Stateful Serverless Computing with Crucial

DANIEL BARCELONA-PONS, