?
- Q2** Does Netherite reduce storage traffic compared to the original DF implementation?
- Q3** How does pipelining affect the latency of Netherite?
- Q4** How does Netherite compare with DF alternatives when considering the latency of a single work flow?

### 6.1 Throughput Experiments (Q1, Q3)

We start our evaluation with the following four benchmarks that measure the throughput of Netherite and the original DF implementation.

**Hello5.** A "hello world" workflow, each of which calls five tasks in sequence.

**Bank** The workflow from Figure 4 that implements a reliable transfer of currency between accounts.

**WordCount.** A MapReduce style workflow to analyze word frequency in books sourced from the Gutenberg dataset [46]. Mappers and reducers are represented by entities. Mappers parse the books and, for each word, send a message to the corresponding reducer.

**CollisionSearch.** A workflow that searches an integer interval for hash collisions. It is implemented using recursive divide-and-conquer: the interval is divided and sent to 10 sub-orchestrations, until is 1 billion or smaller, at which point it is searched in a sequential loop.

Each benchmark's size is variable: for the first two, we vary the number of parallel invocations. For WordCount, we vary the number of words, and for CollisionSearch, we vary the size of the interval.

**Methodology.** To get a meaningful comparison, we run both engines (original DF and Netherite) on the same deployment. We use Azure Elastic Premium plans (EP1, EP2, or EP3), configured to use a fixed number of hosts (1, 4, 8, or 12) with different numbers of cores per host (1 on EP1, 2 on EP2, 4 on EP3). We write  $N \times C$  to indicate nodes and cores in a configuration; For example,  $4 \times 2$  indicates a 4-node cluster where each node has 2 cores. The partition count is always 12. We compute throughput by dividing the size of the benchmark by the total time taken.

**Throughput Results (Q2).** Figure 9 shows that even on a single core ( $1 \times 1$ ), Netherite has higher throughput than the original implementation ( $1.1 \times - 3.5 \times$ ). When adding nodes and cores, the difference becomes much more pronounced, indicating that Netherite can take better advantage of them. Compared to the original implementation, Netherite improves the throughput by up to  $\times 12.2$  for Hello5,  $\times 7.8$  for Bank Application,  $\times 18.6$  for Word Count, and  $\times 2.35$  for Collision Search applications.

The improvements are largest on benchmarks that send large number of messages (e.g. WordCount) and less pronounced on benchmarks that are dominated by coarse-grained CPU-intensive tasks (e.g. CollisionSearch).

**Pipelining Results (Q3).** Our experiments showed that pipelining does not significantly affect the throughput. This is expected, as pipelining is a latency optimization - it does not reduce the amount of work. Still, we saw minor improvements: 6%, 1.2%, 0.8% and 0.4% for Hello5, Bank, WordCount, and Collision, respectively.

*Take away:* Netherite shows significantly better throughput and than the original implementation in all configurations, and especially at larger scales, exceeding an order of magnitude in some cases ( $\times 18.6$ ).

Figure 9: Throughput and Scalability. For each benchmark, each engine and each configuration NxC we show the mean throughput over 5 runs, normalized to the throughput of the original implementation on 1x1 (a single node and core).

## 6.2 Storage Traffic (Q2)

To better understand the throughput gains enabled by batching, we measure Netherite’s improvement over the original DF implementation in terms of storage traffic.

**Number of Storage Requests (Q2).** Figure 10 (top) shows that the batching optimization very significantly reduces the number of storage accesses (x4.4 – x71.6). The effect is particularly extreme on the WordCount benchmark, which transmits many messages.

**Amount of Data (Q2).** Figure 10 (bottom) shows that even though each partition is continuously writing to a commit log, Netherite does not increase the overall amount of data written. In fact, fewer bytes are read and written compared to the original implementation. This reduction is primarily due to Netherite representing and transmitting messages and instance states in a more compact, binary format. Again, the effect is most pronounced on WordCount, which transmits a large number of messages.

*Take away:* Netherite drastically reduces the number of storage accesses compared to the original DF implementation, addressing its main performance limitation.

Figure 10: Number of storage accesses (top) and storage access volume (bottom).

## 6.3 Comparison to Common Alternatives (Q4)

In this section we compare the performance of Netherite with commonly used solutions for serverless workflows that are offered by cloud providers. Our baselines include workflow solutions offered by both Azure and AWS: (i) composition of serverless functions with queues or triggers, (ii) AWS Step Functions [5], a declarative solution for authoring serverless workflows using JSON, and (iii) the existing Durable Functions implementation.

