Durable Functions: Semantics for Stateful Serverless

SEBASTIAN BURCKHARDT, Microsoft Research, USA
CHRIS GILLUM, Microsoft Azure, USA
DAVID JUSTO, Microsoft Azure, USA
KONSTANTINOS KALLAS, University of Pennsylvania, USA
CONNOR MCMAHON, Microsoft Azure, USA
CHRISTOPHER S. MEIKLEJOHN, Carnegie Mellon University, USA

Serverless, or Functions-as-a-Service (FaaS), is an increasingly popular paradigm for application development, as it provides implicit elastic scaling and load based billing. However, the weak execution guarantees and intrinsic compute-storage separation of FaaS create serious challenges when developing applications that require persistent state, reliable progress, or synchronization. This has motivated a new generation of serverless frameworks that provide stateful abstractions. For instance, Azure's Durable Functions (DF) programming model enhances FaaS with actors, workflows, and critical sections.

As a programming model, DF is interesting because it combines task and actor parallelism, which makes it suitable for a wide range of serverless applications. We describe DF both informally, using examples, and formally, using an idealized high-level model based on the untyped lambda calculus. Next, we demystify how the DF runtime can (1) execute in a distributed unreliable serverless environment with compute-storage separation, yet still conform to the fault-free high-level model, and (2) persist execution progress without requiring checkpointing support by the language runtime. To this end we define two progressively more complex execution models, which contain the compute-storage separation and the record-replay, and prove that they are equivalent to the high-level model.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Distributed programming languages**; • **Software and its engineering** → *Concurrent programming structures*; Frameworks.

Additional Key Words and Phrases: Durable Functions, Serverless, Services, Service Composition, Programming, Workflows, Reliable

ACM Reference Format:
Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (October 2021), 27 pages. https://doi.org/10.1145/3485510

1 INTRODUCTION

Serverless computing, also known as *Functions-as-a-Service* (FaaS), has emerged as a fast-growing paradigm for developing cloud applications. A FaaS application specifies only the function definitions and the events that trigger them. The entire provisioning and scaling of machines, and the distribution of the function executions over those machines, is automatic. This reduces the

Authors' addresses: Sebastian Burckhardt, Microsoft Research, USA, sburckha@microsoft.com; Chris Gillum, Microsoft Azure, USA, cgillum@microsoft.com; David Justo, Microsoft Azure, USA, dajusto@microsoft.com; Konstantinos Kallas, University of Pennsylvania, USA, kallas@seas.upenn.edu; Connor McMahon, Microsoft Azure, USA, comcmaho@microsoft.com; Christopher S. Meiklejohn, Carnegie Mellon University, USA, cmeiklej@cs.cmu.edu.

2475-1421/2021/10-ART133
https://doi.org/10.1145/3485510

```
import azure.functions as func
import urllib.request

def main(req: func.HttpRequest):
    url = req.get_body().decode("utf-8")
    fid = urllib.request.urlopen(url)
    webpage = fid.read().decode('utf-8')
    found_free = webpage.find("free") >= 0
    return str(found_free)
```

```
"bindings": [ { "authLevel": "anonymous",
                "name": "req",
                "type": "httpTrigger",
                "direction": "in",
                "route": "checkWebPage",
                "methods": [ "get" ] },
              { "name": "$return",
                "type": "http",
                "direction": "out" } ]
```

Fig. 1. An example of a simple serverless function (left) and its trigger declarations (right). The function is triggered by an Http request, downloads a webpage from the URL specified in the request, tests if the page contains the word "free", and returns the boolean result.

development effort, provides *elastic scaling*, and is *cost-effective*. For example, it takes just a few lines of code to create a service where each web request triggers a function execution that returns a result (Figure 1). And yet, this simple function can (a) scale out automatically to handle hundreds of thousands of requests per second, and (b) operate cheaply under low load, because *load-based billing* means that the cost is proportional to the execution time and memory that was consumed.

Because of its benefits, FaaS has already been shown to be a good match for compute-intensive highly parallelizable workloads [Fouladi et al. 2019, 2017; Müller et al. 2020; Perron et al. 2020] and it continues to grow in other application domains, such as workflow processing, and microservices. A central assumption behind FaaS, and key to its relevance for developers, is the *compute-storage separation* that is becoming a dominant architectural principle for cloud services. By separating the application into ephemeral workers and a storage service, the state of an application is decoupled from the computation, making it available when a processing worker crashes or is shut down due to low load, and allowing for the workers to scale independently of the storage service.

Developer Challenges. FaaS is not limited to stateless input/output computations. Functions can call external services, such as key-value stores, queues, or databases. However, dealing with compute-storage separation can be very *cumbersome* for developers. The following desiderata are particularly challenging to implement using serverless functions.

C1 Durable Execution Progress. Many applications require reliable execution of large or long-running computations or workflows. But FaaS functions can fail transiently, and are subject to strict time limits. Thus, execution progress must be saved to durable storage. As mainstream language runtimes do not support checkpointing the state of an executing program (including the call stack, the state of variables, and the heap), it is challenging to do so in a simple, automatic, and robust manner.

C2 Durable Application State. All durable state must be stored in external storage or databases, and must be explicitly read and written whenever used. This is inconvenient and can become challenging, considering the weak execution guaranteed on FaaS, and the intricacies of the storage APIs (such as transient errors, retries, and concurrency control).

C3 Exactly-Once Processing. Storage triggers meant to process events reliably do not in fact guarantee exactly-once execution. For example, a trigger that invokes a function to process each message in a queue may launch multiple executions for the same message, possibly concurrently.

C4 Synchronization. Concurrency control must be achieved via some external service, which is quite tricky. For pessimistic concurrency control, many storage services offer *leases*, which temporarily grant exclusive access to the holder. For optimistic concurrency control, *e-tags* are common, which can detect write-write conflicts. Both of these techniques are difficult to master: leases force developers to work with timing assumptions, and e-tags can only detect conflicts, but not prevent them.

Stateful Abstractions. Asking application developers to understand and solve the issues listed above means we lose the original promise of FaaS: to cleanly separate the application logic from the concurrency, parallelism, distribution, and fault tolerance issues that are pervasive in the cloud environment. This provides a strong motivation to *augment the FaaS programming model with abstractions for state and synchronization*, hiding the challenges (**C1**–**C4**) within the runtime implementation and thereby simplifying application development. Such abstractions can also improve application performance, since they delegate the implementation of challenging patterns to the experts implementing the runtime.

Two common stateful serverless abstractions we have seen emerge are **workflows** and **actors**. Serverless workflow frameworks such as Amazon Step Functions, Azure Durable Functions (DF), and IBM Composer allow functions to be composed into larger computations [Garcia Lopez et al. 2018]. On the other hand, serverless actor frameworks such as Orleans [Bykov et al. 2011], Cloudflare's Durable Objects [CloudFlare 2020a,b], Lightbend's Akka Serverless [Bonér 2020], or Azure Durable Entities [Microsoft 2020] allow application state to be encapsulated in actors.

As new frameworks providing stateful serverless abstractions are constantly appearing, it is important to strive for a deeper understanding of the underlying principles. Online documentation, code examples, blog posts, and debates often leave many questions unanswered, particularly in the face of concurrency, parallelism, distribution, and faults. We thus follow the example of [Jangda et al. 2019], and use formal semantics to shed light on recent developments in this space.

Durable Functions. In this paper, we focus on DF [Microsoft 2020] as our target of study. DF supports multiple languages and can run in a hosted environment with load-based billing. From a semantic perspective, DF is interesting because it contains a uniquely comprehensive set of abstractions (including both workflows and actors, as well as critical sections). We start with an informal description of the DF programming model (§3), using examples. Our results are based on the publicly available documentation, the open-source implementations on Github, and on conversations with the DF team at Microsoft.

Idealized Semantics. Our formal development proceeds as a sequence of three progressively finer operational models, each of which serves a different purpose. The first, called the *high-level model* (§4), is meant to serve as a semantic reference that exposes the core programming model concepts while separating implementation details. It extends the untyped lambda calculus with an async/await construct and the DF primitives, to represent code executing inside functions, workflows, and actors. The high level model is the first formal study of DF and describes its idealized semantics, that is it assumes that there are no faults and that all processing is reliable and happens exactly once.

Reliable Execution Under Compute-Storage Separation. The second model is called the *compute-storage model* (§5) and it describes an implementation of DF that separates the runtime into two components: a durable store representing the persisted state of workflows and actors, and an elastic collection of volatile workers that execute application code. This separation is inherent to stateful serverless and enables serverless execution and billing. Unfortunately, it also poses significant correctness challenges because workers are volatile and could fail at any time, or execute the same work more than once. To validate the model in that respect, we prove that it guarantees *observably exactly-once* execution of the application; the correctness condition is a simulation relation between the compute-storage model and the high-level model (Thm. 5.3). The proof relies on the storage system providing an atomic commit mechanism that serverless workers can use to persist progress (including updated states and message queues). [1] The implication of this proof is that the internal state of a DF application, i.e., the state of all actors, queues, and the progress of all orchestrations, is not visibly affected by any transient faults or recoveries of the underlying infrastructure.

Correctness of Record-Replay. The third model is called the *replay-based model* (§6). It refines the compute storage model by illustrating how the DF runtime uses a record-replay mechanism to persist and restore the intermediate state of a workflow. This is important, as it allows DF orchestrations to be authored in languages whose runtime does not support checkpointing, such as all mainstream programming languages, yet still allows the runtime to persist and recover the workflow progress in case of a crash, as well as unload workflows from memory to avoid billing charges while they are waiting for a result. We then prove (§6.3) that this record-replay mechanism is sound for deterministic orchestrations, by showing that the compute-storage model and the replay-based model are bisimilar (Thm. 6.4).

Collaboration with the DF Team. The development of all three models and their clean separation required extracting interspersed information from documentation, code, and discussions with the DF team. A beneficial side-effect of our collaboration was that our work helped identify areas of underspecified behavior, as well as influenced the design of the two new DF backends [github 2021a,b].

1.1 Contributions

Overall, we make the following contributions:

(1) We describe how the DF abstractions help developers address the challenges (**C1**–**C4**) of stateful serverless (§3).
(2) We define an idealized (fault-free) semantics of the DF programming model, formalizing the intuition about its behavior given in the previous section (§4).
(3) We show how DF can be implemented using a compute-storage separated infrastructure in the presence of faults by defining the semantics of such an implementation and proving its equivalence with the idealized semantics (§5).
(4) We show how the progress of workflows written in general purpose programming languages can be persisted and restored by defining the semantics of a replay-based DF implementation and proving its equivalence with the compute-storage separated one (§6).

2 BACKGROUND: FAAS

Serverless frameworks, also known as Functions as a Service (FaaS), are provided by all major cloud providers as an alternative to platforms where the user has to explicitly manage the provisioning and maintenance of the servers running their application. Serverless offerings promise elasticity, i.e., fast scaling of computational resources when the load increases, and at the same time no use of resources, and therefore no cost, when there is no load. All of this administration cost is transferred from the user to the provider, which is the one that assumes management of VMs, shutting them down when there is no load, and launching more VMs and routing traffic to them when load increases above the current capacity.

Serverless offerings have not only promised, but have also successfully delivered, as a lot of recent work has shown how to leverage this abstraction for fast and cost efficient execution. For example, the gg framework [Fouladi et al. 2019] achieves significant speedups on compute-heavy tasks, e.g., the compilation of big software projects, by executing them on serverless instead of multicore cloud

[1]Atomic commit is directly supported by database services offering transactions, like AzureSQL [Microsoft Azure 2021], or can be implemented by the runtime using an appropriately designed commit protocol [Burckhardt et al. 2021]

```
def main(message: QueueTrigger) -> bool:
    current = storage.read("messagecount");
    current = current + 1;
    storage.write("messagecount", current);
```

Fig. 2. Pseudocode illustrating a naïve, flawed attempt of using a serverless function with a queue trigger to count the requests in a queue. The function is triggered for each message in the queue, reads the current count from storage, increments it, and writes it back. This counter is vulnerable to both under- and over-counting because of races and duplicate executions.

instances, with only a fraction of the cost! Other work that showcases the performance benefits of serverless includes a query engine [Perron et al. 2020] and a video processing application [Fouladi et al. 2017].

The basic building block of applications built on top of a serverless platforms is a serverless function. At first glance, a serverless function looks just like any other function. For example, Fig. 1, on the left, shows a Python function that downloads a webpage from a URL passed as an argument, tests if the page contains the word "free", and returns the boolean result. However, unlike a regular function, a FaaS function is not invoked using a standard function call. Rather, it comes with additional declarations that specifies what events should trigger the function. For example, on the Azure Functions platform, the JSON in Fig. 1 on the right indicates that the function is triggered by Http GET requests, and the result is returned to that request. Most FaaS environments include a rich set of triggers, including triggers that allow the processing of messages from storage queues.

2.1 Weak Execution Guarantees

The main attraction of FaaS is how it simplifies development while providing elastic scale and load-based billing. However, *expanding the scope from individual functions to entire applications* is not as easy as it may appear at first. For one, FaaS makes rather weak execution guarantees.

2.1.1 Execution Time Limit. Serverless functions come with a strict execution time limit; the typical default for this limit is 5 minutes. For a simple function as in Fig. 1, it may be just fine to ignore webpages that cannot be processed within 5 minutes. But this does not work for larger computations or workflows that do not terminate within a short time.

2.1.2 Partial Execution. Because FaaS functions execute in a distributed environment, they are inherently subject to faults. Besides timeouts, there are many other reasons why functions may stop mid-execution. Incomplete executions can be caused by the application itself (e.g. out-of-memory errors), the runtime (e.g. shutdown of host machines), or even the underlying hardware.

2.1.3 At-Least-Once Triggers. Storage triggers are designed to retry when uncertain whether the function executed to completion. This means a single event can trigger multiple function executions.

2.2 Threats to Consistency

The weak execution guarantees and the implicit parallelism of FaaS are harmless for stateless computations that are free of interference. However, when functions interact with state (e.g. by calling a storage service, or just any stateful service in general), consistency issues can arise.

2.2.1 Non-Atomic Updates. If a function means to update multiple storage locations (e.g. update two account balances), a partial execution (§2.1.2) means that it may execute only some of the updates.

2.2.2 Concurrency Control. Functions are inherently parallel, which can cause races. For example, the naive counter in Fig. 2 has a race condition which can lead to under-counting: two concurrent invocations may interleave in such a way that the count is incremented only once. Solving this problem can be quite difficult in general, as it requires the use of external synchronization mechanisms (such as storage leases or storage e-tags) which are difficult to master.

2.2.3 Effect Duplication. At-least once triggers (§2.1.3) can lead to effect duplication. For example, the function in Fig. 2 may over-count the number of requests if invoked multiple times for the same message. To deal with duplicate executions, an often-repeated advice for programmers is to "make their functions idempotent", which is not always easy. The practical consequence is that application code is often sub-correct, or littered with deduplication mechanisms, or both.

2.3 Serverless Workflows

To enable developers to execute long-running computations and workflows in a FaaS environment requires dealing with the issues outlined above. A number of methodologies and runtime frameworks have been proposed to this end. All of them require the application developer to explicitly decompose computations into small steps that fit within the serverless time limit and can be retried when necessary. But they differ significantly in the generality of supported workflows, how those workflows are expressed, and how they are executed:

- *(External Coordinator)* One can execute the orchestration code on a coordinator machine that is not serverless itself [Fouladi et al. 2019].
- *(Trigger-based workflows)* One can use triggers to compose functions. For example, to sequence two functions, the first function can write to a file or queue, which then triggers the execution of a second function. Conventional triggers have limited expressivity, but more powerful triggers can be added to support workflows involving aggregation [López et al. 2020].
- *(Declarative workflows)* One can use declarative schemas, such as XML-based workflows [Shegalov et al. 2001], JSON-based step functions [Amazon 2020], or visual design tools [Microsoft Azure 2020].
- *(Replay-based workflows)* One can express orchestrations using regular code, but execute them in a special mode that logs progress and can resume intermediate states via replay [Burckhardt et al. 2021; Zhang et al. 2020a].

The replay-based approach, also known as *workflows-as-code*, is what underlies DF orchestrations, and we will explain this solution, formalize it, and prove its salient properties in §4–§6.

2.4 Serverless Actors

Actors are a well-known paradigm for handling the challenges of concurrency and distribution. In some sense, they are a stateful equivalent of functions, as both represent a "minimal unit" that can then be composed to build large systems. Actor systems like Erlang [Armstrong 1997] or Akka [Haller 2012] are widely used in industry to build scalable distributed systems. Orleans [Bykov et al. 2011] can be considered a pioneer of "serverless" actors in the sense that it provides location transparency and can automatically load-balance actors over an elastic cluster. However, Orleans is not available in a hosted environment with load-based billing. More recent examples of actor-like abstractions making an appearance on hosted platforms include Azure's Durable Entities (§3.3), Cloudflare's Durable Objects[2] [CloudFlare 2020a,b] and Lightbend's Akka Serverless[3] [Bonér 2020].

[2]Durable Objects closed beta was announced on September 28th, 2020.
[3]Akka Serverless was formerly known as Lightbend CloudState.

function type	programming paradigm	unit of progress	determinism required
Activity	FaaS	completion	no
Orchestration	async/await task parallelism	completed tasks	yes
Entity	virtual actors	operations	no

Fig. 3. Types of functions used by the DF programming model.

2.4.1 Reliability. The vast majority of actor frameworks make either no reliability guarantees, or give only partial guarantees (at-least-once or at-most-once). The only actor systems to provide exactly-once delivery, to the best of our knowledge, are DF and Ambrosia [Goldstein et al. 2020]. Ambrosia's actors are not intended for a serverless architecture: they are coarse-grained, not load-balanced, and offer no load-based billing.

3 DURABLE FUNCTIONS

In this section we introduce the Durable Functions (DF) programming model, which is a component of Azure Functions, Microsoft's FaaS platform. We use Python for all code here, but DF supports several programming language frontends including JavaScript, C#, and PowerShell. DF defines three types of functions (see also Fig. 3):

(1) *Activities* are the DF-equivalent of stateless FaaS functions.
(2) *Entities* are actors that encapsulate application state and process operations one at a time.
(3) *Orchestrations* use task-parallel async-await style code to coordinate activities and entities.

As the name suggests, *durability is implicit* in DF: the progress of orchestrations and the state of entities are automatically saved to storage and transparently restored after faults. In that sense, the DF model *unifies* compute and storage. This ability to run failure oblivious code in a failure resilient manner is also known as *virtual resiliency* [Goldstein et al. 2020].

3.1 Activities

Activities are the equivalent of FaaS functions in DF, and their definitions look similar to Fig. 1. Activities are automatically retried under partial execution (§2.1.2). However, if an activity exceeds the FaaS time limit (§2.1.1), or if the application code in the activity throws an unhandled exception, it is *not* automatically retried. Rather, the activity is considered to have completed with an exception result, and the exception is re-thrown in the parent orchestration that invoked it.

3.2 Orchestrations

Orchestrations allow developers to create long-running computations and workflows by decomposing the computation into *tasks*. In DF, those tasks can be either activities, entity operations, timers, or sub-orchestrations. The tasks are composed following the async/await paradigm; where in Python, the `yield` keyword is used to represent await [Chris Gillum 2021]. The DF runtime automatically saves the progress of orchestrations to storage and can recover the result of already completed tasks after faults without reexecuting them (addressing **C1, C3**).

3.2.1 Sequential composition. The orchestration in Figure 4 composes three activities in sequence. It first calls an activity that downloads its input data (Line 3), which could happen using a call to a storage service such as a database. The orchestrations then waits, as expressed by the `yield` on line 3, for the activity to finish and return the `dataset` result. After the data is downloaded, it calls the next activity that processes the data and waits for the outputs (line 4). Then it calls a summarization activity that produces a summary of the processing results (line 5) and returns it to the user (line 6).

```python
1 def orchestrate_pipeline(context: df.DurableOrchestrationContext):
2     try:
3         dataset = yield context.call_activity('DownloadData')
4         outputs = yield context.call_activity('Process', dataset)
5         summary = yield context.call_activity('Summarize', outputs)
6         return summary
7     except Exception as exc:
8         yield context.call_activity('CleanUp')
9         return f"Something went wrong" {exc}"
```

Fig. 4. Sequencing three activities DownloadData, Process, and Summarize using a durable functions orchestration written in Python.

```python
    def orchestrate_thumbnails(context: df.DurableOrchestrationContext):
        # Get the directory that was given as an input argument
        directory = context.get_input()
        # Call an activity that lists all images in a directory and wait for its result
        images = yield context.call_activity('GetImageList', directory)
        # For each image, call activity without waiting and store the task in a list
        tasks = []
        for img in images:
            tasks.append(context.call_activity('CreateThumbnail', img))
        # Wait for all the tasks to complete
        results = yield context.task_all(tasks)
        # Return sum of all sizes
        return reduce(lambda x, y: x + y, results, 0)
```

Fig. 5. Example orchestration that (i) calls an activity `GetImageList`, and then, in parallel, `CreateThumbnail` for each image. It then waits for all to complete and returns the aggregated size.

```python
def orchestrate_periodic_job(context: df.DurableOrchestrationContext):
    # Call an activity that runs at a regular interval
    yield context.call_activity('PeriodicJob')
    wakeup_time = context.current_utc_datetime + timedelta(days=1)
    yield context.create_timer(wakeup_time)
    context.continue_as_new()
    return
```

Fig. 6. Example eternal orchestration that performs some task, waits for a day, and then starts over from the beginning.

Since orchestrations use standard Python control flow semantics, exceptions can be handled as usual. In this example, we enclosed the three activities in a **try** expression that handles exceptions raised by any of the three activities (line 7), calling some cleanup activity (line 8).

3.2.2 Parallel composition. Instead of waiting for a task to finish executing before invoking the next, we can also start multiple tasks at once, and then wait for them to complete later. Figure 5 shows an orchestration that uses this pattern to execute a dynamically determined number of independent activities in parallel. It first calls an activity to get the image list in an external storage service, and then it calls an activity to create a thumbnail for each image in parallel, waiting for all of them to finish to return an aggregate of their sizes. This *fan-out, fan-in* pattern is very common; it strongly benefits from the elastic scaling of the underlying FaaS platform, but is difficult to express on pure FaaS.

3.2.3 Eternal orchestrations. It can be desirable for orchestrations to last for a long time, or even forever. For example, to execute a periodic job once a day, an orchestration can run an infinite loop and use a timer. Writing an infinite loop directly in code is however problematic, as DF uses a record/replay mechanism (more about this in §3.5) that could lead to unbounded history growth. To support infinite loops, DF thus provides a `continue_as_new(x)` primitive which restarts the orchestration from the beginning, passing x as the input. Fig. 6 shows an example of an eternal orchestration that runs a job (line 3), creates a day-long timer and waits for it to complete (line 4), and then starts from the beginning (lines 5,6).

3.2.4 Client Operations. A `client` object is used to interface with the DF runtime. When starting an orchestration, one may optionally specify an instance id for the orchestration. If so, the system checks if that instance already exists and filters duplicates:[4]

```
try:
    await client.start_new("myOrchestration", instance_Id, input_argument)
    print("new orchestration started")
except InstanceIDExc:
    print("orchestration already exists")
```

Clients can also check the status of an orchestration with a specific instanceId, to determine if it completed, and access the output value:

```
status = await client.get_status(instance_id)
if status.runtime_status is df.OrchestrationRuntimeStatus.Completed:
    print("the result is", status.output)
```

3.3 Entities

Entities allow applications to encapsulate durable state, and to define the operations that can be performed on it (addressing **C2**). Like actors, entities store operation requests in a queue and execute them one at a time (addressing **C4**). Like virtual actors in Orleans [Bykov et al. 2011], entities are identified by an entity ID that consists of two strings, the entity name and the entity key. For example, an account entity may be identified by ("Account", "000-7-17-12-0-14-26"). Fig. 7 shows example code for an account entity that supports three operations, for depositing, withdrawing, and checking the account balance.

3.3.1 Orchestration Calls and Signals. Orchestrations can access entities either by *calling* an operation, which means the orchestration creates a task that can be awaited and that returns a result, or by *signaling* an operation, which sends a "fire-and-forget" one-way message. For example, the following code first deposits money (fire-and-forget) and then checks the balance (waits for response message):

```
entityId = df.EntityId("Account", "000-7-17-12-0-14-26")
context.signal_entity(entityId, "deposit", 1)
new_balance = yield context.call_entity(entityId, "get")
print("new balance =", new_balance)
```

DF guarantees that all messages sent to the same entity by the same orchestration are delivered in order. Thus, for the above example it guarantees that the deposit operation is performed before the get operation.

3.3.2 Entity-To-Entity Signals. Entities can signal other entities. This enables useful patterns, such as using entities to represent stateful streaming operators, or even complex dataflow graphs. Also, signals can be used to implement indexing or view maintenance.

[4]The precise conditions for deduplication are controlled by the ‘OverridableExistingInstanceStates‘ configuration parameter.

```python
class Account:
    # Initialize the entity state, which defaults to 0 balance
    def __init__(self):
        self.balance = 0

    # Return the balance
    def get(self):
        return self.balance

    # Deposit an amount, increasing the balance
    def deposit(self, amount):
        self.balance += amount
        return self.balance

    # Withdraw an amount, reducing the balance
    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

Fig. 7. Example entity that corresponds to a bank account supporting three operations: (i) depositing money to the account, (ii) withdrawing money, and (iii) simply getting its balance.

3.3.3 No Entity-To-Entity Calls. To prevent deadlocks, DF does not allow entities to call other entities. For example, consider two entities k_1, k_2 that each are currently executing some operation o_1 and o_2, respectively. If we allowed entity-to-entity calls, o_1 could call an operation on k_2, and o_2 could call an operation on k_1. Then, both requests would be stuck and both entities would be (durably and reliably) deadlocked forever.

3.3.4 Scheduled Signals. DF supports signals to be scheduled for a specified delivery time. This can be useful for operations that need to be performed at a specific time. An entity can also implement a periodic background operation by sending a scheduled signal to itself. Another interesting use of scheduled signals is to deliver high-priority operations. For example, consider an entity whose input queue is backlogged (e.g. because operations are processed too slowly). Sending a scheduled message can bypass that queue and be delivered to the entity directly, because scheduled messages are not subject to the in-order delivery guarantee.

3.4 Critical Sections

A common requirement when interacting with multiple entities is that invocations across entities must happen at once, atomically, to ensure that a specific invariant is not violated by concurrent invocations. Durable Functions addresses this by supporting critical sections, i.e., regions of the code where only one orchestration is allowed to call specific entities. This enables orchestrations to modify or read from multiple entities atomically (addressing **C4**). Figure 8 shows an orchestration that transfers money between accounts ensuring that the source account has an adequate balance to prevent overdraft. It achieves that using a critical section (line 6). [5] The orchestration first obtains its inputs and then acquires the locks for both account entities to ensure that no other invocation to the entities happens while the locks are kept. It then checks if the balance covers the amount, and if so performs the transfer, returning `True`, and releasing the locks.

[5]Critical sections are only supported by the C# API at the time of writing, but we show the expected interface using Python to avoid confusing readers with multiple languages.

```
def transfer_safe_orchestration(context: df.DurableOrchestrationContext):
    # From input, get entity ids for source and destination account, and the amount
    [source, dest, amount] = context.get_input()
    # Critical Section: Acquire a lock for each entity
    with (yield context.lock([source, dest])):
        # Make sure that the source account has adequate balance to avoid overdraft
        source_balance = yield context.call_entity(source, "get")
        if source_balance < amount:
            return False
        else:
            # Wait for transfer to complete
            yield context.task_all([context.call_entity(source, "withdraw", amount),
                                    context.call_entity(dest, "deposit", amount)]);
            return True
```

Fig. 8. Example of an orchestration with a critical section that reliably transfers money between account entities checking for overdraft.

3.4.1 *Progress and Fairness.*

To ensure progress, a critical section must specify all the entities it wishes to access: while executing, it can call only entities that are already locked, it cannot call sub-orchestrations, and it cannot enter another critical section. This allows the runtime implementation to use a fair, deadlock-free distributed locking protocol.[6]

3.4.2 *Critical Sections vs. Transactions.*

Like transactions, critical sections guarantee atomicity and isolation. Unlike transactions, they do not require the programmer to handle failures or issue retries, and operate reliably even in the presence of contention.

3.5 Correct Use of Orchestrations

Internally, DF persists and restores the intermediate state of an orchestration by recording and replaying task results in a history (§6). However, this only works if the user follows certain guidelines.[7]

For one, the orchestration code must be *deterministically replayable*. Any code that reads non-deterministic data sources or calls I/O should not be done directly in the orchestration, but must be wrapped in an activity, to ensure it is properly recorded and replayed. For commonly used operations (e.g. read the current time, create a random new GUID), the API also provides built-in support for convenience.

Secondly, *history size* can become an issue if it grows too large to be replayed quickly. Developers whose workflows execute a very large number of tasks must structure them so that only small portions of the overall history have to be replayed at any given time. This can be done either by calling sub-orchestrations (sequentially or in parallel), or by using continue-as-new (§3.2.3).

4 HIGH-LEVEL MODEL

We now formalize the DF programming model, by modeling the system as a collection of components, and representing the code executing within a component as an untyped lambda expression.

4.1 Expression Syntax

We use a standard call-by-value evaluation, with a minimal syntax based on variables x, functions $\lambda x.e$, values v, expressions e, and function application $e\ e'$:

[6]The current implementation achieves this by acquiring the locks in a globally consistent order, one at a time.

[7]Using DF orchestrations correctly can be tricky for new users. For C# applications, DF thus includes a static analyzer that can warn users about suspected error.

		context restrictions			
Syntax	Meaning	Act.	Orch.	Entities	CS
$e ::= \; v \mid e\ e' \mid \mathit{await}\ e$	(basic evaluation)	✓	✓	✓	✓
$\mid \mathit{call}\ n_A(e)$	(call activity)	–	✓	–	–
$\mid \mathit{call}\ n_E.o(e, e)$	(call entity)	–	✓	–	$\in$ lockset
$\mid \mathit{call}\ n_X(e)$	(call external service)	✓	–	✓	–
$\mid \mathit{continue}\ n_O(e)$	(continue as new)	–	✓	–	–
$\mid \mathit{signal}\ n_E.o(e, e)$	(signal entity)	–	✓	✓	$\notin$ lockset
$\mid \mathit{get}$	(read entity state)	–	–	✓	–
$\mid \mathit{set}\ e$	(update entity state)	–	–	✓	–
$\mid \mathit{lock}\langle \overline{e} \rangle\ e$	(critical section)	–	✓	–	–

execution context: $E[\circ] ::= \circ \mid E[\circ]\ e \mid f\ E[\circ] \mid \mathit{await}\ E[\circ] \mid \mathit{call}\ n_O(E[\circ]) \mid \mathit{call}\ n_A(E[\circ]) \mid \mathit{call}\ n_E.o(E[\circ], e) \mid \mathit{call}\ n_E.o(v, E[\circ]) \mid \mathit{call}\ n_X(E[\circ]) \mid \mathit{signal}\ n_E.o(E[\circ], e) \mid \mathit{signal}\ n_E.o(v, E[\circ]) \mid \mathit{set}\ E[\circ] \mid \mathit{continue}\ n_O(E[\circ]) \mid \mathit{lock}\langle \overline{v}\ E[\circ]\ \overline{e} \rangle\ e$

Fig. 9. Expression syntax (top) and definition of execution context (bottom).

$f ::= \lambda x.e$	(function)
$x ::= \ldots$	(variable)
$c ::= () \mid \ldots$	(constants)
$v ::= c \mid x \mid \mathit{error} \mid f \ldots$	(value)
$e ::= v \mid e\ e' \mid \ldots$	(expressions, see Fig. 9 for complete list)

Functions are defined by a variable name x and a body e. Constants are defined as usual, with () representing the unit or "void" value. Expressions include values and function application, as well as all the operations for communication and synchronization shown in (Fig. 9). Values include constants, variables, and functions as usual, as well as a constant *error* used to propagate runtime errors. (Fig. 9) shows the list of all expressions. For each expression it also indicates the contexts (activities, entities, orchestrations, or critical sections) in which it can be used.

4.1.1 Extensions and Sugar. Our calculus is meant to illustrate semantics and enable proofs, not write actual applications, so we keep it minimal. In principle though, it could be extended to include advanced types, features and control structures (including imperative ones) if desired [Pierce 2002]. For example, let-expressions and sequential composition can be defined as syntactic sugar:

$$\begin{aligned} \mathbf{let}\ x = e\ \mathbf{in}\ e' &\equiv (\lambda x.e')\ e \\ e;\ e' &\equiv \mathbf{let}\ x = e\ \mathbf{in}\ e' \end{aligned}$$

where on the second line, x is assumed to not be free in e'.

4.1.2 Asynchrony. The ability to overlap the latency of multiple calls is paramount for service-oriented applications. We enable this in our formalization by supporting *futures* [Flanagan and Felleisen 1995; Halstead 1985; Moreau 1996]. With futures, we can easily execute two calls n_1, n_2 in parallel: $\mathbf{let}\ x_1 = \mathit{call}\ n_1()\ \mathbf{in}\ (\mathbf{let}\ x_2 = \mathit{call}\ n_2()\ \mathbf{in}\ (\mathit{await}\ x_1; \mathit{await}\ x_2))$. We represent futures by adding unresolved and resolved *placeholders* to the syntax of values:

$v ::= \cdots \mid p \mid \mathit{done}\ \rho$	(value)
$\rho ::= c \mid \mathit{error}$	(execution result)
$p ::= P_i$	(placeholder)
$i \in I$	(unique identifier)

A placeholder P_i represents the result of an asynchronous request with identifier i. When resolved, a placeholder is replaced by *done* ρ where ρ is either a constant or the special *error* value.

4.1.3 Local Evaluation $\boxed{e \longrightarrow e'}$. To formalize call-by-value evaluation of expressions, we define the execution context $E[e]$ in Fig. 9. The meaning of $e = E[e']$ is that e' is the subexpression of e that takes the next step. We define a local evaluation step $e \longrightarrow e'$ of expressions as follows:

$$\text{Eval-App}\ \frac{}{E[(\lambda x.e)\ v] \longrightarrow E[e[v/x]]} \qquad \text{Eval-Await}\ \frac{}{E[\mathit{await\ done}\ \rho] \longrightarrow E[\rho]}$$

$$\text{Eval-Error}\ \frac{}{E[\mathit{error}] \longrightarrow \mathit{error}}$$

Our model does not yet support error handling, so errors always propagate to the top level. Adding exception handling should pose no major difficulties, however.

4.1.4 Application Definition. Applications are defined by mapping names to function definitions. The namespaces n_A, n_O, n_E, n_X represent activities, orchestrations, entities, and external service calls, respectively, and o represents entity operation names.

n	$::= n_A \mid n_O \mid n_E \mid n_X$	(invocation target name)
o	$::= \ldots$	(entity operation name)
$\mathcal{A}$	$::= \overline{(n_O : f)}\ \overline{(n_A : f)}\ \overline{(n_E.o : f)}$	(application definition)

4.2 System States and Transitions

We use a labeled transition system to describe our model and reason about observational equivalence. A *system state* C is defined to be an unordered sequence of components $\mathbb{C}$, which include activities and orchestrations (§4.3) as well as entities (§4.4). The initial state C^0 is the empty sequence ϵ.

A system transition α is of the form $C \xRightarrow{\ell_1 \ell_2 \ldots \ell_n}_{\mathcal{A}} C'$ (where $n \geq 0$). It indicates that given an application $\mathcal{A}$, the state $\mathit{pre}(\alpha) = C$ can transition to the state $\mathit{post}(\alpha) = C'$, with global effects indicated by the sequence of transition labels $\mathit{labels}(\alpha) = \ell_1 \ell_2 \ldots \ell_n$. The labels are:

transition label	$\ell ::=$	$\epsilon \mid i \mid \mathit{in}(m) \mid \mathit{out}(m)$
unique identifier	$i ::=$	$\ldots$

ϵ represents the absence of a label. The label i indicates the creation of a fresh identifier. It is included for proof-technical purposes; it allows us ensure that "fresh" identifiers are globally distinct. The labels *in* and *out* model communication with the environment, and are considered *externally observable.* $\mathit{in}(m)$ indicates that a message m is received from the external environment, and $\mathit{out}(m)$ indicates that a message m is sent to the external environment.

Definition 4.1. An *execution* α of $\mathcal{A}$ is a finite or infinite sequence of transitions $\alpha_0, \alpha_1, \ldots$ such that $\mathit{pre}(\alpha_0) = C^0$, $\mathit{post}(\alpha_k) = \mathit{pre}(\alpha_{k+1})$ for all k, and the identifiers i appearing in $\mathit{labels}(\alpha_k)$ are distinct.

External Requests and Responses. All messages exchanged with the environment are either request messages or response messages (but both of those can go in either direction). We use unique identifiers i to correlate them, and we use the position (subscript or superscript) of the identifier to syntactically distinguish requests from responses. In the high-level model, the messages are:

message $\quad m \quad ::= \quad \mathit{startnew}(d, n_O, c)_i \mid \mathit{ok}^i \mid \mathit{alreadyexists}^i \mid n_X(c)_i \mid \rho^i$

The $\mathit{startnew}(d, n_O, c)_i$ is sent to the system to start a new orchestration with instance id d; the system replies with either ok^i or $\mathit{alreadyexists}^i$ (§4.3.1). The message $n_X(c)_i$ is sent from the system to the environment and represents a call to an external service n_X with argument c (§4.5). Its reply is of the form ρ^i, where ρ is either a constant or runtime error.

Abbreviated Notation for Transitions. For better readability, we drop $\mathcal{A}$ if there is no ambiguity to it, and we use + to indicate consumed external messages on the left and produced external messages on the right of the arrow; for example,

$$C + m_1 \overset{i}{\Longrightarrow} C + m_2 \quad \text{is short for} \quad C \xRightarrow{in(m_1)\ i\ out(m_2)}_{\mathcal{A}} C'$$

4.3 Orchestrations and Activities

We define the following state components:

component	$\mathbb{C}$	$::=$	$\mathbb{A}_i(x_A) \mid \mathbb{O}_d(x_O) \mid \ldots$
instance id	d	$::=$	c
activity execution state	x_A	$::=$	$\mathit{busy}_d\ e \mid \mathit{completed}\ \rho$
orchestration execution state	x_O	$::=$	$\mathit{busy}\ e \mid \mathit{completed}\ \rho$

