Kappa: A Programming Framework for Serverless Computing

Wen Zhang
UC Berkeley
zhangwen@cs.berkeley.edu

Vivian Fang
UC Berkeley
vivian@cs.berkeley.edu

Aurojit Panda
NYU
apanda@cs.nyu.edu

Scott Shenker
UC Berkeley/ICSI
shenker@icsi.berkeley.edu

Abstract

Serverless computing has recently emerged as a new paradigm for running software on the cloud. In this paradigm, programs need to be expressed as a set of short-lived tasks, each of which can complete within a short bounded time (e.g., 15 minutes on AWS Lambda). Serverless computing is beneficial to cloud providers—by allowing them to better utilize resources—and to users—by simplifying management and enabling greater elasticity. However, developing applications to run in this environment is challenging, requiring users to appropriately partition their code, develop new coordination mechanisms, and deal with failure recovery. In this paper, we propose Kappa, a framework that simplifies serverless development. It uses checkpointing to handle lambda function timeouts, and provides concurrency mechanisms that enable parallel computation and coordination.

CCS Concepts

• **Computer systems organization** → **Cloud computing**.

Keywords

Serverless, distributed computing

ACM Reference Format:
Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3419111.3421277

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421277

1 Introduction

Serverless computing is a new cloud paradigm where, instead of provisioning virtual machines (VMs), tenants register *event handlers* (e.g., Python functions) with the platform. When an event occurs, the platform invokes the handler on a *lambda function*, a short-lived, stateless execution environment. A lambda function can execute for a bounded duration (e.g., 15 min on AWS) before being terminated.

Serverless computing benefits both cloud tenants and providers. Tenants no longer need to provision or scale their VMs, and enjoy greater elasticity from fast lambda boot times (100s of ms on AWS). Providers can periodically terminate jobs running on lambda functions (and place the next invocation elsewhere if desired), improving datacenter utilization.

Due to these benefits, serverless computing is being offered by all major cloud providers [7, 32, 38, 55] and rapidly adopted by tenants [23], mainly for event-driven workloads (e.g., file transcoders). Recent work, however, has explored more general applications on serverless including video processing [11, 27], numerical computing [40, 81], and analytics [44, 70, 76]. These examples indicate significant interest in exploiting serverless computing beyond event handling, presumably because of its elasticity and ease of deployment.

Despite this interest, developing general-purpose parallel applications on today's serverless platforms remains difficult due to two main challenges: (1) programmers must manually partition their computation to fit within the lambda function time limit; and (2) programmers have no concurrency or synchronization primitives at their disposal, and so must either implement such primitives, restrict themselves to use share-nothing parallelism, or eschew the use of parallel lambdas. Although several *serverless frameworks* [26, 27, 40] have been developed to simplify development, they fail to address Challenge 1, and they address Challenge 2 by requiring the application to be expressed in a special form that deviates from ordinary programming (e.g., as a state machine).

This paper presents Kappa, a programming framework for general-purpose, parallel severless applications. Kappa aims

to make serverless development as close to ordinary parallel programming as possible—a programmer can write ordinary Python code using a familiar concurrency API (e.g., tasks and futures), and Kappa runs the code on an unmodified serverless platform like AWS Lambda using parallel lambda functions. Kappa offers three main features:

Checkpointing. To run long tasks on time-bounded lambdas, Kappa checkpoints program state periodically and restores from this checkpoint upon lambda function timeout. Our continuation-based checkpointing mechanism (§ 3.2) operates in user mode and requires no modifications to the serverless platform.

Concurrency API. To program parallel lambdas, Kappa provides a concurrency API that supports spawning tasks, waiting on futures, and passing messages between lambdas (§ 3.3). This API is modeled after Python's built-in multiprocessing package and should be familiar to programmers.

Fault tolerance. Kappa tasks can exhibit nondeterminism and side effects. Using checkpoints, Kappa ensures that execution never diverges due to nondeterminism, and that any side effects invoked within the system are never re-executed in face of arbitrary lambda function timeouts (§ 3.1).

Kappa requires no changes to the platform, thus allowing general applications to run on existing serverless offerings. Producing such a framework is the main contribution of our work. While similar techniques have been used in other contexts (§ 7.2), our main insight lies in recognizing that these techniques enable more general use of serverless computing.

We implemented five applications using Kappa (§ 5.3), ranging from bulk-synchronous workloads (e.g., MapReduce) to ones with more complex concurrency patterns (e.g., distributed web crawling). We discuss our limitations in § 6.

We have open-sourced Kappa. The code and sample applications can be found at https://github.com/NetSys/kappa, and documentation at https://kappa.cs.berkeley.edu.

2 Background and Motivation

Many recent papers [11, 26, 27, 40, 43, 70, 81] have explored using serverless for tasks beyond event handling. A main motivation is that lambda functions boot much faster than VMs[1], allowing tenants to quickly launch many compute cores without provisioning a long-running cluster. Serverless platforms thus provide a scalable computation substrate where the amount of computational resources available to a task can be rapidly altered. Prior work has exploited this flexibility for video processing [11, 27], numerical computation [15, 25, 33, 34, 40, 81, 96], data analytics [43, 44, 70, 76], machine learning [18, 39, 83], and parallel compilation [26].

[1]For example, we found that AWS Lambda functions launch at least 30× faster than Amazon EC2 c4.4xlarge VMs with similar compute capacity.

However, given that serverless computing originally targeted event-driven workloads, does it makes sense to use serverless for general applications? In our work, we found that the many benefits of serverless carry over naturally to general computing—the cloud provider continues to enjoy improved statistical multiplexing from the short-livedness of serverless functions, and the tenant, fast task startup and freedom from managing infrastructure [41]. Although serverless functions can cost more per unit compute than VMs, for jobs that exhibit widely varying degrees of parallelism throughout their execution [70, 81], serverless allows flexibly scaling up or down compute resources according to the parallelism needed in each stage of the computation, paying for only what is used. For example, Pu et al. [70] demonstrate that for analytics workloads, a serverless system can achieve comparable end-to-end performance and cost to a Spark cluster.

2.1 Comparison to Existing Frameworks

Unfortunately, it is difficult to reap the benefits of general-purpose serverless computing. As soon as one goes beyond event handling workloads, serverless development becomes significantly harder due to a mismatch between what the developer needs and what the serverless platform provides:

- While developers are used to decomposing applications into parallel tasks, they commonly assume that each task can run for an *arbitrary duration* until it completes. Lambda functions, however, are time-bounded and thus might not complete a task of arbitrary length.[2]
- While developers have experience writing parallel code using a *familiar programming model* (e.g., tasks and futures), no such concurrency API is provided by current serverless platforms, which only invoke lambdas concurrently as events fire simultaneously.

This mismatch has led to the development of several frameworks aimed at simplifying serverless programming. Examples include mu [27], PyWren [40], and gg [26] from academia; and AWS Step Functions [8], Azure Durable Functions [58], and Azure Logic Apps [56] from cloud providers. To use these frameworks, the programmer must:

(1) **Partition** the computation into small components, each of which must fit within the lambda function time limit[3] and meet other framework-specific restrictions (e.g., to simplify fault tolerance); and,
(2) **Compose** these components into a serverless application using a framework-specific format.

[2]Although the lambda function time limit varies across platforms and is subject to change, it currently exists in every major serverless offering and a shorter time limit provides more scheduling flexibility to the provider.
[3]For the AWS and Azure offerings, a "component" can also be a call to a managed service, e.g., a facial recognition API; the lambda time limit doesn't apply in this case. Kappa also supports invoking external services (§ 3.4).

Table 1: Comparisons of serverless frameworks.

	Usage Requirements		Features	
	Program representation	Program must be partitioned	Fault tolerant	External service support
mu [27]	State machine	Yes	✗	✗
PyWren [40]	`map`, `wait`	Yes	✗	✗
gg [26]	Data-flow graph	Yes	✓	✗
AWS Step Functions [8]	State machine	Yes	✓	✓
Azure Logic Apps [56]	State machine	Yes	✓	✓
Azure Durable Functions [58]	High-level lang.	Yes	✓	✓
Kappa	High-level lang.	No	✓	✓

While these frameworks simplify serverless development, they share a few limitations as each step of the workflow creates programming burden. For example, chunking a program into components based on running time is a task most programmers are unfamiliar with (unless the workload is already chunked) and requires them to reason about, e.g., the wait time for completing I/O. Once chunked in this manner, combining these components often requires using an unfamiliar specification format (e.g., finite state machines [8, 27, 56] or data-flow graphs [26]), adding further cognitive overhead.

For example, an AWS Step Functions tutorial [9] explains how to "break up a long-running execution into smaller chunks" by first framing an application as repeated executions of a small task, and then implementing a loop as a state machine with *five* states using a JSON-like language [6]. This is significantly more cumbersome than the simple for-loop that would have sufficed in any high-level language.

Table 1 compares Kappa to the prior frameworks. Kappa's checkpointing mechanism (§ 3.2) can resume a task when a lambda function times out, freeing the programmer from having to manually partition their computation. Kappa's high-level concurrency API (§ 3.3) lets programmers develop parallel serverless applications using a familiar programming model. We present additional points of comparison in § 7.1.

Of these frameworks, the closest to solving our challenges (and the most similar to us) is Azure Durable Functions [58], where the programmer creates serverless workflows by writing *orchestrator functions* using a high-level language. An orchestrator can use the familiar `async`/`await` pattern (similar to Kappa's concurrency API) to invoke parallel *activity functions*, each executing a task on a lambda. Kappa differs from Azure Durable Functions in two major ways:

- Durable Functions still requires partitioning computation (and *chaining* the parts in the orchestrator) as activity functions are subject to the time limit, while Kappa's checkpointing deals with lambda timeouts.
- Due to its fault tolerance mechanism, a Durable Functions orchestrator function must be deterministic, and must be manually restarted if it runs for too long (§ 7.1). Kappa does not impose such restrictions.

2.2 Lambda Function Time Limit

As previously discussed, a main constraint of serverless computing is that lambda functions are time-bounded. For example, the AWS Lambda platform started off with a 5-minute limit when it launched, and raised it to 15 minutes four years later to support more applications [4]. Why does the time limit exist, and will providers eventually do away with it?

We are not a cloud provider, and hence cannot definitively know the reason behind these time limits. However, we believe that time-limited functions benefit operators by providing an alternative mechanism for changing task placement. In VM-based cloud computing, operators often need to migrate long-running VMs for consolidation or maintenance purposes using live migration [20], which can be complex, error-prone [75, 93], and resource-consuming [21]. For example, Azure finds live migration "particularly problematic" due to its resource consumption and, to reduce migrations, uses VM lifetime predictions to colocate VMs with similar completion times and refrain from migrating VMs expected to terminate soon [21]. By contrast, in allowing operators to kill jobs after a short time, serverless obviates the operator's need to predict job lifetimes or migrate jobs.

Kappa enables long-running computation on time-bounded lambda functions through checkpointing (§ 3.2). Because checkpointing in Kappa is fast (e.g., less than 5 ms for sizes up to 100 KB; see § 5.1), applications can afford to checkpoint frequently, thereby tolerating time limits even more stringent than imposed by today's platforms. For example, in § 5.3 we demonstrate reasonable performance for two applications under time limits of 1 min and 15 s respectively, where the exact time limit is unknown to the applications. Thus, Kappa offers cloud providers greater flexibility in setting lambda time limits and can even enable non-uniform time limits[4], further improving resource utilization and task placement.

3 Kappa Design

Kappa executes parallel code of arbitrary duration using short-lived lambda functions. In Kappa, a *task* represents a logical thread of execution running in its own memory space, and physically runs on one or more lambda functions. We allow tasks to span multiple lambda functions by periodically

[4]For example, Azure Functions' Premium Plan [59] provides an option where serverless function execution is "guaranteed for 60 minutes, but technically unbounded" [60].

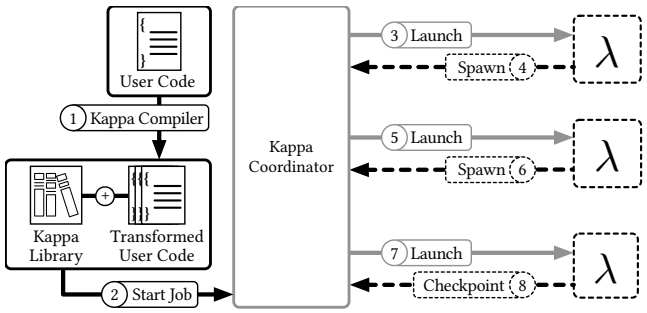


Figure 1: Kappa workflow. Gray lines represent coordinator actions and dashed black lines represent RPCs.

Table 2: Core coordinator RPCs.

Name	Description
`checkpoint()`	Takes a checkpoint.
`fut = spawn(f, args)`	Spawns a task to run `f(args)`; returns a future for the result.
`exit(ret)`	Exits with result `ret`.
`ret = fut.wait()`	Gets result; blocks until ready.
`q = make_queue(sz)`	Creates a queue with max size `sz`.
`q.enqueue(obj)`	Enqueues into `q`; blocks if full.
`obj = q.dequeue()`	Dequeues from `q`; blocks if empty.

checkpointing them. When a lambda function executing a task dies, the task is resumed on another lambda function by restoring the checkpoint. While tasks are single-threaded, Kappa enables concurrent processing by allowing each task to *spawn* other tasks which execute in parallel, and by providing inter-task communication mechanisms that allow tasks to communicate and coordinate with each other.

Kappa has three components: (1) a *coordinator* responsible for launching and resuming tasks (§ 3.1) and for implementing Kappa's concurrency primitives (§ 3.3); (2) a *compiler* responsible for generating code required for checkpointing (§ 3.2); and (3) a *library* used by tasks for checkpointing, concurrent processing, and synchronization.

As shown in Figure 1, when using Kappa, a programmer writes code similar to what runs on a traditional platform, with minor modifications required by Kappa (§ 6). Next, the compiler transforms this code into a form suitable for use by Kappa. Lastly, this program is packaged with the Kappa library and launched by the coordinator, which starts by running a designated "main" task.

3.1 Coordinator

The Kappa coordinator is responsible for scheduling tasks on lambda functions, implementing synchronization and cross-task communication, tracking task metadata (including checkpoints), and providing fault tolerance. Similar to other serverless systems [11, 18, 27, 40], the Kappa coordinator runs as a regular process either on a VM instance in the cloud or on a local machine. We merely require that the lambda functions be able to initiate network connections to the coordinator, and that the coordinator be able to access the same storage services (e.g., S3) as lambda functions.

The coordinator tracks the latest checkpoint of each task. A task writes checkpoint content directly to storage (e.g., S3 or Redis), and the coordinator maintains only checkpoint *locations*. Tasks communicate with the coordinator through the remote procedure calls (RPCs) summarized in Table 2. For instance, after writing a checkpoint, a task uses the `checkpoint` RPC to update its *checkpoint ID* with the coordinator. When a lambda running task t has timed out or failed, the coordinator restarts task t from its latest checkpoint on a new lambda function. Tasks may lose some progress and re-execute some code as a result of being restarted.

Code re-execution can be problematic when making calls with side effects. In Kappa, we assume that all effectful calls are processed through the coordinator, and we ensure that coordinator RPCs are executed once or are never executed, the latter being possible when the task times out (or fails) before successfully contacting the coordinator. We implement this guarantee by requiring the task making an RPC to also take a checkpoint that resumes execution after the RPC. The coordinator points the task's metadata to this checkpoint as it executes the RPC, so that it can resume a failed task from the point where the RPC returns. Compared to relying on user-provided unique IDs or runtime-generated sequential IDs, our mechanism requires neither user intervention nor special handling for nondeterministic code. The Kappa library invokes this mechanism automatically for RPCs.

Coordinator RPCs are synchronous by default, although we also support asynchronous RPCs, which return as soon as the checkpoint has been serialized locally; a background process then persists the checkpoint to storage and contacts the coordinator. Because the actual RPC logic is executed at the coordinator, no processing occurs until the coordinator has been contacted. As a result, in case the lambda is killed before the coordinator is contacted, it is safe to restore the task to its previous checkpoint (before the RPC is issued).

Some RPCs are blocking (e.g., `wait`). A task blocks by first busy-waiting for a configurable period (1 s by default), and then quitting the lambda function. In the latter case, the task is resumed by the coordinator once unblocked. This hybrid approach provides better resource efficiency than pure busy waiting and better performance than immediately blocking.

For fault tolerance of the *coordinator itself*, Kappa provides the option to continuously replicate coordinator state

to a backing store (currently, a Redis cluster using primary-backup replication). With this option enabled, every time the coordinator processes an RPC or a lambda function timeout, it sends a state update to the backing store and waits for it to be persisted. After a failure, the coordinator can reconstruct its previous state from the store and resume the workload. Kappa also supports checkpoint replication to tolerate storage node failures (§ 4). Our evaluations (§ 5) demonstrate low overhead even with both coordinator state and checkpoint replication. If needed, the overhead can be further reduced by having the coordinator send batch updates periodically.

3.2 Checkpointing

As mentioned in § 3.1, Kappa uses checkpoints to tolerate lambda function timeouts and to prevent RPC duplication. Checkpoints in Kappa are implemented using continuations [74], a language-level mechanism executed in user mode. Continuations are a well understood technique for suspending and resuming execution, and have been used in past systems for checkpointing and fault tolerance [79, 88], task migration [28, 78, 79, 88], asynchronous calls [51, 54], context switching [86], and debugging [16]. Kappa's checkpointing technique is nearly identical to that of previous systems, although there might be implementation differences.

For background, we briefly introduce continuations and their usage in Kappa. Our implementation is specialized to Python, and for simplicity we describe our methods in this context. However, continuations have been implemented in many other languages and platforms including Java [28, 78, 79, 86, 88], JavaScript [16, 51, 54], and .NET [66], and our techniques can be easily extended to other languages.

A continuation can be thought of as a closure (i.e., a function with some associated data) that captures program state and control flow information at some execution point; calling the closure resumes execution from this point in the program. Kappa takes a checkpoint by generating a continuation and serializing it to storage, and restores from a checkpoint by deserializing and invoking a previously stored continuation.

Using Listing 1a as an example, we will explain how continuations are generated for the `checkpoint()` call on Line 7. The code is first transformed by the Kappa compiler, which identifies all the *pause points* in the code, i.e., locations where execution can be suspended for checkpointing. The two pause points in Listing 1a are highlighted; note that the call site `bar(x, y)` is identified as a pause point *transitively* since the callee `bar` can cause a checkpoint to be taken.

For each pause point, the compiler generates a *continuation function* definition and inserts it into the source code. As shown in Listing 1b, a continuation function contains all the code that executes after the corresponding pause point,

Listing 1: Example of continuations in Kappa.

(a) Sample application with pause points highlighted.

```
1  def foo(x, y):
2      b = bar(x, y)
3      return b * y
4
5  def bar(x, y):
6      if x < y:
7          checkpoint()
8          z = x + 1
9      else:
10         z = y + 2
11     return z + 3
```

(b) Continuation functions inserted by the compiler.

```
def cont_foo(b, y):
    return b * y


def cont_bar(x):
    z = x + 1
    # Skip the else branch,
    # which is not executed.
    return z + 3
```

(c) The pause point in `foo` is wrapped in exception handling code to unwind the call stack.

```
def foo(x, y):
    try:
        b = bar(x, y)
    except CoordinatorRPCException as e:
        e.add_continuation(cont_foo, y=y)
        raise  # Re-raise to continue unwinding.

    return b * y
```

and takes as arguments any variable whose value is accessed by the subsequent code (these are called *live* variables).[5]

Finally, consider the function call `foo(3,4)` at execution time. The checkpoint taken by `bar` can be written as a list consisting of one continuation per frame on the call stack:

$$[(\texttt{cont\_bar}, \texttt{x} = 3), (\texttt{cont\_foo}, \texttt{b} = \square, \texttt{y} = 4)],$$

where a continuation is a tuple of (1) its continuation function's name and (2) live variables in that frame; "□" denotes a *hole* to be filled in with the previous continuation's result.[6]

To resume from this checkpoint, the Kappa library invokes the continuations in order (e.g., `cont_bar(x=3)`), substituting each continuation's return value into its successor's hole. The return value of the topmost function—i.e., the task's final result—is reported to the coordinator (§ 3.1).

We now provide more details on some mechanisms and designs mentioned in this example.

Generating continuation functions. To reduce runtime overhead, Kappa generates continuation code for each pause point at compile time. Pause points can be inserted manually (by invoking an RPC) or automatically by the compiler using a simple heuristic—before each function call, checkpoint if five seconds has elapsed since the previous checkpoint. We defer finding better heuristics to future work.

[5]Note, for example, that `cont_bar` does not include any code from the `else` branch because it will not be executed afterwards. Nor does it *capture* variable `y`, whose value is not accessed in the continuation code.

[6]The same mechanism is used to return values from RPCs.

For each pause point, the compiler statically identifies live variables [2] and teases out the code that executes afterwards by analyzing the control structure around the pause point (similar to Sekiguchi et al. [78] and Tao [88]), thereby constructing a continuation function. We omit details of this analysis and refer interested readers to prior work. Our compiler can generate continuations for common control flow constructs including `if` statements, `for` and `while` loops, and `continue` and `break` statements; it currently lacks support for some other Python features, which we detail in § 6.

Recall that the compiler must generate continuation functions for transitive pause points, i.e., calls to functions that may checkpoint. Precisely identifying transitive pause points at compile time is challenging for Python, a dynamically typed language with first-class functions. Instead, the Kappa compiler conservatively assumes *every* function call to be a transitive pause point and generates a continuation function for it. Although this strategy bloats the transformed code, for our applications the resulting code size is still negligible compared to the Kappa and third-party libraries.

Runtime behavior. Given the statically generated continuation functions, the Kappa library checkpoints at runtime by unwinding the call stack to create a continuation for each frame. It unwinds the stack using a common exception-based mechanism [28, 51, 66, 79, 88], which we briefly describe.

At every pause point, our compiler inserts an exception handler that creates a continuation upon catching an exception (Listing 1c). To make an RPC, the Kappa library records details of the call and raises a special exception, triggering the exception handler at every level of the call stack. Each handler appends a new continuation and re-raises the exception. Finally, the top-most handler, part of the Kappa library, serializes and persists the list of continuations.

Because the compiler conservatively assumes that every function could checkpoint, it wraps *every* function call in a `try`/`except` block. Even so, the normal execution overhead is minimal since `try`/`except` in Python is "extremely efficient if no exceptions are raised" [71]—our benchmark showed a 10 ns (≈ 8 %) overhead over a no-op function call on AWS Lambda (Python 3.6). During checkpointing, the exception handling overhead is negligible compared to storage I/O.

Why language-level checkpoints? Kappa checkpoints within the high-level language that the application is written in. An alternative, language-agnostic approach is to save a process's address space as in `libckpt` [67] and DMTCP [10]. Although more transparent, such an approach is less flexible in checkpoint construction. For example, while Kappa saves only live variables, a lower level strategy might save extraneous data such as dead values or garbage pages that haven't been returned to the OS. Kappa is also more portable—it works on any serverless platform that supports Python regardless of the underlying OS, including any platform that restricts lambdas to only execute Python code.

Why not use `yield`? Python's `yield` keyword suspends a function and transfers control to its caller. We decided against using `yield` to suspend computation in Kappa (as Pivot [54] does for JavaScript) as we are unaware of any portable technique for serializing the suspended state.[7]

Listing 2: Sample concurrent Kappa program.

```
def count(q):
    ctr = 0
    while q.dequeue() != "":
        ctr += 1
    return ctr

def gen(q):
    q.enqueue("a")
    q.enqueue("")

@on_coordinator
def main():
    q = make_queue(sz = 1)
    fut = spawn(count, (q,))
    spawn(gen, (q,))
    assert fut.wait() == 1
```

3.3 Concurrency API

Kappa provides mechanisms for launching and synchronizing parallel tasks, making it easier to exploit the resource elasticity offered by serverless platforms. Specifically, we provide two basic concurrency abstractions:

Spawn. The `spawn` RPC launches a new task to execute a function call `f(args)` in parallel and returns a future [50] for the result. A spawn is implemented by creating an *initial checkpoint* that, when resumed, executes the function call; the coordinator then invokes a lambda that restores from this checkpoint. We also provide a `map_spawn` RPC, which spawns multiple tasks that run the same function on different arguments; the spawned tasks share an initial checkpoint.

FIFO queues. Kappa tasks can communicate using multi-producer multi-consumer FIFO queues. These queues have bounded size—a task blocks when it enqueues to a full queue or dequeues from an empty queue. These semantics allow queues to be used not only for inter-process communication but also as locks and semaphores.

Listing 2 illustrates the use of these primitives. The entry point function `main` is marked with the `on_coordinator` decorator, indicating that it runs as a process on the coordinator rather than on a lambda function. On-coordinator tasks should be lightweight (e.g., control tasks like `main` here) so that the coordinator machine does not become a bottleneck.

As shown in the code, we modeled our API after Python's built-in multiprocessing API and expect its usage to resemble ordinary parallel programming in Python.

[7] Stackless Python [47] requires a custom Python interpreter, and the `generator_tools` package [77] manipulates Python bytecode, which is neither stable across Python versions nor portable across Python VMs [72].

All aforementioned concurrency operations are implemented as coordinator RPCs, and thus enjoy the RPC fault tolerance guarantees (§ 3.1). Furthermore, by directing lambda-to-lambda communication through the coordinator, Kappa works around the restriction that lambda functions cannot accept inbound network connections [36, 41].

The `enqueue` RPC supports either passing the object to the coordinator as part of the RPC message, or storing the object in storage and only passing a handle. The former mode is faster for small objects (by saving a round trip to storage), while the latter is better for big objects (since the object content doesn't go through RPC processing, also avoiding a coordinator bottleneck). By default, `enqueue` picks a mode using a threshold on object size (see § 5.2).

3.4 External Services

A Kappa task can call services external to the platform (e.g., a REST API for computer vision). Interactions with external services pose two fault tolerance challenges: Kappa must ensure that (1) external calls with side effects be issued only once even when lambdas time out;[8] and (2) calls that last longer than the lambda time limit make progress.

Kappa solves both challenges in an extensible manner using `spawn`: the programmer wraps a stateful external call in a child task, spawns it *on the coordinator* (§ 3.3), and `wait`s for it to finish.[9] The RPC mechanism (§ 3.1) ensures that the `spawn`, and thus the external service call, is never duplicated (assuming no coordinator failures). In case of a long-lasting call, the `wait` would block, causing the parent task to terminate and restart when the child finishes. Note that since the coordinator is not executed on a lambda function, Kappa can run several on-coordinator tasks in parallel.

External storage services are treated specially. Writing to storage is effectful and thus must happen on the coordinator (except for tasks idempotent with respect to storage). To avoid routing all write content through the coordinator, Kappa provides a helper that first writes the content to a temporary location in storage, and then issues a coordinator RPC that moves the temporary file to its intended location. We have implemented this mechanism for S3.

4 Implementation

We have implemented Kappa. Our prototype implementation executes Python 3.6 code on AWS Lambda, although Kappa can be extended to other languages and serverless platforms.

Compiler and library. We implemented the Kappa compiler in Python as described in § 3.2. The compiler-generated code is packaged with the Kappa library, a Python module that implements checkpointing and RPCs. It serializes continuations using Python's built-in and widely used `pickle` library, hence requiring all variables captured in a checkpoint to be serializable using `pickle`.[10] We can relax this requirement by using other serialization libraries, or by having the compiler generate code for unserializable objects.

Coordinator. Our coordinator, written in Go, uses goroutines to schedule Kappa tasks. Each task is managed by a goroutine, which invokes lambda functions synchronously to run the task and processes RPCs from it. Synchronization between tasks maps to synchronization between goroutines inside the coordinator. For example, each Kappa queue is backed by a Go channel; a Kappa task blocking on an `enqueue` RPC is then implemented as its "manager" goroutine blocking on a send to the corresponding Go channel. The coordinator launches Python processes locally to run on-coordinator tasks (§ 3.3).

With fault tolerance enabled, the coordinator persists state changes before performing any action. We use locks in the coordinator and Redis transactions to ensure that state changes and corresponding Redis updates are atomic, providing both durability and consistent ordering for state updates.

Storage. Kappa uses storage services for checkpoints (§ 3.1) and large queue elements (§ 3.3); we refer to both as *stored objects*. We currently support using S3 and Redis for storage.

Kappa supports replicating stored objects for fault tolerance using a user-specified minimum replication factor[11], and can use a higher replication factor to load balance an initial checkpoint shared by many tasks created by a `map_spawn`.

Stored objects are garbage collected using reference counting (to account for checkpoint sharing from `map_spawn`). When an RPC drops an object's reference count to zero, the coordinator instructs the issuing task to delete the object. Garbage collection (GC) is currently implemented only for Redis; since S3 storage is cheap, we simply delete all S3 objects when a workload ends.

5 Evaluation

We now demonstrate the performance and generality of Kappa. For performance, we measure the overhead of checkpointing (§ 5.1) and concurrency operations (§ 5.2) using microbenchmarks, then the end-to-end performance of five applications. These applications come from diverse domains (e.g., SQL queries, streaming analytics, and web crawling)

[8] This subsection assumes no coordinator failures—we cannot guarantee that external side effects be issued only once if the coordinator can fail.

[9] Since such tasks merely block on I/O (e.g., a REST API call), they require little resource and are unlikely to make the coordinator a bottleneck.

[10] The `pickle` module supports most common object types including scalars and collections of serializable objects. See its documentation [73] for details.

[11] A Redis instance can offer a replication factor of > 1 if it has any backups connected to it. When writing a checkpoint, the Kappa library waits for the primary Redis instance and all backups to acknowledge the write.

and exhibit different parallelism patterns (e.g., fork-join, message passing), demonstrating the generality of our approach.

We performed our evaluations on AWS (us-west-2 region). The Kappa coordinator runs on a m5.4xlarge EC2 instance and creates AWS Lambda functions with maximum memory (3008 MB). Unless otherwise noted, we enable coordinator fault tolerance (§ 3.1) and checkpoint replication (§ 4)—the coordinator replicates to a pair of primary-backup Redis instances each on a m5.large VM (two vCPUs and up to 10 Gbps of network), and checkpoints are stored in Redis with a minimum replication factor of 2 (using the same setup).

For applicable microbenchmarks (§ 5.2), we compare to gg [26], PyWren [40] (v0.4), and AWS Step Functions [8]. We configure gg to use a Redis instance (as specified above) as storage; PyWren only supports S3. To improve their performance, we prepackage binaries into gg's lambda functions, and all Python dependencies into PyWren's (so it can skip downloading the Anaconda runtime). We also configured a small 100 ms polling interval for PyWren's `wait` function, which polls S3 for lambda completion.

For end-to-end evaluation (§ 5.3), we compare several Kappa applications to PyWren and Spark implementations (the latter on VMs). We do not compare against gg, which expects ELF executables as input [26] and has no first-class Python support. Nor do we compare against AWS Step Functions, whose lambda startup overhead is too high for any meaningful application measurement (§ 5.2). Because our prototype only supports AWS, we defer comparisons against Azure Logic Apps and Durable Functions to future work.

5.1 Checkpointing Overhead

We run a task on a lambda function that checkpoints every 100 ms and measure the latency. We look at both synchronous checkpoints (where application processing is paused until the checkpoint is persisted and the RPC returns) and asynchronous checkpoints (where a background step persists the checkpoint and makes the RPC). We overlap asynchronous checkpointing with CPU-intensive foreground computation to match expected application behavior. Checkpoints are stored on either S3 or Redis; although S3 provides worse performance (see below), it is more price efficient and requires no additional setup as opposed to Redis, thus representing the easiest option for most users. Finally, each experiment is run with and without replication. For the replication-enabled runs, coordinator state and Redis checkpoints are replicated on a (separate) pair of Redis instances; we do not actively replicate S3 checkpoints.

Figure 2 shows the checkpoint latency for various sizes. Redis outperforms S3 for all sizes although the gap narrows for larger sizes, where throughput dominates. Replication overhead is less than 2 ms for Redis checkpoints of up to 100 KB, but grows more rapidly afterwards as replication becomes throughput-bound. Finally, asynchronous checkpointing is slower as it is overlapped with foreground computation.

Table 3: Checkpoint scaling—latency of parallel lambdas synchronously taking replicated checkpoints of 0.5 KB at a 100 ms interval.

Parallel lambdas	1	10	100	1000
Median latency	3.55 ms	3.63 ms	3.54 ms	3.54 ms
P95 latency	6.17 ms	6.47 ms	4.70 ms	5.41 ms

In terms of checkpoint content, the call stack depth and the number of objects captured have little impact on checkpoint latency for less than ten levels deep and 10 000 objects.

Overall, the median synchronous checkpoint latency is under 5 ms (with replication) for checkpoints up to 100 KB, which is not expected to significantly impact application performance. Data-intensive applications can use asynchronous RPCs to reduce the impact of checkpointing or use custom serialization to reduce checkpoint size.

Finally, we evaluate the scalability of checkpointing by measuring the latency of concurrent lambdas simultaneously taking 0.5 KB checkpoints (with replication) at a 100 ms interval. Table 3 shows that the median checkpoint latency remains less than 4 ms even with 1000 concurrent lambdas. We can achieve good scaling for larger checkpoint sizes as well by scaling up the Redis store (since the coordinator load is unaffected by checkpoint size).

5.2 Performance of Kappa Library

Spawn latency. We have an on-coordinator task measure the latency of `spawn`ing an empty task and `wait`ing for its completion[12], varying the amount of data passed to the task. For comparison, we measure the base spawn latency of gg [26], PyWren [40], and a minimal Go launcher that directly calls the AWS SDK.

As Table 4a shows, even when passing a large 1 MB argument, Kappa launches a task within 55 ms over 95 % of the time. Compared to the minimal Go launcher, a base Kappa spawn is at median 5.5 ms slower due to RPC overhead (e.g., checkpointing and coordinator state replication) and checkpoint loading. This mostly excludes any setup cost because after the first spawn of each run, all subsequent tasks were found to be running on a reused "warm" Python runtime.

In comparison, gg has slightly higher spawn latency partly due to process launches and file operations. PyWren, on the other hand, is significantly slower as its lambda handler polls

[12]Since the `wait` RPC has no side effect, we enabled an optimization that allows `wait` to not take a checkpoint.

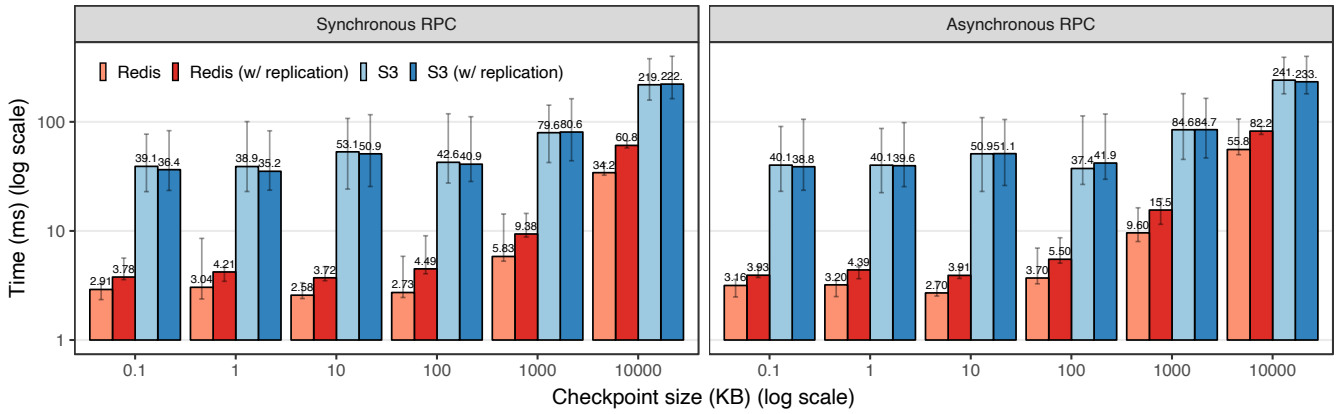


Figure 2: Checkpoint latency by checkpoint size. Each bar shows the median, 5th percentile (P5), and 95th percentile (P95) of 500 measurements. Up to 2% of asynchronous S3 calls failed as they hadn't finished before the next call was issued; all Redis calls succeeded. Redis checkpoints are replicated (where noted) to two instances; S3 checkpoints are not actively replicated.

Table 4: Spawn performance. The "Data" column shows the amount of data passed to the spawned task. The "Go launcher" is a minimal Go program that creates a new lambda function and invokes it directly using the AWS SDK.

(a) Spawn latency of a single task (five runs of 100 sequential spawns each).

		Spawn latency	
	Data size	Median	P95
Kappa	0	15.5 ms	20.5 ms
	100 KB	17.4 ms	23.8 ms
	1 MB	45.8 ms	54.1 ms
Go launcher	0	9.99 ms	14.9 ms
gg [26]	0	30.9 ms	37 ms
PyWren [40]	0	2.54 s	2.72 s

(b) Lambda launch times (mean and std. dev.) in a 1000-lambda launch ($n = 20$). The "reuse λs" Go launcher reuses the same lambda handler across runs.

		Launch time		
	Data size	500th λ	990th λ	1000th λ
Kappa	0	467 ± 16.1 ms	609 ± 51.2 ms	1.31 ± 0.736 s
	100 KB	468 ± 15.4 ms	599 ± 51.6 ms	1.26 ± 0.498 s
	1 MB	533 ± 13.6 ms	815 ± 210 ms	1.62 ± 0.132 s
Go launcher	0	487 ± 45.2 ms	601 ± 46.7 ms	1.09 ± 0.442 s
(reuse λs)	0	347 ± 58.5 ms	414 ± 76.8 ms	457 ± 106 ms
gg [26]	0	668 ± 80.4 ms	749 ± 81.6 ms	809 ± 236 ms
PyWren [40]	0	4.13 ± 0.725 s	6.56 ± 1.12 s	7.81 ± 1.22 s

for task completion at a 2 s interval (not configurable using its API), dominating the running time of short tasks.[13]

Spawn throughput. We measure task launch times in a 1000-task spawn.[14] Each task loads a shared checkpoint load-balanced across four Redis pairs and sleeps for 5 s (subtracted from the reported launch times). We compare to gg, PyWren, and two versions of our Go launcher, one creating a new AWS Lambda handler for each run (as in Kappa), the other reusing the same handler across runs (as in PyWren and gg).

Table 4b shows the mean launch times for the 500th, 990th, and 1000th lambdas. For up to 990 tasks, Kappa adds a roughly 10 ms overhead over the Go launcher without lambda reuse. Unlike in single-`spawn`, this overhead includes one-time setup costs since parallel lambdas cannot recycle execution environments. For the 1000th lambda, Kappa exhibits higher mean and variance than gg. We attribute this gap to Kappa not reusing lambda handlers, given the similar gap between the two Go launchers. PyWren, unlike the other systems, launches lambdas asynchronously and detects their completion by polling S3, leading to worse performance.

Finally, we measure the spawn throughput of AWS Step Functions, which supports both dynamic parallelism (`Map`) and static parallelism (`Parallel`). With `Map`, we encountered

[13] PyWren's lambda handler runs user tasks in a separate process on the lambda and checks for the subprocess' completion periodically. This is separate from the `wait` function, which polls from the "coordinator" machine.
[14] 1000 is our per-region concurrency limit on AWS Lambda.

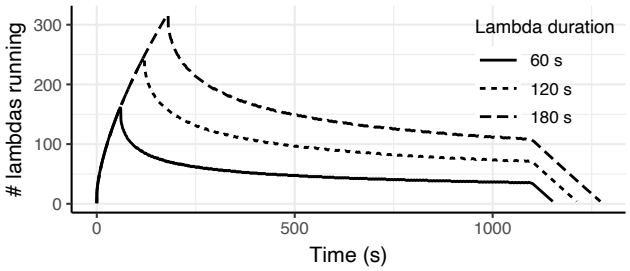


Figure 3: Using AWS Step Functions to launch 1000 lambdas, each sleeping for a fixed duration.

an undocumented concurrency limit of less than 50, too low for most of our applications. With `Parallel`, we launched 1000 lambdas each sleeping for an identical duration. Figure 3 shows the number of lambdas running at each point in time. Although we hit no concurrency ceiling, it took more than *18 minutes* for all 1000 lambdas to launch, 130× to 2300× the duration of the other systems (Table 4b). This overhead would dominate our applications (§ 5.3), many of which repeatedly invoke batches of short tasks. We thus do not offer application comparisons against AWS Step Functions.

Message passing. To quantify task-to-task communication overhead, we report half the round-trip time to send a message to another task and have it echoed back using shared queues (with the receiver busy-waiting). The message is stored either on the coordinator or in Redis (§ 3.3); in both cases it is replicated to two copies, the former as part of coordinator state, and the latter by stored object replication.

Figure 4 shows message passing latency for various configurations. Passing messages through the coordinator is faster for small messages (tens of KB) since it saves a round-trip to Redis, but is slower for larger messages due to RPC processing overhead. It might also make the coordinator a bottleneck, while Redis storage can scale more easily to handle large messages. The `enqueue` RPC, by default, picks a mode using a message size threshold (100 KB in our implementation).

Fault-tolerant writes. For fault-tolerant S3 writes (§ 3.4), we measure the duration from when the write is issued to when the object is moved to its intended location (although asynchronous writes allow application code to proceed during the move). Figure 5 shows fault-tolerant writes to be costly (2.08×–3.06× median overhead for synchronous writes), because we had to implement S3 moves using a slow `COPY-DELETE` sequence. Applications idempotent with respect to storage can avoid this overhead by writing to storage directly.

5.3 Applications

We present five Kappa applications to demonstrate the generality and performance of our framework. For some applications we also created Spark baselines, which we ran on Amazon EMR (Spark 2.3.1) using a m5.4xlarge VM as the master node (same as the Kappa coordinator) and c4.4xlarge VMs as workers. Since Kappa uses only one of two hyperthreads of a lambda, we configured Spark workers to also only use half of all hyperthreads.

TPC-DS. We implemented four SQL queries from the TPC-DS benchmark [90][15] as multi-stage MapReduce jobs on top of Kappa tasks. Tasks in each stage are launched using `map_spawn`, data is shuffled using Kappa queues[16], and query logic is implemented using the *pandas* library [53].

For comparison, we run the same queries with PyWren and Spark using identical query plans; and with Spark SQL, which computes query plans using its query optimizer [13]. All four read input from S3 and are given the same number of worker cores. Kappa and PyWren invoke lambdas from the same VM and are given the same number of Redis instances.[17]

We run the queries using the TPC-DS qualification parameters on data created using the data generator (scale = 100 GB). Kappa launches up to 499 parallel lambda functions (for Q16), and shuffles 5400 (Q1) to 41 865 (Q94, Q95) items.