We compare latency only, i.e. the time taken to execute an individual workflow, because for solutions other than DF we cannot control the provisioning of machines. Thus, we cannot determine the resource efficiency, i.e. the “throughput per core” for these

implementations. Moreover, since we do not have access to the implementations of AWS Step Functions, triggers, and queues, we only report the results but do not justify the differences.

**Applications.** We use four representative workflows that vary in complexity and execution characteristics. The first two correspond to sequences of tasks: Hello3 is the same as the one used in Section 6.1 but with 3 instead of five tasks, and Sequence is a sequential workflow that takes the number of tasks = as its input argument. In our experiments, = = 10. The other two applications are more complex workflows that are taken from real applications implemented using AWS Step Functions on Github. Both of the latter two extensively use AWS Lambda and other services provided

by AWS, such as Amazon Rekognition and DynamoDB. The first is a workflow that recognizes objects in a given image and creates a thumbnail for it. It is part of a larger image processing application<sup>4</sup>. The second is taken from a real application used for database snapshot obfuscation<sup>5</sup>. The workflow state machine in Step Functions contains 27 states that interact with a variety of AWS services. Some of the tasks that it calls include user authorization, creation of database snapshots, validation of the snapshots, obfuscation of the snapshots, and publishing the snapshots in a production environment. We do not use all of the applications from Section 6.1 because some of them contain entities and critical sections and therefore cannot be implemented directly by the DF alternatives.

**Methodology.** For all workflows except the snapshot obfuscation, requests are issued at a fixed, low rate (0.1–25 requests per second) for 3–60 minutes. We then compute the empirical cumulative distribution function (eCDF) of the orchestration latency, i.e. the time it takes for an orchestration to complete, using the timestamps reported by the system. We use the system-reported latency of workflows, as opposed to the client-observed latency, because not all solutions provide a way to synchronously wait for the completion of a workflow.

**Results.** Latency results for three of the four workflows are shown in Figure 11. For the fourth, the snapshot obfuscation workflow, there is no appreciable performance difference between the implementations (Original DF, Step Functions, and Netherite); the total latency (20–25 minutes) is dominated by executing the time-consuming tasks (taking a snapshot, obfuscating it, restoring the database from a snapshot, etc.). An interesting observation is that this snapshot obfuscation workflow, which is the most complex by far, showcases programmability differences in DF and Step Functions: the Step Functions definition contains 27 states and is written using 700 lines of JSON with a lot of redundancy for error handling, while the DF implementation is more concise (about 200 lines of C# code) and uses functions for abstracting error handling.

**FaaS Sequences with Queues/Triggers.** Composition of functions with queues or triggers can be used to implement the Task Sequence workflow. As can be seen in Figure 11, in the middle, triggers<sup>6</sup> have significantly higher latencies (x1k–x10k) than Netherite. Using queues for constructing sequential workflows performs better than triggers but Netherite still achieves an order of magnitude lower latencies (median x19, 95th x29).

**Step Functions.** Step Functions does not support Sequence so it is not included in that experiment. For the other two workflows Netherite achieves better latencies (Hello3: median x30, 95th x25, image recognition: median 5%, 95th 8%). An interesting take-away is that Netherite achieves lower latency in the image recognition experiment even though Netherite is deployed on Azure and invokes AWS lambdas as its tasks using their HTTP interfaces, while AWS Step Functions invoke the lambdas directly (avoiding both the network RTT and the HTTP overhead).

**Original Durable Functions Implementation.** Compared to the original implementation of Durable Functions, Netherite achieves better latency in all experiments. In particular, it shows x16, x14, 17%, improvements in median and x31, x19, 33% improvements

in 95th percentile latency, for the Hello3, Sequence, and Image Recognition workflows, respectively.

**Pipelining Benefits (Q3).** The benefits of pipelining are apparent in all plots of Figure 11. In image recognition, the pipelining benefits are smallest because workflow latency is dominated by the execution time of the image recognition tasks. In total, the median / 95th percentile latency are improved by x7.3 / x7.1 for the Hello3 experiment, by x7.7 / x7.7 for the Sequence experiment, and by 8% / 6% for the Image Recognition experiment.

*Take away:* Netherite achieves better latencies than all other solutions in all of our experiments. Pipelining significantly improves Netherite's latency. For a workflow taken from an AWS application, Netherite achieves better latency than Step Functions even though it pays communication and HTTP costs due to being deployed in Azure and calling stateless functions deployed in AWS.