$\mathbb{A}_i(x_A)$ is an activity with identifier i and $\mathbb{O}_d(x_O)$ is an orchestration with instance id d, which is a constant string c. The execution states of activities and orchestrations can either be busy, where e indicates the current evaluation state, or completed with some result ρ. For an activity, the subscript d indicates the parent orchestration. To match both busy states with a single and short syntax in our inference rules, we define $b\ e ::= \mathit{busy}\ e \mid \mathit{busy}_d\ e$.

4.3.1 Starting Orchestrations. Clients can start a new orchestration. If an instance with the given id already exists, an error is returned.

$$\text{StartNew-Fresh}\ \frac{\mathbb{O}_d \notin C \qquad (n_O : f) \in \mathcal{A}}{C + \mathit{startnew}(d, n_O, c)_i \Rightarrow C\ \mathbb{O}_d(\mathit{busy}\ (f\ c)) + \mathit{ok}^i}$$

$$\text{StartNew-Conflict}\ \frac{}{C\ \mathbb{O}_d(b\ e) + \mathit{startnew}(d, n_O, c)_i \Rightarrow C\ \mathbb{O}_d(b\ e) + \mathit{alreadyexists}^i}$$

4.3.2 Local Evaluation. Local evaluation is simply lifted from expression evaluation $e \longrightarrow e'$:

$$\text{Act-Local}\ \frac{e \longrightarrow e'}{C\ \mathbb{A}_i(b\ e) \Rightarrow C\ \mathbb{A}_i(b\ e')} \qquad \text{Orch-Local}\ \frac{e \longrightarrow e'}{C\ \mathbb{O}_d(b\ e) \Rightarrow C\ \mathbb{O}_d(b\ e')}$$

4.3.3 Calling Activities. When calling an activity, a fresh identifier i is generated, and the call evaluates to a placeholder P_i. The identifier i is also used as a suffix on the activity component.

$$\text{Orch-Call-Act}\ \frac{(n_A : f) \in \mathcal{A}}{C\ \mathbb{O}_d(b\ E[\mathit{call}\ n_A(c)]) \overset{i}{\Longrightarrow} C\ \mathbb{O}_d(b\ E[P_i])\ \mathbb{A}_i(\mathit{busy}_d\ (f\ c))}$$

4.3.4 Activity Timeouts. Activities are subject to FaaS time limits and can therefore time out. We model this using a nondeterministic rule that replaces a busy execution state with an error state:

$$\text{Act-Timeout}\ \frac{}{C\ \mathbb{A}_i(b\ e) \Rightarrow C\ \mathbb{A}_i(b\ \mathit{error})}$$

4.3.5 Completing Activities and Orchestrations. Activities and orchestrations are complete when the expression has evaluated to a result ρ, which is either a constant c or *error*. They then transition into a state *completed* ρ. Activities also replace all matching placeholders in the calling orchestration with the result. Note that it is not enough to replace a single occurrence of the placeholder, since placeholders can get replicated during expression evaluation.

$$\text{Orch-Done}\ \frac{}{C\ \mathbb{O}_d(\mathit{busy}\ \rho) \Rightarrow C\ \mathbb{O}_d(\mathit{completed}\ \rho)}$$

$$\text{Act-Done}\ \frac{}{C\ \mathbb{O}_d(x_O)\ \mathbb{A}_i(\mathit{busy}_d\ \rho) \Rightarrow C\ \mathbb{O}_d(x_O[\mathit{done}\ \rho / P_i])\ \mathbb{A}_i(\mathit{completed}\ \rho)}$$

4.3.6 Continue-as-new. An orchestration that calls *continue* $n_O(c)$ immediately restarts the orchestration, passing c as the input argument.

$$\text{Orch-Continue}\ \frac{(n_O : f) \in \mathcal{A}}{C\ \mathbb{O}_d(busy\ E[continue\ n_O(c)]) \Rightarrow C\ \mathbb{O}_d(busy\ (f\ c))}$$

4.4 Entities

We define the following state component to represent entities:

component	$\mathbb{C}$	::=	$\cdots \mid \mathbb{E}_k(q, x_E)$
entity key	k	::=	(n_E, c)
entity execution state	x_E	::=	$idle_\sigma \mid busy_\sigma\ e \mid busy_{\sigma,d,i}\ e \mid idle^j_\sigma \mid busy^j_{\sigma,d,i}\ e$
entity state	σ	::=	c
request queue	q	::=	$\overline{r}$
request	r	::=	$(d, o, c) \mid (k, o, c) \mid (d, o, c, i)$

An entity component $\mathbb{E}_k(q, x_E)$ represents the state of the entity scheduler of the entity with key k. The key k is a pair consisting of the entity name n_E, and a string c. The first component q is an ordered queue of operation requests that are waiting to be serviced.

The execution state s can represent that the entity is idle, busy processing a signal, or busy processing a call. The subscript σ is the current entity state, which is a constant expression c. The subscript i is the unique identifier of the currently executing call. The superscript j, if present, indicates that the entity is currently locked by the critical section j. Entity requests can be separated in signals (3-tuples) and calls (4-tuples):

- The first component, an entity key k or an orchestration id d, is the source of the request.
- The second and third components are the operation name o and an input argument c.
- For calls, there is a fourth component i, which is the unique identifier for the request.

4.4.1 Lifecycle. Like virtual actors, entities cannot be created or deleted; but their state can be "uninteresting", if it is the default state () and the queue is empty. We model this by nondeterministic transitions that can add or remove uninteresting entity components.

$$\text{AutoStart}\ \frac{\mathbb{E}_k \notin C \qquad k = (n_E, c) \qquad \sigma = ()}{C \Longrightarrow C\ \mathbb{E}_k(\epsilon, idle_\sigma)} \qquad \text{Collect}\ \frac{\sigma = ()}{C\ \mathbb{E}_k(\epsilon, idle_\sigma) \Longrightarrow C}$$

4.4.2 Local Reduction Steps. To match all the busy execution states with a single syntax, we define

$$b_\sigma\ e ::= busy_{\sigma,d,i}\ e \mid busy_\sigma\ e \mid busy^j_{\sigma,d,i}\ e$$

Then the basic local evaluation steps are covered by:

$$\text{Ent-Local}\ \frac{e \longrightarrow e'}{C\ \mathbb{E}_k(q, b_\sigma\ e) \Rightarrow C\ \mathbb{E}_k(q, b_\sigma\ e')}$$

4.4.3 Accessing Entity State. Application code executing on entities can read or update the state:

$$\text{Ent-Get}\ \frac{\sigma = c}{C\ \mathbb{E}_k(q, b_\sigma\ E[get]) \Rightarrow C\ \mathbb{E}_k(q, b_\sigma\ E[c])}$$

$$\text{Ent-Set}\ \frac{\sigma' = c}{C\ \mathbb{E}_k(q, b_\sigma\ E[set\ c]) \Rightarrow C\ \mathbb{E}_k(q, b_{\sigma'}\ E[()])}$$

4.4.4 Enqueueing Operations. Orchestrations can signal or call an entity operation, which enqueues a request on the left of the queue:

$$\text{Orch-Signal-Ent}\ \frac{e = \mathit{signal}\ n_E.o(c_k, c_i) \qquad k = (n_E, c_k) \qquad r = (d, o, c_i)}{C\ \mathbb{O}_d(b\ E[e])\ \mathbb{E}_k(q, x_E) \Rightarrow C\ \mathbb{O}_d(b\ E[()])\ \mathbb{E}_k(r\ q, x_E)}$$

$$\text{Orch-Call-Ent}\ \frac{e = \mathit{call}\ n_E.o(c_k, c_i) \qquad k = (n_E, c_k) \qquad r = (d, o, c_i, i)}{C\ \mathbb{O}_d(b\ E[e])\ \mathbb{E}_k(q, x_E) \overset{i}{\Longrightarrow} C\ \mathbb{O}_d(b\ E[P_i])\ \mathbb{E}_k(r\ q, x_E)}$$

An entity can also signal another entity, or itself:

$$\text{Ent-Signal-Ent}\ \frac{e = \mathit{signal}\ n_E.o(c_k, c_i) \qquad k' = (n_E, c_k) \qquad r = (k, o, c_i)}{C\ \mathbb{E}_k(q, b_\sigma\ E[e])\ \mathbb{E}_{k'}(q', x_E) \Rightarrow C\ \mathbb{E}_k(q, b_\sigma\ E[()])\ \mathbb{E}_{k'}(r\ q', x_E)}$$

$$\text{Ent-Signal-Self}\ \frac{e = \mathit{signal}\ n_E.o(c_k, c_i) \qquad k = (n_E, c_k) \qquad r = (k, o, c_i)}{C\ \mathbb{E}_k(q, b_\sigma\ E[e]) \Rightarrow C\ \mathbb{E}_k(r\ q, b_\sigma\ E[()])}$$

4.4.5 Dequeueing Operations. An idle entity can take a request (call or signal) from the queue and start executing it. The queue is FIFO-per-origin: only the right-most (oldest) request from a particular origin can be dequeued.

$$\text{Ent-Take-Call}\ \frac{(d, \dots) \notin q' \qquad k = (n_E, c_k) \qquad (n_E.o : f) \in \mathcal{A}}{C\ \mathbb{E}_k(q\ (d, o, c, i)\ q', \mathit{idle}_\sigma) \Rightarrow C\ \mathbb{E}_k(q\ q', \mathit{busy}_{\sigma,d,i}\ (f\ c))}$$

$$\text{Ent-Take-Signal}\ \frac{(y, \dots) \notin q' \qquad k = (n_E, c_k) \qquad (n_E.o : f) \in \mathcal{A}}{C\ \mathbb{E}_k(q\ (y, o, c)\ q', \mathit{idle}_\sigma) \Rightarrow C\ \mathbb{E}_k(q\ q', \mathit{busy}_\sigma\ (f\ c))}$$

4.4.6 Completed Operations. When the request processing has completed, the entity goes back to idle. If the request was a signal, nothing else happens. If the request was a call, the placeholders in the calling orchestration are replaced with the final value.

$$\text{Ent-EndSignal}\ \frac{}{C\ \mathbb{E}_k(q, \mathit{busy}_\sigma\ \rho) \Rightarrow C\ \mathbb{E}_k(q, \mathit{idle}_\sigma)}$$

$$\text{Ent-EndCall}\ \frac{x'_O = x_O[\mathit{done}\ \rho / P_i]}{C\ \mathbb{O}_d(x_O)\ \mathbb{E}_k(q, \mathit{busy}_{\sigma,d,i}\ \rho) \Rightarrow C\ \mathbb{O}_d(x'_O)\ \mathbb{E}_k(q, \mathit{idle}_\sigma)}$$

4.5 External Service Calls

Activities and entities can make external calls, which evaluate to a placeholder, and send a request message to the environment. When a response c^i is received from the environment, it replaces all placeholders in the system.

$$\text{Act-Send}\ \frac{}{C\ \mathbb{A}_{i'}(b\ E[\mathit{call}\ n_X(c)]) \overset{i}{\Longrightarrow} C\ \mathbb{A}_{i'}(b\ E[P_i]) + n_X(c)_i}$$

$$\text{Ent-Send}\ \frac{}{C\ \mathbb{E}_k(q, b_\sigma\ E[\mathit{call}\ n_X(c)]) \overset{i}{\Longrightarrow} C\ \mathbb{E}_k(q, b_\sigma\ E[P_i]) + n_X(c)_i}$$

$$\text{System-Receive}\ \frac{}{C + \rho^i \Rightarrow C[\mathit{done}\ \rho / P_i]}$$

Including external calls in our formalization is interesting, because it allows us to understand the weaker guarantees of subsequent models. Specifically, in a compute-storage separated system, external calls can be duplicated if there are multiple attempts at executing a work item.

4.6 Critical Sections

We extend expressions with a special runtime expression that indicates an active critical section, with a unique identifier j for the critical section:

$$e ::= \dots \mid lock^j \langle k_1 \cdots k_n \rangle\ e$$

4.6.1 Enter and Exit. When entering a critical section, it generates a unique identifier j for it, and locks all the entities specified in the lock expression. Conversely, upon exiting the critical section, it unlocks all the same entities:

$$\text{CS}^j\text{-Enter}\ \frac{}{\begin{array}{ll} & C \quad \mathbb{E}_{k_1}(q_1, idle_{\sigma_1}) \cdots \mathbb{E}_{k_n}(q_n, idle_{\sigma_n}) \quad \mathbb{O}_d(b\ E[lock\langle k_1 \cdots k_n \rangle\ e]) \\ \overset{j}{\Longrightarrow} & C \quad \mathbb{E}_{k_1}(q_1, idle^j_{\sigma_1}) \cdots\ \mathbb{E}_{k_n}(q_n, idle^j_{\sigma_n}) \quad \mathbb{O}_d(b\ E[lock^j\langle k_1 \cdots k_n \rangle\ e]) \end{array}}$$

$$\text{CS}^j\text{-Exit}\ \frac{}{\begin{array}{ll} & C \quad \mathbb{E}_{k_1}(q_1, idle^j_{\sigma_1}) \cdots\ \mathbb{E}_{k_n}(q_n, idle^j_{\sigma_n}) \quad \mathbb{O}_d(b\ E[lock^j\langle k_1 \cdots k_n \rangle\ v]) \\ \Longrightarrow & C \quad \mathbb{E}_{k_1}(q_1, idle_{\sigma_1}) \cdots \mathbb{E}_{k_n}(q_n, idle_{\sigma_n}) \quad \mathbb{O}_d(b\ E[v]) \end{array}}$$

4.6.2 Executing the Critical Section. Once a critical section has been entered, it can do local evaluation steps as below. Note that there is no overlap with earlier execution rules because contexts E do not reach into critical sections.

$$\text{CS}^j\text{-Local}\ \frac{e \longrightarrow e'}{C\ \mathbb{O}_d(b\ E[lock^j\langle \dots \rangle\ e]) \Longrightarrow C\ \mathbb{O}_d(b\ E[lock^j\langle \dots \rangle\ e'])}$$

It can also call locked entities, and receive results when they finish executing:

$$\text{CS}^j\text{-Call-Ent}\ \frac{e = call\ n_E.o(c_k, c_i) \qquad k = (n_E, c_k) \qquad (n_E.o : f) \in \mathcal{A}}{\begin{array}{ll} & C\ \mathbb{O}_d(b\ E[lock^j\langle \dots k \dots \rangle\ E'[e]])\ \mathbb{E}_k(q, idle^j_\sigma) \\ \overset{i}{\Longrightarrow} & C\ \mathbb{O}_d(b\ E[lock^j\langle \dots k \dots \rangle\ E'[P_i]])\ \mathbb{E}_k(q, busy^j_{\sigma,d,i}\ (f\ c)) \end{array}}$$

$$\text{CS}^j\text{-EndCall}\ \frac{}{\begin{array}{ll} & C\ \mathbb{O}_d(b\ E[lock^j\langle \dots k \dots \rangle\ e])\ \mathbb{E}_k(q, busy^j_{\sigma,d,i}\ \rho) \\ \Longrightarrow & C\ \mathbb{O}_d(b\ E[lock^j\langle \dots k \dots \rangle\ e[\rho/i]])\ \mathbb{E}_k(q, idle^j_\sigma) \end{array}}$$

Finally, it can signal non-locked entities:

$$\text{CS}^j\text{-Signal-Ent}\ \frac{e = signal\ n_E.o(c_k, c_i) \qquad k = (n_E, c_k) \qquad k \notin \overline{k} \qquad r = (d, o, c_i)}{\begin{array}{ll} & C\ \mathbb{O}_d(b\ E[lock^j\langle \overline{k} \rangle\ E'[e]])\ \mathbb{E}_k(q, x_E) \\ \Longrightarrow & C\ \mathbb{O}_d(b\ E[lock^j\langle \overline{k} \rangle\ E'[()]])\ \mathbb{E}_k(r\ q, x_E) \end{array}}$$

$\mathcal{S} ::= \epsilon \mid \mathcal{S}\ \kappa\langle\overline{g}, x\rangle$	(storage system state)
$\kappa ::= i \mid d \mid k$	(storage key)
$x ::= \bot \mid x_A \mid x_O \mid x_E$	(execution state)
$g ::=$	(task message)
$\mid k.r$	(request for entity k)
$\mid d.(\rho/i)$	(response for orchestration d)
$\mid d.n_O(c)$	(start orchestration d)
$\mid i.n_A(c, d)$	(start activity i)

Fig. 10. States of the compute-storage model.

5 COMPUTE-STORAGE MODEL

We now show how to implement the high-level operational model in a system where the compute is separated from storage. The idea is to replace the failure-free system described in Section 4 with a more realistic system that is composed by:

(1) A storage system that reliably stores the current state of all orchestrations and entities, and supports an atomic commit operation.
(2) A variable number of stateless workers which fetch work-items from the storage, execute them, and then commit them to storage.

Work-items are an important abstraction in the serverless context, since they act as the unit of billing and application progress. The nature of a work-item depends on the type of function:

- For activities, a work-item is a single, complete execution of the activity function.
- For entities, a work item consists of executing a batch of queued operations to completion.
- For orchestrations, a work-item starts a new execution, or processes responses to a waiting execution. It continues execution until the code completes or gets stuck waiting on a response.

Since workers stop the execution of waiting orchestrations, DF avoids the double-billing issue [Baldini et al. 2017], where users have to pay for the execution of both the caller and the callee while the caller waits on the callee to return a result.

5.0.1 Organization. We start by describing the storage system and its transitions in Section 5.1. We then describe workers (Section 5.2) and the transitions that describe work-item processing (Appendix A.1). Finally, in Section 5.3 we define what it means for the compute-storage system to be correct with respect to the high-level system, and we prove that it is satisfied.

5.0.2 Future Extensions. The compute-storage model currently does not include external calls or critical sections. Adding external calls to the formalism is technically easy, but requires an alternate formulation of the correctness guarantees, because duplication of external calls (unlike internal calls) is observable, and can happen when workers make repeated attempts at processing a work item. Adding critical sections requires inclusion of the distributed locking protocol employed by DF. Both of these extensions are interesting avenues for future study.

5.1 Storage System

The storage system is organized like a key-value store with a queue attached to each entry (Fig. 10). Each key κ is either an activity id i, and instance id d, or an entity key k. The store contains tuples of the form $\kappa\langle\overline{g}, x\rangle$, where x is the execution state, and $\overline{g}$ is a queue of *task messages*. The task messages represent "work" that needs to be processed for a given key.

5.1.1 Enqueueing of Task Messages. First, we define a helper function $enq(\overline{g}, \mathcal{S})$ that distributes a sequence of task messages $\overline{g}$ into their respective destination queues. Messages are enqueued on the left. If a message targets a non-existent entry, a new one is added with empty queue ϵ and undefined execution state $\perp$.

$$\text{Enq-New}\ \frac{g = \kappa.\_ \qquad \kappa \notin \text{keys}(\mathcal{S})}{enq(g, \mathcal{S}) = \mathcal{S}\ \kappa\langle g, \perp\rangle} \qquad \text{Enq-Existing}\ \frac{g = \kappa.\_}{enq(g, \mathcal{S}\ \kappa\langle \overline{g}, x\rangle) = \mathcal{S}\ \kappa\langle g\ \overline{g}, x\rangle}$$

$$\text{Enq-Empty}\ \frac{}{enq(\epsilon, \mathcal{S}) = \mathcal{S}} \qquad \text{Enq-Multiple}\ \frac{enq(\overline{g_1}, \mathcal{S}) = \mathcal{S}' \qquad enq(g_2, \mathcal{S}') = \mathcal{S}''}{enq(g_2\ \overline{g_1}, \mathcal{S})) = \mathcal{S}''}$$

5.1.2 External Application Requests. Orchestrations are started the same way as in the high-level model, but instead of starting the orchestration directly, a start message is added to the queue.

$$\text{S-StartNew-Fresh}\ \frac{d \notin \text{keys}(\mathcal{S}) \qquad g = d.n_O(c)}{\mathcal{S} + startnew(d, n_O, c)_i \Rightarrow \mathcal{S}\ d\langle g, \perp\rangle + ok^i}$$

$$\text{S-StartNew-Conflict}\ \frac{d \in \text{keys}(\mathcal{S})}{\mathcal{S} + startnew(d, n_O, c)_i \Rightarrow \mathcal{S} + alreadyexists^i}$$

5.1.3 Commit. Workers continuously look for work items, i.e. entries $\kappa\langle \overline{g_{in}}, x_{pre}\rangle$ for which $\overline{g_{in}}$ is not empty. After a worker processes such an item (as defined in §5.2), it tries to atomically commit the new execution state x_{post}, as well as all task messages $\overline{g_{out}}$ that were produced by the execution. We model this with a message $commit(\kappa, \overline{g_{in}}, x_{pre}, x_{post}, \overline{g_{out}})$. The rule below shows how, and under what conditions, the commit is accepted and applied.

$$\text{S-Commit}\ \frac{}{\mathcal{S}\ \kappa\langle \overline{g_{new}}\ \overline{g_{in}}, x_{pre}\rangle + commit(\kappa, \overline{g_{in}}, x_{pre}, x_{post}, \overline{g_{out}})_i \Rightarrow enq(\overline{g_{out}}, \mathcal{S}\ \kappa\langle \overline{g_{new}}, x_{post}\rangle) + ok^i}$$

There are several interesting observations about the (S-Commit) rule:

- If the queue has received new messages $\overline{g_{new}}$ in the meantime, those remain in the queue after the commit.
- The commit happens only if the original $\overline{g_{in}}$ and x_{pre} match. This ensures that even if multiple workers attempt to execute the same work item, only one worker can commit it. This mechanism is conceptually similar to compare-and-swap in shared-memory multiprocessors.
- The produced messages $\overline{g_{out}}$ are distributed over all the queues, possibly including the queue for the same κ.

5.2 Work-Item Processing

We now describe workers and how they execute work-items. The execution steps are defined in the appendix (§A.1). They fall into in three categories:

x	$\rightarrow_\kappa$	x'	(update execution state x to x')
$x + g$	$\rightarrow_\kappa$	x'	(receive message g and update execution state x to x')
x	$\rightarrow_\kappa$	$x' + g$	(update state x to x', send message g)

Based on these, we then formulate a worker transition system, where the worker state is a triple $\overline{g_{in}} \mid x \mid \overline{g_{out}}$ of an input buffer, an execution state, and an output buffer:

$$\text{WIn}\ \frac{g + x \longrightarrow_\kappa x'}{\overline{g_{in}}\, g \mid x \mid \overline{g_{out}} \ \longrightarrow_\kappa\ \overline{g_{in}} \mid x' \mid \overline{g_{out}}} \qquad \text{WOut}\ \frac{\text{not (WIn)} \qquad x \longrightarrow_\kappa x' + g}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \ \longrightarrow_\kappa\ \overline{g_{in}} \mid x' \mid g\, \overline{g_{out}}}$$

$$\text{WStep}\ \frac{\text{not (WIn)} \qquad x \longrightarrow_\kappa x'}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \ \longrightarrow_\kappa\ \overline{g_{in}} \mid x' \mid \overline{g_{out}}}$$

Note that workers apply inputs more eagerly than they take other steps; this is indicated by the precondition "not (WIn)" in (WOut) and (WStep), meaning that the latter apply only if (WIn) does not. We now formalize the idea that workers execute work items *as long as possible*; to completion or until the execution blocks.

Definition 5.1. Define the big-step *execution relation* $\boxed{\overline{g_{in}} \mid x \twoheadrightarrow_\kappa x' \mid \overline{g_{out}}}$ to represent a sequence $\overline{g_{in}} \mid x \mid \epsilon \longrightarrow_\kappa^* \epsilon \mid x' \mid \overline{g_{out}}$ of zero or more $\longrightarrow_\kappa$ steps such that the final execution state x' is of the form $(completed\ \rho)$, $(b\ E(p))$, or $(idle_\sigma)$.

Workers only send commit messages to the storage system when they execute a work-item to completion. Formally:

Definition 5.2. Given some application $\mathcal{A}$, any commit message $commit(\kappa, \overline{g_{in}}, x_{pre}, x_{post}, \overline{g_{out}})$ sent by a worker satisfies $\overline{g_{in}} \mid x_{pre} \twoheadrightarrow_\kappa x_{post} \mid \overline{g_{out}}$.

5.3 Execution Guarantees under Faults

As in any system with compute-storage separation, we consider the state of the workers, and the communication between the workers and the storage system, to be volatile. This means that is possible for workers to fail mid-execution. Since it is important for workflows to never stall, we need new workers to pick up and retry any stalled work items. Unfortunately, since it is in general impossible to reliably detect failure of a previous worker, this means that multiple workers may be concurrently executing the same work item. Nevertheless, the strongly consistent storage system ensures that they can be committed only once. This means that the internal state of the system is not exposed to any faults or duplication effects!

We now state and prove the main result of this section: all executions of the compute-storage model correspond to an execution of the high-level model. To this end, we define a simulation relation $\mathcal{S} \sim C$ between the respective states (Fig. 11). The correspondence is defined individually for each key. For entities, the correspondence is straightforward: both the queue and the execution states have to match (sim-ent). For activities or orchestrations, however, there are no queues in the high-level model; thus we define the simulation on the state obtained after applying all effects in the

$$\boxed{\mathcal{S} \sim C} \qquad \text{sim-empty}\ \frac{}{\epsilon \sim \epsilon} \qquad \text{sim-ind}\ \frac{\mathcal{S} \sim C \qquad \kappa \notin \text{keys}(\mathcal{S}) \qquad k\langle \overline{g}, x\rangle \sim \mathbb{C}}{\mathcal{S}\ k\langle \overline{g}, x\rangle \sim C\ \mathbb{C}}$$

$$\boxed{\kappa\langle \overline{g}, x\rangle \sim \mathbb{C}} \qquad \text{sim-ent}\ \frac{\forall i : g_i = k.r_i}{k\langle g_1 \ldots g_n, x\rangle \sim \mathbb{E}_k(r_1 \ldots r_n, x)}$$

$$\text{sim-act}\ \frac{}{i\langle g_n \ldots g_1, x_1\rangle \sim \mathbb{A}_i((g_n \cdots g_1)(x_1))} \qquad \text{sim-orc}\ \frac{}{d\langle g_n \ldots g_1, x_1\rangle \sim \mathbb{O}_d((g_n \cdots g_1)(x_1))}$$

Fig. 11. Definition of the relation $\mathcal{S} \sim C$ for a compute-storage state $\mathcal{S}$ and a high-level state $\mathbb{C}$.

queue (sim-act), (sim-orc); where the notation $(g_n \cdots g_1)(x_1) = x_n$ means that the effects $g_n \cdots g_1$ transform x_1 to x_n, that is, that there exist a κ and execution states x_i such that $g_i + x_i \rightarrow_\kappa x_{i+1}$.

The following theorem implies that the compute-storage model is a correct implementation of the high-level model: that is, for all executions of the implementation there exists an observationally equivalent execution of the specification. It states that the specification simulates the implementation, that is, the specification can mimic (and thus validate) all of the behaviors of the implementation.

Theorem 5.3. *$\sim$ is a weak simulation relation, when hiding commit messages.*

To prove this, we add small-step transitions, prove the weak simulation on the small-step transitions, and then show that the big-step commit transition are just sequences of small-step transitions. The proof details are in the appendix B, which is included in the full version of this paper.

6 REPLAY-BASED MODEL

We have shown how to implement the idealized high level model using a combination of a storage system and a number of stateless workers. However, we have not explained how to store the intermediate execution states x_O of orchestrations. The state and progress of an orchestration that is written in a mainstream programming language is not serializable by default, e.g., as is the case for lambda expressions. Rather, the state of the orchestration can be dispersed, including variables and the execution location, as well as arbitrary non-serializable objects on the heap, such as promises and tasks, whose representation is managed by the runtime or by other libraries. To make matters worse, state may even be external to the application heap, for example when importing a linear algebra library that is implemented in C. While it may be possible to guarantee serializability and provide checkpointing in some circumstances, such as by using a specialized type system [Miller et al. 2014], it is not something we can expect to be available in all the host languages supported by DF.

To address this, Durable Functions provides a replay-based model that stores orchestration execution states as histories of events. Given that the orchestration code is deterministic, the runtime can then replay the history, rehydrating the state of the orchestration in memory. This approach has two main benefits. First, it supports most mainstream programming languages without requiring special treatment, since the history captures execution state in a language agnostic manner. Second, saving the history improves debugging and observability, since the users can later inspect not only the final execution state of the application, but also all of its intermediate steps. On the other hand, record-replay means that nondeterminism or large histories can be a problem for orchestrations that do not follow the guidelines (§3.5).

In this section, we extend the compute storage system introduced in Section 5 with histories, and we prove that the extended system is bisimilar to the original, establishing (i) that persisting histories is equivalent to persisting intermediate states, and (ii) that this model is a correct implementation of the high-level model.

6.1 History Storage

The storage system remains largely unchanged, except that we replace execution states x with histories h, where a history is simply a sequence of incoming messages $in(g)$ and outgoing messages $out(g)$. Also, we extend the task messages g with a new message $k.\sigma$, which represents initializing the state of entity k to σ.

$$\boxed{g \mid x \mid g \mid h \mathrel{\circ\!\!\longrightarrow_\kappa} g \mid x \mid g \mid h} \qquad \text{RIn}\;\frac{g + x \longrightarrow_\kappa x'}{\overline{g_{in}}\; g \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_\kappa} \overline{g_{in}} \mid x' \mid \overline{g_{out}} \mid h_{out}\; in(g)}$$

$$\text{ROut}\;\frac{\text{not (RIn)} \qquad x \longrightarrow_\kappa x' + g}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_\kappa} \overline{g_{in}} \mid x' \mid g\; \overline{g_{out}} \mid h_{out}\; out(g)}$$

$$\text{RStep-Cont}\;\frac{\text{not (RIn)} \qquad x = b\; E[continue\; n_O(c)] \qquad (n_O : f) \in \mathcal{A} \qquad x' = busy\;(f\; c)}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_d} \overline{g_{in}} \mid x' \mid \overline{g_{out}} \mid in(d.n_O(c))}$$

$$\text{RStep-DoneSignal}\;\frac{\text{not (RIn)} \qquad x = busy_\sigma\; \rho \qquad x' = idle_\sigma}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_k} \overline{g_{in}} \mid x' \mid \overline{g_{out}} \mid in(k.\sigma)}$$

$$\text{RStep-DoneCall}\;\frac{\text{not (RIn)} \qquad x = busy_{\sigma,d,i}\; \rho \qquad x' = idle_\sigma}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_k} \overline{g_{in}} \mid x' \mid d.(\rho/i)\; \overline{g_{out}} \mid in(k.\sigma)}$$

$$\text{RStep}\;\frac{\text{not (RIn) or (RStep-\_)} \qquad x \longrightarrow_\kappa x'}{\overline{g_{in}} \mid x \mid \overline{g_{out}} \mid h_{out} \mathrel{\circ\!\!\longrightarrow_\kappa} \overline{g_{in}} \mid x' \mid \overline{g_{out}} \mid h_{out}}$$

Fig. 12. Recording Transitions.

$$\boxed{h \mathrel{\triangleright\!\!\longrightarrow_\kappa} x} \qquad \text{YIn}\;\frac{g + x \longrightarrow_\kappa x'}{in(g)\; h_{in} \mid x \mathrel{\triangleright\!\!\longrightarrow_\kappa} h_{in} \mid x'} \qquad \text{YOut}\;\frac{\text{not (YIn)} \qquad x \longrightarrow_\kappa x' + g}{out(g)\; h_{in} \mid x \mathrel{\triangleright\!\!\longrightarrow_\kappa} h_{in} \mid x'}$$

$$\text{YStep}\;\frac{\text{not (YIn)} \qquad x \longrightarrow_\kappa x'}{h_{in} \mid x \mathrel{\triangleright\!\!\longrightarrow_\kappa} h_{in} \mid x'}$$

Fig. 13. Replay Transitions.

$\mathcal{S} ::= \epsilon \mid \mathcal{S}\; \kappa\langle q, h \rangle$	storage system state
$w ::= in(g) \mid out(g)$	history entry
$h ::= \overline{w}$	history
$g ::= \cdots \mid k.\sigma$	(init state of entity k)

All the rules for the storage system are straightforwardly adapted by replacing x with h. A commit message now has the form $commit(\kappa, \overline{g_{in}}, h_{in}, h_{out}, \overline{g_{out}})$.

6.2 Worker Semantics

When a worker fetches a work item, it has to first replay the history. The point of this "replay execution" is to "rehydrate" the execution state. Any other effects are suppressed: for example, messages that were already sent by the original execution are not sent again during replay. When replay is complete, "recording execution" starts. Effects are normally applied, and also recorded into the history to enable future replay. Record and replay are defined in Fig. 12 and 13, respectively.

Definition 6.1. Define the big-step *recording execution relation* $\boxed{\overline{g_{in}} \mid x \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow_\kappa} x' \mid \overline{g_{out}} \mid h_{out}}$ to represent a sequence $\overline{g_{in}} \mid x \mid \epsilon \mid h_{in} \mathrel{\circ\!\!\longrightarrow_\kappa^*} \epsilon \mid x' \mid \overline{g_{out}} \mid h_{out}$ such that the final execution state x' is of the form $(completed\; \rho)$, $(b\; E(p))$, or $(idle_\sigma)$.

Definition 6.2. Define the big-step *replay execution relation* $\boxed{h \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x}$ to represent a sequence $h \mid \bot \mathrel{\rhd\!\!\longrightarrow}^*_\kappa \epsilon \mid x'$ where the final execution state x' is of the form $(\mathit{completed}\ \rho)$, $(b\ E(p))$, or (idle_σ).

6.2.1 *Worker Semantics.* We now define the worker semantics by combining replay and record:

Definition 6.3. Given some application $\mathcal{A}$, any message $\mathit{commit}(\kappa, \overline{g_{in}}, h_{in}, h_{out}, \overline{g_{out}})$ sent by a worker means that there exist execution states x_{pre} and x_{post} such that $h_{in} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x_{pre}$ and $\overline{g_{in}} \mid x_{pre} \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow}_\kappa x_{post} \mid \overline{g_{out}} \mid h_{out}$.

6.3 Observational Equivalence

We now show that the original storage system (§5.1) and the replay-based storage system (§6.1) are observationally equivalent, by proving that they are bisimilar. We are proving bisimulation because we are interested in both directions, namely, that persisting histories is equivalent to persisting intermediate states. In contrast, in Theorem 5.3 we use a simulation because we are interested in one of the directions, i.e., that any behavior of the implementation (compute storage model) is equivalent to a behavior of the specification (high-level model).

First, consider again the relation $(h \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x)$. It means that the history h replays to the execution state x. We now extend this relation so it applies to the whole storage system configurations defined by (§5.1) and (§6.1):

$$\frac{}{\epsilon \sim \epsilon} \qquad\qquad \frac{h \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x \qquad \mathcal{S} \sim \mathcal{S}'}{\mathcal{S}\ \kappa\langle q, h\rangle \ \sim\ \mathcal{S}'\ \kappa\langle q, x\rangle}$$

Theorem 6.4. *$\sim$ is a bisimulation.*

Proof. As the transition systems are constructed identically, save for swapping the h and x components, the proof is boilerplate except for the transition (S-Commit), where we need to show that the two workers from definitions 5.2 and 6.3 can simulate each other. Starting with the simulation relation established, and thus $(h_{in} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x_{pre})$,

- if the non-history-worker does $\mathit{commit}(\kappa, \overline{g_{in}}, x_{pre}, x_{post}, \overline{g_{out}})$ then $\overline{g_{in}} \mid x_{pre} \twoheadrightarrow_\kappa x_{post} \mid \overline{g_{out}}$, then by Lemma 6.7(2) we get $\overline{g_{in}} \mid x_{pre} \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow}_\kappa x_{post} \mid \overline{g_{out}} \mid h_{out}$, thus the history worker can do $\mathit{commit}(\kappa, \overline{g_{in}}, h_{in}, h_{out}, \overline{g_{out}})$, and by Lemma 6.6 we get $h_{out} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x_{post}$, establishing the simulation relation again.
- if the history worker does $\mathit{commit}(\kappa, \overline{g_{in}}, h_{in}, h_{out}, \overline{g_{out}})$, then $(h_{in} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x)$ and $\overline{g_{in}} \mid x \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow}_\kappa x' \mid \overline{g_{out}} \mid h_{out}$ for some x, x'. By Lemma 6.7(1), $\overline{g_{in}} \mid x \twoheadrightarrow_\kappa x' \mid \overline{g_{out}}$. By Lemma 6.5, $x = x_{pre}$. Thus the non-history worker can do $\mathit{commit}(\kappa, \overline{g_{in}}, x_{pre}, x', \overline{g_{out}})$. And by Lemma 6.6 we get $h_{out} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x'$, establishing the simulation relation again.

□

Proofs for all the lemmas below can be found in the appendix B, which is included in the full version of this paper. Note that Lemma 6.5 relies on the determinism of the execution, which in this model, is easy to prove for our simple lambda calculus. In a mainstream programming language, nondeterminism is not implicitly guaranteed, and requires programmers to be careful when writing orchestrations (§3.5).

Lemma 6.5 (Deterministic Replay). *If $h \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x$ and $h \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x'$, then $x = x'$.*

Lemma 6.6 (Record/Replay). *Given a recording execution $\overline{g_{in}} \mid x \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow}_\kappa x' \mid \overline{g_{out}} \mid h_{out}$ such that $h_{in} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x$, then $h_{out} \mathrel{\rhd\!\!\twoheadrightarrow}_\kappa x'$.*

Lemma 6.7 (Transparency of Recording). *Both of the following are true:*

(1) *If $\overline{g_{in}} \mid x \mid h_{in} \mathrel{\circ\!\!\twoheadrightarrow}_\kappa x' \mid \overline{g_{out}} \mid h_{out}$ is a recording execution, then $\overline{g_{in}} \mid x \twoheadrightarrow_\kappa x' \mid \overline{g_{out}}$ is a regular execution.*

(2) *If* $\overline{g_{in}} \mid x \twoheadrightarrow_{\kappa} x' \mid \overline{g_{out}}$ *is an execution, and* h_{in} *is a history, then* $\overline{g_{in}} \mid x \mid h_{in} \circ\!\!\twoheadrightarrow_{\kappa} x' \mid \overline{g_{out}} \mid h_{out}$ *is a recording execution for some* h_{out}.

7 RELATED WORK

Formal Semantics for Serverless. [Jangda et al. 2019] pioneered the study of serverless semantics, presenting a formal model for FaaS and explaining its limitations. They also show how to compose functions using a proposed language called SPL and how to create stateful applications by using an external key-value store and transactions. While similar in spirit, our formal development is focused on DF (a stateful serverless programming framework by a major cloud provider) which uses a different and more comprehensive set of abstractions (see the discussion in the next section).

Stateful Serverless Abstractions. There has been a recent surge of proposals for stateful serverless programming models that provide abstractions for developing serverless applications that maintain state from both academia [Fouladi et al. 2019, 2017; Jonas et al. 2017; Sreekanti et al. 2020a,b; Zhang et al. 2020b] and industry [Amazon 2020; Bonér 2020; CloudFlare 2020a,b]. Most of these systems do not provide strong reliability guarantees, but still require programmers to detect and handle transient storage errors. Also, none of them includes formal semantics or proofs.

[Zhang et al. 2020a] propose a runtime for serverless workflows that guarantees exactly-once execution in the presence of faults for applications whose state is saved in a key-value store. This is achieved by instrumenting storage accesses to enable record/replay.

DF's tasks, actors, and critical sections provide a more expressive model than key-value stores with transactions, which are used by [Zhang et al. 2020a] and [Jangda et al. 2019]. Ordered queues, for example, cannot be efficiently represented by the latter. Also, optimistic transactions (unlike critical sections or entities) expose transient failures to the application, rather than hiding them, and perform poorly under congestion.

Kappa [Zhang et al. 2020b] is similarly expressive as DF, as it offers both task parallelism and blocking queues (with which actors and critical sections can be expressed). Kappa's design differs from DF in several aspects; the decomposition into steps and tasks is semi-implicit (with some boundaries inferred, others annotated), rather than explicit as in DF. And application progress is persisted by an automatic checkpointing implementation using Python's pickling feature, rather than based on record/replay as in DF.

Reliable Programming Models. Much prior work has studied programming models that can execute reliably in a distributed environment, hiding the presence of faults. These models are usually focused on a specific application domain, proposing highly specialized solutions. A popular such domain is data processing, some examples include MapReduce [Dean and Ghemawat 2008], Apache Spark [Zaharia et al. 2012], and Apache Flink [Carbone et al. 2015]. Recently, AMBROSIA [Goldstein et al. 2020] proposed "immortal" actors, a reliable actor abstraction for distributed application development. Recent work on λ_{FAIL} [Ramalingam and Vaswani 2013] also proposes a semantics for distributed services that execute on top of reliable storage, also providing a compilation procedure using monads that guarantees correct execution in the presence of faults. DF is more general as it supports both task- and actor parallelism, and can save intermediate execution states, even for mainstream programming languages, via record/replay.

DF Implementations. DF is based on a precursor project called Durable Task Framework (DTFx) [github [n.d.]]. DTFx contains the work-item and record-replay abstractions but is not by itself serverless. DTFx also defines the notion of an *orchestration service*, which allows different storage backends to be used. Multiple such backends are available on github, and use slightly different methods for storing the application state and atomically committing work items. The

DurableTask.SqlServer backend [github 2021a] keeps all state in a relational database and uses SQL transactions for atomically committing work items. The *DurableTask.Netherite* backend [github 2021b] uses static partitions that can atomically commit work items using a per-partition commit log, and communicate via ordered persistent queues [Burckhardt et al. 2021].

The models we presented here are not meant to describe real implementations but are rather focused on two implementation problems that are inherent to any stateful serverless solution: (i) the unreliability of compute workers and (ii) the lack of checkpointing support for arbitrary programming languages. Nevertheless, our compute storage model (§5) and the SqlServer backend [github 2021a] are actually very similar, as they both achieve reliable execution using an atomic commit primitive.

8 CONCLUSION AND FUTURE WORK

We have studied DF from two viewpoints. First, we showed how DF enables stateful serverless patterns informally (with examples) and formally (using an idealized high-level model). Second, we defined two more models and proved them equivalent. Thereby, we demystified how (i) DF can simulate reliable execution on unreliable serverless workers, and (ii) how DF can persist execution progress without checkpoints. In future work, we plan to complete the compute-storage model (§5.0.2), and to formalize and prove the CCC guarantee of the sharded Netherite implementation [Burckhardt et al. 2021].

A DETAILS FOR THE COMPUTE-STORAGE MODEL

A.1 Execution Transition Rules

A.1.1 *Starting Execution.*

$$\text{W-StartAct}\ \frac{(n_A : f) \in \mathcal{A}}{i.n_A(c, d) + \bot \rightarrow_i busy_d\ f\ c} \qquad \text{W-StartOrc}\ \frac{(n_O : f) \in \mathcal{A}}{d.n_O(c) + \bot \rightarrow_d busy\ f\ c}$$

$$\text{W-StartOp1}\ \frac{k = (n_E, c) \qquad (n_E.o : f) \in \mathcal{A}}{k.(d, o, c, i) + idle_\sigma \rightarrow_k busy_{\sigma,d,i}\ f\ c}$$

$$\text{W-StartOp2}\ \frac{k = (n_E, c) \qquad (n_E.o : f) \in \mathcal{A}}{k.(y, o, c) + idle_\sigma \rightarrow_k busy_\sigma\ f\ c} \qquad \text{W-AutoStart}\ \frac{k = (n_E, c) \qquad k.r + idle_{()} \rightarrow_k x}{k.r + \bot \rightarrow_k x}$$

A.1.2 *Completing Execution.*

$$\text{W-DoneAct}\ \frac{}{busy_d\ \rho \rightarrow_i completed\ \rho + d.[\rho/i]} \qquad \text{W-DoneSignal}\ \frac{}{busy_\sigma\ \rho \rightarrow_k idle_\sigma}$$

$$\text{W-DoneCall}\ \frac{}{busy_{\sigma,d,i}\ \rho \rightarrow_k idle_\sigma + d.[\rho/i]} \qquad \text{W-Cont}\ \frac{(n_O : f) \in \mathcal{A}}{busy\ E[continue\ n_O(c)] \rightarrow_d busy\ (f\ c)}$$

A.1.3 *Local steps.*

$$\text{W-ActStep}\ \frac{e \rightarrow e'}{b\ e \rightarrow_i b\ e'} \qquad \text{W-OrcStep}\ \frac{e \rightarrow e'}{b\ e \rightarrow_d b\ e'} \qquad \text{W-EntStep}\ \frac{e \rightarrow e'}{b_\sigma\ e \rightarrow_k b_\sigma\ e'}$$

$$\text{W-OpGet}\ \frac{}{b_\sigma\ E[get] \rightarrow_k b_\sigma\ E[\sigma]} \qquad \text{W-OpSet}\ \frac{\sigma' = c}{b_\sigma\ E[set\ c] \rightarrow_k b_{\sigma'}\ E[()]}$$

A.1.4 Sending Calls and Signals.

$$\text{W-Call1}\;\frac{(n_A\!:\!f) \in \mathcal{A} \qquad i \text{ fresh}}{b\; E[\mathit{call}\; n_A(c)] \rightarrow_d b\; E[P_i] + i.n_A(c,d)}$$

$$\text{W-Call3}\;\frac{k = (n_E, c_k) \qquad (n_E.o\!:\!f) \in \mathcal{A} \qquad i \text{ fresh}}{b\; E[\mathit{call}\; n_E.o(c_k, c)] \rightarrow_d b\; E[P_i] + k.(d,o,c,i)}$$

$$\text{W-Sig1}\;\frac{k = (n_E, c_k) \qquad (n_E.o\!:\!f) \in \mathcal{A}}{b\; E[\mathit{signal}\; n_E.o(c_k, c)] \rightarrow_d b\; E[()] + k.(d,o,c)}$$

$$\text{W-Sig2}\;\frac{k' = (n_E, c_k)}{b_\sigma\; E[\mathit{signal}\; n_E.o(c_k, c)] \rightarrow_k b_\sigma\; E[()] + k'.(k,o,c)}$$

A.1.5 Receiving Responses.

$$\text{W-OrchResp}\;\frac{}{d.(c/i) + b\; e \rightarrow_d b\; e[\mathit{done}\; c/P_i]}$$

REFERENCES

Amazon 2020. AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.

Joe Armstrong. 1997. The development of Erlang. In *ICFP*, Vol. 97. 196–203.

Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 89–103. https://doi.org/10.1145/3133850.3133855

Jonas Bonér. 2020. Towards Stateful Serverless. https://www.youtube.com/watch?v=DVTf5WQlgB8.

Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Serverless Workflows with Durable Functions and Netherite. arXiv:2103.00033 [cs.DC]

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.

Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

Chris Gillum 2021. Note on usage of yield for async/await. https://twitter.com/cgillum/status/1378128701317992451. Accessed: 2021-04-13.

CloudFlare 2020a. Using Durable Objects, Cloudflare Docs. https://developers.cloudflare.com/workers/learning/using-durable-objects.

CloudFlare 2020b. Workers Durable Objects Beta: A New Approach to Stateful Serverless. https://blog.cloudflare.com/introducing-workers-durable-objects/.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

C. Flanagan and M. Felleisen. 1995. The Semantics of Future and Its Use in Program Optimization. In *Rice University*. 209–220.

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.

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.

Pedro Garcia Lopez, Marc Sanchez-Artigas, Gerard Paris, Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (Dec 2018). https://doi.org/10.1109/ucc-companion.2018.00049

github. [n.d.]. Durable Task Framework. https://github.com/Azure/durabletask.

github. 2021a. DurableTask.MSSQL Backend. https://github.com/microsoft/durabletask-mssql.

github. 2021b. DurableTask.Netherite Backend. https://github.com/microsoft/durabletask-netherite.

Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee 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

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.

Robert H. Halstead. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, OOPSLA (2019).

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.

Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2020. Triggerflow: trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 3–14.

Microsoft 2020. What are Durable Functions? - Microsoft Azure. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

Microsoft Azure 2020. Azure Logic Apps Service. https://azure.microsoft.com/en-us/services/logic-apps/.

Microsoft Azure 2021. Azure SQL. https://azure.microsoft.com/en-us/services/azure-sql/. Accessed: 2021-04-13.

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

Luc Moreau. 1996. The Semantics of Scheme with Future. In *In In ACM SIGPLAN International Conference on Functional Programming (ICFP'96*. 146–156.

Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 115–130.

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.

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault tolerance via idempotence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 249–262.

German Shegalov, Michael Gillmann, and Gerhard Weikum. 2001. XML-enabled workflow management for e-services across heterogeneous platforms. *The VLDB Journal* 10, 1 (2001), 91–103.

Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. 2020a. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020b. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

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

Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020b. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277