Figure 6 shows that for each query, Kappa achieves lower runtime than PyWren despite providing stronger fault tolerance (see discussions of PyWren's inefficiencies in § 5.2). Compared to Spark and Spark SQL, Kappa performs comparably or better even without taking into account the time taken to set up a Spark cluster, which can be minutes.

Word count. We implemented word count in the canonical MapReduce style; it reads input from and writes output to S3, and shuffles intermediate data through Kappa queues (backed by two pairs of m5.large Redis VMs). We evaluate Kappa word count, as well as a Spark baseline, using 32 GB of text from Project Gutenberg [68] in 64 MB chunks, and define a "word" as a maximal substring of at least three English letters (case-insensitive). The output consists of 486 697 unique words and their counts in text form, amounting to 7.3 MB in size. Figure 7a shows that Kappa scales well and takes 5.9 % to 21.6 % longer than Spark at median, with the greatest slowdown coming from the largest worker count (= 128).

Parallel grep. This application counts the occurrences of a string in a *single* file split into chunks. Each worker

[15] We used the queries for which PyWren implementations are available [69] (Q1, Q16, Q94, and Q95); we modified them to improve their performance.
[16] These queues are configured to pass data through Redis. By the fault tolerance setup (§ 5), Kappa stores two copies of all shuffle data.
[17] We use 18 m5.large VMs (each has 2 vCPUs and runs one Redis process). For Kappa, they are configured as 9 master-slave pairs, one for coordinator state replication and the rest for checkpoints and shuffle data. The PyWren version uses all instances for data shuffling with no replication.

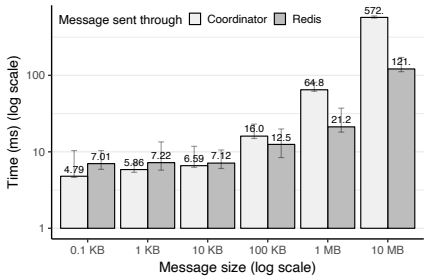


Figure 4: Latency of message passing between tasks (median, P5, and P95).

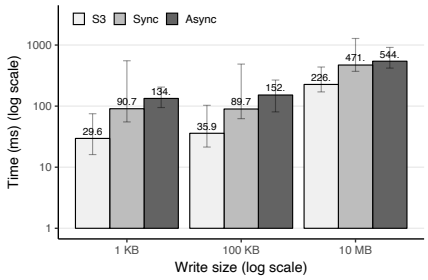


Figure 5: Latency of fault-tolerant vs raw S3 writes (median, P5, and P95).

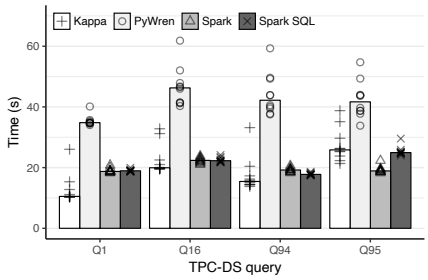
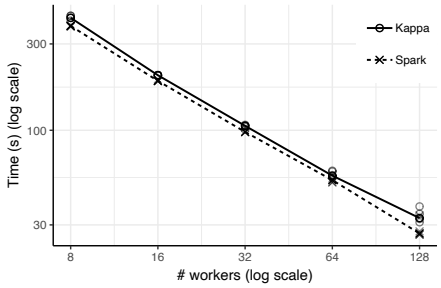
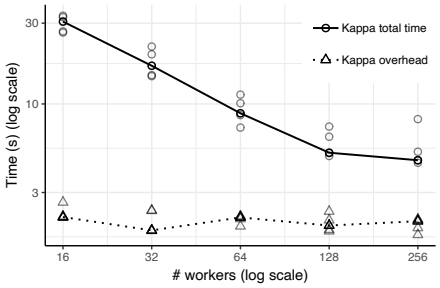


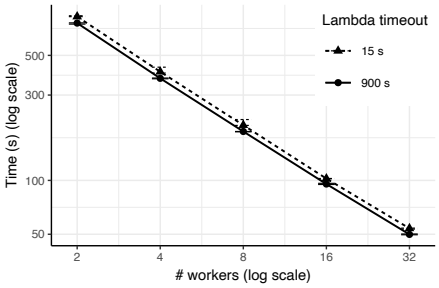
Figure 6: TPC-DS query run time. Bars show the median of ten runs.



(a) Word count (median, $n = 5$).



(b) Parallel grep (median, $n = 5$).



(c) Streaming (median, min/max, $n = 3$).

Figure 7: Strong scaling results for three applications.

processes a contiguous range of chunks and, to deal with substrings that straddle chunks, communicates boundary content with neighbors using queues.

We run parallel grep to search 32 GB of data on S3 for a three-character string that occurs roughly 200 million times. In addition to reporting the running time, we compute the workload's *actual* duration—the makespan computed using the *actual* duration of each task and the task dependency graph; a task's *actual* duration is a conservative measurement of time spent reading input, computing, and writing output. We then report the overhead (defined as the difference), which includes lambda invocation, checkpointing, etc.

Figure 7b shows that parallel grep scales well up to 128 workers (each processing two chunks), while the overhead remains roughly constant at around 2 s. We did not compare against a Spark implementation since the Spark programming model does not naturally support dealing with the boundary between two consecutive chunks of the input.[18]

Streaming. This workload uses parallel workers to compute the average number of hashtags per tweet in a stream of tweets. Workers pull from a shared work queue and send periodic updates to an aggregator task; they checkpoint before processing each chunk as a result of calling dequeue. For evaluation, we use 64 GB of tweets in JSON [89] stored as 128 MB chunks on S3; downloading and processing a chunk takes roughly 5 s. To stress test timeout recovery, we imposed a short lambda timeout of 15 s and compare to runs that experienced no timeouts. Figure 7c shows that the application scales well and that the 15 s timeout increases the median duration by 6.8 % to 8.9 %. A less aggressive timeout of 60 s (not shown) adds a 0.9 % to 3.2 % overhead to impacted runs.

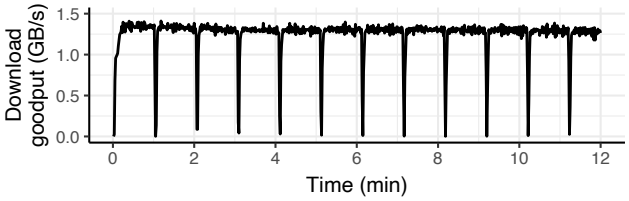


Figure 8: Web crawler with 1000 workers and a 1 min lambda timeout—Aggregate download rate of bytes from *new* pages.

[18] Although Spark supports seeking to the first newline (or a custom delimiter) after an input chunk boundary, it remains a challenge to, e.g., search for a byte sequence in a binary input where no natural delimiter exists.

Web crawler. We implemented a distributed web crawler based off UbiCrawler [17] that downloads web pages starting from some seed domains [5]. A hash function partitions domains between workers. When a worker sees a domain outside its partition, it uses a Kappa queue to send the domain name to a scheduling task for de-duplication and reassignment. Each worker uses Python asynchronous I/O to issue 30 concurrent downloads.[19] Because our prototype compiler doesn't handle `async`/`await` (§ 6), we treat our I/O code as an "external library" within which no checkpoints are taken. When its URL frontier grows beyond a certain size, a worker stores the remainder of the frontier in Redis (by enqueueing to a queue) and retrieves it when its frontier runs empty.

We ran the web crawler with 1000 workers for 12 min; each worker checkpoints after processing every 20 pages. To exercise Kappa's fault tolerance, we imposed a short lambda timeout of 1 min. Figure 8 shows the aggregate rate of bytes downloaded from *new* pages. Sudden dips in throughput, which last from 1 s to 2 s, indicate workers timing out and restarting. The crawler achieved a median stable throughput of 15 587 page/s, and downloaded 10.9 million unique pages in total. The coordinator processed an average of 976 RPC/s.

6 Limitations and Future Work

While Kappa already supports a wide range of applications, it has a few limitations, which we discuss below.

Unsupported Python features. We have not yet added support for some Python features to the compiler's continuation generation logic. These features include `try`/`except`, `yield`, `async`/`await`, nested function definitions, and global variables. These limitations are not fundamental—prior work has shown how to implement these features in continuation passing style, a paradigm that our approach depends on. For example, Stopify [16] shows how to generate continuations for nested functions and exception handling in JavaScript.

Other restrictions on input code. Beyond the aforementioned restrictions on Python features, Kappa requires minor modifications to application code. The programmer must:

- Insert **`checkpoint()`** calls at appropriate points in the program, e.g., before calling external library functions that might take a long time;
- Mark calls that have **externally visible side-effects** (e.g., resulting in I/O) with `@on_coordinator`, ensuring that such calls are executed only once (§ 3.4); and,
- Use Kappa's **concurrency primitives** (§ 3.3) instead of primitives such as Python threads.

We did not find these requirements burdensome. Among all the applications from § 5.3, we only had to insert *one* `checkpoint()` call in *one* application.[20] In all other cases, the program already checkpoints frequently enough by invoking RPCs, and library calls do not last long enough to require checkpoints. Nor was our concurrency API any hurdle to use as it resembles Python's built-in multiprocessing API.

Static pause points. Recall that Kappa identifies pause points statically (§ 3.1). This approach reduces the runtime overhead of checkpointing, but restricts where checkpoints can be generated and precludes deciding checkpoint locations at runtime. Relaxing this limitation through dynamic continuation computation is left to future work.

Can only checkpoint in transformed code. Kappa can checkpoint only in code transformed by the its compiler and not in, e.g., a Python C extension like numpy. Control must therefore return to Kappa-instrumented code once in a while for checkpoints to be taken.

Lack of static checking. Python's dynamic nature makes it challenging to statically analyze application code (as noted in prior work [63]). For example, the Kappa compiler does not ensure at compile time that every variable captured by a checkpoint is serializable. A future direction is to implement static checking by leveraging Python type hints [31, 92].

Unimplemented GC features. Our garbage collection implementation (§ 4) currently does not delete (1) a task's last checkpoint after it exits, or (2) any object "orphaned" due to lambda timeout or failure (i.e., written to storage but not reported to the coordinator); these objects are deleted when the workload finishes. Adding these features does not change our design and would add little overhead to the critical path.

7 Related Work

7.1 Serverless Programming Frameworks

In § 2.1, we compared Kappa to existing serverless frameworks in terms of their usage models. Here we compare the other features listed in Table 1.

Fault tolerance. A fault-tolerant serverless framework avoids restarting a workload from scratch when lambda functions and/or the coordinator-equivalent fails. The mu [27] and PyWren [40] frameworks provide no fault tolerance. gg [26] and Azure Durable Functions [58] adopt replay-based fault tolerance—a failed component restarts from the beginning and skips any completed tasks by consulting a history table. These replay-based approaches, while performant and transparent, have two drawbacks:

- They require application execution to be deterministic (otherwise execution may diverge during replay).

[19] To avoid overwhelming the crawled websites, we make sure to issue at most one connection to each distinct domain across all workers.

[20] The web crawler has a `checkpoint()` call inserted to make sure that a checkpoint is taken for at most every 20 pages fetched.

- The history table can grow unboundedly even if program state has bounded size—e.g., a long-running orchestrator function on Azure must be restarted manually once in a while to avoid memory exhaustion [57].

Kappa handles nondeterminism by checkpointing before each RPC, and avoids blowup by storing state (i.e., checkpoints and coordinator state) rather than history.

AWS Step Functions [8] and Azure Logic Apps [56] support specifying retry policies for failed tasks; only the former handles availability zone failures [30]. The academic frameworks do not handle provider datacenter failures.

External services. Kappa supports calling external services (§ 3.4). The mu, PyWren, and gg frameworks lack this support, while the frameworks from cloud providers support integrating external services into serverless workflows.

Other features. Some of the features provided by previous systems that Kappa lacks include RPC pipelining [27], dependency inference [26], straggler mitigation [26], and workflow visualization [8, 56]. Integrating these features into Kappa, where applicable, is deferred to future work.

7.2 Other Related Work

Checkpoint and restart. Our underlying techniques—saving program state upon interrupts or failures and later resuming from it—are pervasive in the systems literature. Prior systems have used user-mode checkpointing for fault tolerance (e.g., `libckpt` [67] and DMTCP [10]), process migration (e.g., `c2ftc` [87]), asynchronous programming (e.g., Tame [46] and SEDA [95]), and virtualization [45] and resilience to power failure [52, 91] in embedded devices. Others, like VMADump [37, § 3.4], CRAK [99], and BLCR [35], rely on in-kernel support to checkpoint and/or migrate user programs. This latter category is not a good fit for existing serverless environments, where kernel modification (including the loading of kernel modules) is prohibited.

Improvements to serverless platforms. Recent works have proposed serverless-optimized storage and caching solutions (e.g., Pocket [44], Savanna [29, § 3], Cloudburst [85], Aft [84], and HydroCache [97]), security enforcement via information flow control [3], and techniques to optimize the performance and resource usage of serverless platforms [1, 42, 49, 61, 65, 80, 82, 85, 94]. Kappa automatically benefits from transparent platform improvements, and can exploit new storage services by placing checkpoints and large queue elements there (§ 4).

Concurrent processing frameworks. Our concurrency API (§ 3.3), which is based on message passing, resembles those of actor frameworks like Erlang [14] and Akka [48]. An alternative approach, à la MapReduce [22] and Spark [98], relies on assumptions about program structure to parallelize computation. As shown in § 5.3, structured parallelism can be easily implemented using Kappa's lower level primitives. Ray [62] supports task- and actor-based concurrency using APIs similar to Kappa's. Although Ray can checkpoint its actors, it requires manually implementing state saving and restoration for each actor [19]; Kappa automates this process.

Continuations. Continuations have been studied extensively in programming languages and compiler optimization [74]. For example, several compilers translate source code to continuation-passing style (CPS) for some compilation passes [12]. Kappa does not directly translate code to CPS—such a translation would cause significant slowdowns as Python does not optimize for closure creation or tail calls.[21] Our transformation (§ 3.1) avoids this slowdown by largely preserving the code's control structure and only creating continuation objects when a checkpoint is taken.

As mentioned in § 3.2, continuations have been used by many prior systems [16, 28, 51, 66, 78, 79, 86, 88]. In addition, they are used by Mach 3.0 [24] for process blocking and by Ciel [64] for task blocking. Kappa uses similar techniques to allow tasks to block on RPC responses.

8 Conclusion

Although serverless computing originally targeted event handling, recent efforts such as ExCamera [27] and PyWren [40] have enabled the use of serverless for more diverse applications. However, developing serverless applications still requires significant effort. Kappa is a framework that simplifies serverless development by providing a familiar programming model. By reducing the friction of developing serverless programs, Kappa provides an avenue for a larger set of applications to take advantage of the benefits of serverless computing.

Acknowledgments

We thank the anonymous reviewers, James McCauley, Edward Oakes, other members of the UC Berkeley NetSys Lab, Pratyush Patel, Gur-Eyal Sela, Irene Zhang, and Akshay Narayan for their feedback. This work was funded in part by NSF Grants 1817115, 1817116, and 1704941, and by grants from Intel, VMware, Ericsson, Futurewei, Cisco, Amazon, and Microsoft.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* https://www.usenix.org/conference/atc18/presentation/akkus

[2] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization.* https://doi.org/10.1145/800028.808479

[21] Loitsch [51] notes a similar phenomenon for JavaScript.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 118 (Oct. 2018). https://doi.org/10.1145/3276488

[4] Amazon Web Services. 2018. *AWS Lambda enables functions that can run up to 15 minutes.* Retrieved Jan 9, 2020 from https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/

[5] Amazon Web Services. 2019. *Alexa Top 1-Million.* http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

[6] Amazon Web Services. 2019. *Amazon States Language—AWS Step Functions.* https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html

[7] Amazon Web Services. 2019. *AWS Lambda—Serverless Compute—Amazon Web Services.* https://aws.amazon.com/lambda/

[8] Amazon Web Services. 2019. *AWS Step Functions.* https://aws.amazon.com/step-functions/

[9] Amazon Web Services. 2019. *Iterating a Loop Using Lambda—AWS Step Functions.* Retrieved September 19, 2019 from https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html

[10] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09).* https://doi.org/10.1109/IPDPS.2009.5161063

[11] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18).* https://doi.org/10.1145/3267809.3267815

[12] Andrew W. Appel. 2007. *Compiling with Continuations.* Cambridge University Press, USA.

[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).* https://doi.org/10.1145/2723372.2742797

[14] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph.D. Dissertation. The Royal Institute of Technology.

[15] Arda Aytekin and Mikael Johansson. 2019. Harnessing the Power of Serverless Runtimes for Large-Scale Optimization. (2019). arXiv:1901.03161 http://arxiv.org/abs/1901.03161

[16] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. 2018. Putting in All the Stops: Execution Control for JavaScript. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* https://doi.org/10.1145/3192366.3192370

[17] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19).* https://doi.org/10.1145/3357223.3362711

[19] Hao Chen. 2019. *Implement actor checkpointing by raulchen • Pull Request #3839 • ray-project/ray.* Retrieved April 22, 2019 from https://github.com/ray-project/ray/pull/3839

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI '05).*

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* https://doi.org/10.1145/3132747.3132772

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

[23] John Demian. 2018. *Companies using serverless in production.* Retrieved Sep 18, 2019 from https://dashbird.io/blog/companies-using-serverless-in-production/

[24] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. 1991. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91).* https://doi.org/10.1145/121132.121155

[25] L. Feng, P. Kudva, D. Da Silva, and J. Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* 334–341. https://doi.org/10.1109/CLOUD.2018.00049

[26] 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).* https://www.usenix.org/conference/atc19/presentation/fouladi

[27] 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).* https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[28] Stefan Fünfrocken. 1998. Transparent migration of Java-based mobile agents: Capturing and re-establishing the state of Java programs. *Personal Technologies* 2, 2 (01 Jun 1998). https://doi.org/10.1007/BF01324941

[29] Xiang Gao. 2020. *Next Generation Datacenter Architecture.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-30.html

[30] GoncalvesJ and WSilveiraNZ. 2018. *Logic App Geo-Replication / Disaster-Recovery.* Retrieved September 19, 2019 from https://social.msdn.microsoft.com/Forums/azure/en-US/b2fd4ad3-2566-42f6-a0d0-8374b868eaf7/logic-app-georeplicationdisasterrecovery

[31] Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, and Guido van Rossum. 2016. *PEP 526—Syntax for Variable Annotations.* Retrieved Apr 19, 2019 from https://www.python.org/dev/peps/pep-0526/

[32] Google. 2019. *Cloud Functions—Event-driven Serverless Computing | Cloud Functions | Google Cloud.* https://cloud.google.com/functions/

[33] Vipul Gupta, Dominic Carrano, Yaoqing Yang, Vaishaal Shankar, Thomas A. Courtade, and Kannan Ramchandran. 2020. Serverless Straggler Mitigation using Local Error-Correcting Codes. (2020). arXiv:2001.07490 https://arxiv.org/abs/2001.07490

[34] Vipul Gupta, Swanand Kadhe, Thomas A. Courtade, Michael W. Mahoney, and Kannan Ramchandran. 2019. OverSketched Newton: Fast Convex Optimization for Serverless Systems. (2019). arXiv:1903.08857 http://arxiv.org/abs/1903.08857

[35] Paul H Hargrove and Jason C Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46 (sep 2006). https://doi.org/10.1088/1742-6596/46/1/067

[36] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *Conference on Innovative Data Systems Research (CIDR '19)*. https://arxiv.org/abs/1812.03651

[37] Erik Hendriks. 2002. BProc: The Beowulf Distributed Process Space. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. https://doi.org/10.1145/514191.514212

[38] IBM. 2019. *Cloud Functions—Overview | IBM*. https://www.ibm.com/cloud/functions

[39] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018*. https://doi.org/10.1109/IC2E.2018.00052

[40] 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 (SoCC '17)*.

[41] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[42] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. https://doi.org/10.1145/3357223.3362709

[43] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*.

[44] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. https://www.usenix.org/conference/osdi18/presentation/klimovic

[45] Neil Klingensmith and Suman Banerjee. 2018. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications (HotMobile '18)*. https://doi.org/10.1145/3177102.3177103

[46] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events Can Make Sense. In *Proceedings of the USENIX Annual Technical Conference (ATC '07)*.

[47] Anselm Kruis. 2019. *Home • stackless-dev/stackless Wiki*. Retrieved Sep 9, 2019 from https://github.com/stackless-dev/stackless/wiki

[48] Lightbend. 2019. *Akka*. https://akka.io/

[49] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. (2019). arXiv:1903.12221 http://arxiv.org/abs/1903.12221

[50] B. Liskov and L. Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. https://doi.org/10.1145/53990.54016

[51] Florian Loitsch. 2007. Exceptional Continuations in JavaScript. In *2007 Workshop on Scheme and Functional Programming* (Freiburg, Germany). http://www.schemeworkshop.org/2007/procPaper4.pdf

[52] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. https://www.usenix.org/conference/osdi18/presentation/maeng

[53] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[54] James Mickens. 2014. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. https://doi.org/10.1109/SP.2014.24

[55] Microsoft. 2019. *Azure Functions—Develop Faster With Serverless Compute | Microsoft Azure*. https://azure.microsoft.com/en-us/services/functions/

[56] Microsoft. 2019. *Logic App Services*. https://azure.microsoft.com/en-us/services/logic-apps/

[57] Microsoft. 2019. *Orchestrator function code constraints*. Retrieved September 19, 2019 from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-code-constraints

[58] Microsoft. 2019. *What are Durable Functions?* https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[59] Microsoft. 2020. *Azure Functions Premium Plan*. Retrieved September 14, 2020 from https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan

[60] Microsoft. 2020. *host.json reference for Azure Functions 2.x and later*. Retrieved September 14, 2020 from https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#functiontimeout

[61] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. https://www.usenix.org/conference/osdi18/presentation/moritz

[63] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. 2014. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muller

[64] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*. https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing

[65] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. https://www.usenix.org/conference/atc18/presentation/oakes

[66] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from Generalized Stack Inspection. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. https://doi.org/10.1145/1086365.1086393

[67] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON '95)*.

[68] Project Gutenberg [n.d.]. *Project Gutenberg*. Retrieved Aug 2018 from http://www.gutenberg.org

[69] Qifan Pu. 2018. *PyWren TPC-DS scripts*. Retrieved April 17, 2018 from https://github.com/ooq/tpcds-pywren-scripts

[70] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. https://www.usenix.org/conference/nsdi19/presentation/pu

[71] Python Software Foundation. 2019. *Design and History FAQ*. Retrieved April 15, 2019 from https://docs.python.org/3.6/faq/design.html

[72] Python Software Foundation. 2019. `dis`—*Disassembler for Python bytecode*. Retrieved Sep 9, 2019 from https://docs.python.org/3/library/dis.html

[73] Python Software Foundation. 2019. `pickle`—*Python object serialization*. Retrieved Sep 9, 2019 from https://docs.python.org/3/library/pickle.html

[74] John C. Reynolds. 1993. The Discoveries of Continuations. *Lisp Symb. Comput.* 6, 3–4 (Nov. 1993), 233–248. https://doi.org/10.1007/BF01019459

[75] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration At Scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. https://doi.org/10.1145/3186411.3186415

[76] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry (Middleware '18)*. https://doi.org/10.1145/3284028.3284029

[77] Kay Schluehr. 2009. *generator_tools*. Retrieved Sep 9, 2019 from http://www.fiber-space.de/generator_tools/doc/generator_tools.html

[78] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. 1999. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In *Proceedings of the Third International Conference on Coordination Languages and Models (COORDINATION '99)*.

[79] Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. 2001. *Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 217–233. https://doi.org/10.1007/3-540-45407-1_14

[80] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. https://www.usenix.org/conference/atc20/presentation/shahrad

[81] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. *numpywren: serverless linear algebra*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-137.html

[82] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. https://www.usenix.org/conference/atc20/presentation/shillaker

[83] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. 2020. Optimizing Prediction Serving on Low-Latency Serverless Dataflow. (2020). arXiv:2007.05832 https://arxiv.org/abs/2007.05832

[84] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. https://doi.org/10.1145/3342195.3387535

[85] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[86] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP '08)*. https://doi.org/10.1007/978-3-540-70592-5_6

[87] Volker Strumpen and Balkrishna Ramkumar. 1998. *Portable Checkpointing for Heterogeneous Architectures*. Springer US, Boston, MA, 73–91. https://doi.org/10.1007/978-1-4615-5449-3_4

[88] Wei Tao. 2001. *A Portable Mechanism for Thread Persistence and Migration (Mobile Agent)*. Ph.D. Dissertation. Advisor(s) Lindstrom, Gary. AAI3005121.

[89] The Internet Archive. 2015. *Archive Team JSON Download of Twitter Stream 2015-05*. https://archive.org/details/archiveteam-twitter-stream-2015-05

[90] Transaction Processing Performance Council. 2018. TPC Benchmark™ DS Standard Specification Version 2.8.0.

[91] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

[92] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. *PEP 484—Type Hints*. Retrieved Apr 19, 2019 from https://www.python.org/dev/peps/pep-0484/

[93] Scott Van Woudenberg. 2016. *Lessons learned from a year of using live migration in production on Google Cloud*. Retrieved Sep 10, 2019 from https://bit.ly/36M7T3c

[94] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. https://doi.org/10.1145/3302424.3303978

[95] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. https://doi.org/10.1145/502034.502057

[96] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. 2018. Serverless Big Data Processing using Matrix Multiplication as Example. In *IEEE International Conference on Big Data, Big Data 2018*. https://doi.org/10.1109/BigData.2018.8622362

[97] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. https://doi.org/10.1145/3318464.3389710

[98] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[99] Hua Zhong and Jason Nieh. 2001. *CRAK: Linux Checkpoint/Restart As a Kernel Module*. Technical Report CUCS-014-01. Department of Computer Science, Columbia University.