## 7 RELATED WORK

The need to augment FaaS with support for state and synchronization has been acknowledged by both the research and industrial communities [42, 45, 58, 64]. We compare DF to existing frameworks in terms of their programming abstraction and support.

**Reliable Workflows.** Many systems have acknowledged the challenges of providing reliable execution guarantees for long-running workflows. Most follow the declarative approach: Netflix's Conductor [20], Zeebe [26], and AWS Step Functions [5] use a JSON schema for authoring workflows, Fission Workflows [15] supports YAML, and Azure Logic Apps [9] supports visual design tools. Apache Airflow [2] and its productization, Google Cloud Composer [17], and Fn Flow [16], are somewhat more code-based, as the schema is constructed in code. Temporal [21] uses the same workflow-as-code approach as DF. Not all of these workflow systems are truly serverless (i.e. support scale-to-zero and load-based billing), and their internal design is not always documented or publicly available. Nevertheless, it is conceivable that some the benefits of the Netherite architecture that we have described in this paper could be applicable to them as well.

**Actors.** Entities in DF, and the instances in the computation model, are inspired by traditional actor systems like Erlang [28] or Akka [41], and especially the virtual actors of Orleans [29, 31]. The latter support persistence, but may lose actor messages, guaranteeing only "at-most-once" delivery. Similarly, the execution guarantees for Cloudflare's Durable Objects<sup>7</sup> [22, 25] and Lightbend's Akka Serverless<sup>8</sup> [10], apply only to a single object; they do not provide causal consistency guarantees or synchronization primitives that span multiple objects, like DF orchestrations.

Ray [51] a recently proposed framework for developing AI applications extends actors with tasks and provides exactly-once execution guarantees (without accounting for external effects) in the presence of faults. However, Ray does not continuously persist state in a commit log, therefore more progress is lost when recovering from a crash.

<sup>4</sup>Source at: <https://github.com/aws-samples/lambda-refarch-imagerecognition>

<sup>5</sup>Source at: <https://github.com/FINRAOS/maskopy>

<sup>6</sup>Blob in Azure and S3 in AWS.

<sup>7</sup>Durable Objects closed beta was announced on September 28th, 2020.

<sup>8</sup>Akka Serverless was formerly known as Lightbend CloudState.

**Figure 11: Latency (work flow completion time) for Netherite and other work flow solutions. Each plot shows the eCDF (empirical cumulative distribution function) on a logarithmic time scale.**

**Storage Extensions for Stateful Serverless.** There are several frameworks that extend existing storage services with stronger guarantees to hide FaaS crashes and re-executions from developers and users; these frameworks usually come together with a coordinator that orchestrates execution. PyWren [44], mu [38], and gg [37] all propose simple programming frameworks for developing parallel serverless applications by exploiting the scalability of serverless functions. Beldi [66] is a framework that supports serverless function compositions that can perform transactions on a key-value store service. Kappa [67] proposes a programming framework for serverless that addresses two issues with serverless functions: the execution time limit and the lack of coordination between different function invocations. Cloudburst [62] is a framework that extends a storage service to guarantee causal consistency for DAG compositions of serverless functions.

Durable Functions and Netherite differ from these frameworks in three ways. First, orchestrations support `async-await` code enabling complex dynamic parallelism patterns in workflows (such as the parallel thumbnail creation orchestration shown in Figure 2). Second, DF follows an object-oriented approach for application state, supporting entities—complex stateful objects that offer richer interfaces than just the read-write interface that is supported by the other frameworks. Finally, DF supports critical sections that guarantee isolation, i.e., a workflow holding a critical section is the only one that can call the locked entities, but, unlike transactions, do not fail requiring rollback.

Recent work [61, 65] investigates how to guarantee causal consistency for serverless applications, but for a workload of transactions over replicated data, not message-passing workflows. The difference is that in our model, only each message-processing step, not the entire workflow, executes transactionally.

**Engine.** The Netherite architecture is inspired by, and similar to, Ambrosia’s virtual resiliency [40], with the partitions corresponding to immortals. However, instead of a single queue, Netherite separates the commit log and input queue, and parallelizes the persistence of events with their application to the partition state, which enables pipelining. Also, Netherite keeps cold state in storage by virtue of FASTER [34].

**Partitioning.** Data and task partitioning is a concept that has been widely used in many domains, such as relational databases [27, 32, 33, 53, 56], distributed file systems [39, 47, 60], and key-value stores [35, 36]. Partitioning allows systems to scale memory, CPU,

and I/O bandwidth beyond the limits of a single machine. Specific constraints of each domain influence which resource and what challenges are the focus. For example, partitioning in relational databases optimizes for transaction processing and aggregation queries, partitioning in key-value stores optimizes for scalability, availability, and recovery time, and partitioning of computations into tasks enables parallelization and load balancing. Those latter two, parallelization and load balancing, are the most significant benefits of partitioning in Netherite.

**Serverless Functions Semantics.** Jangda et al. [43] present a formal model for FaaS and explain its limitations. They also show how to compose functions using a language called SPL and describe an operational semantics for the composition of FaaS and a key-value store service. However, unlike our work, they do not combine state, messages, and functions into a single serverless model with reliable and causally consistent execution guarantees. Burckhardt et al. [30] present an idealized fault-free semantics for the Durable Functions programming model, as well as show how to correctly implement it in a serverless context. However, they do not discuss how to realize the abstract model as a distributed serverless implementation that can execute efficiently on an elastic cluster.

## 8 CONCLUSION

Our work shows that the efficiency of stateful serverless programming models like DF, which represent applications as fine-grained instances, tasks, and messages, and which implicitly and continuously persist the application state, can be significantly improved by using a combination of partitioning and per-partition persistence optimizations. This lowers the cost and broadens the scope of applications, enabling the development of stateful, serverless applications that extend far beyond the original FaaS concept.

As future work, we would like to more thoroughly investigate whether the benefits of global pipelining can outweigh its cost, using protocols such as [48]. Also, there is a rich literature in recent database work [49, 50, 52, 57, 63] in the areas of workload-adaptive optimization and computation placement, which could prove very useful, in particular for computation-heavy applications. Finally, we are interested in further exploring how the strong execution guarantees can be extended to external effects in specialized circumstances.

## REFERENCES

- [1] [n.d.]. AWS Lambda – Serverless Compute – Amazon Web Services. <https://aws.amazon.com/lambda/>.
- [2] 2022. Apache Airflow. <https://airflow.apache.org/>.
- [3] 2022. Apache Kafka. <http://kafka.apache.org>.
- [4] 2022. Apache OpenWhisk. <https://openwhisk.apache.org/>.
- [5] 2022. AWS Step Functions. <https://aws.amazon.com/step-functions/>.
- [6] 2022. Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [7] 2022. Azure Functions Host repository. <https://github.com/Azure/azure-functions-host>.
- [8] 2022. Azure Functions PowerShell repository. <https://github.com/Azure/azure-functions-powershell-worker>.
- [9] 2022. Azure Logic Apps Service. <https://azure.microsoft.com/en-us/services/logic-apps/>.
- [10] 2022. CloudState—Towards Stateful Serverless by Jonas Bonitz. <https://www.youtube.com/watch?v=DVTf5WQlgB8>.
- [11] 2022. Durable Functions Extension (C-sharp) repository. <https://github.com/Azure/azure-functions-durable-extension>.
- [12] 2022. Durable Functions JavaScript repository. <https://github.com/Azure/azure-functions-durable-js>.
- [13] 2022. Durable Functions Python repository. <https://github.com/Azure/azure-functions-durable-python>.
- [14] 2022. FASTER. <https://github.com/microsoft/FASTER>.
- [15] 2022. Fission: Open source, Kubernetes-native Serverless Framework. <https://fission.io/>.
- [16] 2022. Fn Flow. <https://github.com/fnproject/fn>.
- [17] 2022. Google Cloud Composers. <https://cloud.google.com/composer/>.
- [18] 2022. Google Cloud Functions. <https://cloud.google.com/functions/docs/>.
- [19] 2022. Google Workflows. <https://cloud.google.com/workflows>.
- [20] 2022. Netix Conductor. <https://netix.github.io/conductor/>.
- [21] 2022. Temporal. <https://temporal.io/>.
- [22] 2022. Using Durable Objects, Cloudare Docs. <https://developers.cloudflare.com/workers/learning/using-durable-objects>.
- [23] 2022. What are Durable Functions? - Microsoft Azure. <https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview>.
- [24] 2022. What is Azure Event Hubs? - Microsoft Azure. <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about>.
- [25] 2022. Workers Durable Objects Beta: A New Approach to Stateful Serverless. <https://blog.cloudflare.com/introducing-workers-durable-objects/>.
- [26] 2022. Zeebe: A Workflow Engine for Microservices Orchestration. <https://zeebe.io/>.
- [27] Peter MG Apers. 1988. Data allocation in distributed database systems. *ACM Transactions on Database Systems (TODS)* 13, 3 (1988), 263–304.
- [28] Joe Armstrong. 1997. The development of Erlang. In *ICFP*, Vol. 97. 196–203.
- [29] Phil Bernstein. 2018. Actor-Oriented Database Systems. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering* (proceedings of the 2018 IEEE 34th international conference on data engineering ed.). IEEE Computer Society, 13–14. <https://www.microsoft.com/en-us/research/publication/actor-oriented-database-systems/>
- [30] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), Article–133.
- [31] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 16.
- [32] Stefano Ceri, Shamkant Navathe, and Gio Wiederhold. 1983. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering* 4 (1983), 487–504.
- [33] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. 128–136.
- [34] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.
- [35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon’s highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
- [36] Robert Escriva, Bernard Wong, and Emin Güzener. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 25–36.
- [37] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC’19)*. 475–488.
- [38] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. 363–376.
- [39] Sanjay Ghemawat, Howard Gobio, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.
- [40] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Raheel Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proc. VLDB Endow.* 13, 5 (2020), 588–601. <http://www.vldb.org/pvldb/vol13/p588-goldstein.pdf>
- [41] Philipp Haller. 2012. On the integration of the actor model in mainstream technologies: the Scala perspective. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*. ACM, 1–6.
- [42] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. <http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf>
- [43] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proceedings of the ACM on Programming Languages (PACML)* 3, OOPSLA (2019).
- [44] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.
- [45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). [arXiv:1902.03383](http://arxiv.org/abs/1902.03383) <http://arxiv.org/abs/1902.03383>
- [46] Shibamouli Lahiri. 2014. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Gothenburg, Sweden, 96–105. <http://www.aclweb.org/anthology/E14-3011>
- [47] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–15.
- [48] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recovery for Fast Distributed Stores. In *SIGMOD 2021 (Virtual Event, China, June 2021)*. 1090–1102.
- [49] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrisnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (Beijing, China) (SIGCOMM ’19)*. Association for Computing Machinery, New York, NY, USA, 270–288. <https://doi.org/10.1145/3341302.3342080>
- [50] Hyun Moon, Hakan Haciguzel, Yun Chi, and Wang-Pin Hsiung. 2013. SWAT: A lightweight load balancing method for multitenant databases. *ACM International Conference Proceeding Series* (03 2013). <https://doi.org/10.1145/2452376.2452385>
- [51] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’18)*. 561–577.
- [52] Vivek Narasayya and Surajit Chaudhuri. 2021. Cloud Data Services: Workloads, Architectures and Multi-Tenancy. *Foundations and Trends® in Databases* 10, 1 (2021), 1–107. <https://doi.org/10.1561/19000000060>
- [53] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. 1984. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 680–710.
- [54] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 131–141.
- [55] R. A. P. Rajan. 2018. Serverless Architecture - A Revolution in Cloud Computing. In *2018 Tenth International Conference on Advanced Computing (ICoAC)*. 88–93.
- [56] Domenico Sacca and Gio Wiederhold. 1985. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)* 10, 1 (1985), 29–56.
- [57] Jan Schaner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs. 2013. RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, New York, USA) (SIGMOD ’13)*. Association for Computing Machinery, New York, NY, USA, 773–784. <https://doi.org/10.1145/2463676.2465302>

- [58] Johann Schleier-Smith. 2019. Serverless Foundations for Elastic Database Systems. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. [http://cidrdb.org/cidr2019/gongshow/abstracts/cidr2019\\_140.pdf](http://cidrdb.org/cidr2019/gongshow/abstracts/cidr2019_140.pdf)
- [59] Mohammad Shahrad, Rodrigo Fonseca, i2%1go Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX. <https://www.microsoft.com/en-us/research/publication/serverless-in-the-wild-characterizing-and-optimizing-the-serverless-workload-at-a-large-cloud-provider/>
- [60] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.
- [61] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems (Heraklion, Greece) (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. <https://doi.org/10.1145/3342195.3387535>
- [62] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment* (2020), 2438–2452.
- [63] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (Santa Clara, CA, USA) (SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 388–400. <https://doi.org/10.1145/2987550.2987575>
- [64] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uğur, and A. Iosup. 2018. Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing* 22, 5 (Sep. 2018), 8–17. <https://doi.org/10.1109/MIC.2018.053681358>
- [65] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 83–97. <https://doi.org/10.1145/3318464.3389710>
- [66] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and Transactional Stateful Serverless Workflows. In *14th FUSENIXg Symposium on Operating Systems Design and Implementation (FOSD'20)*.
- [67] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, 16.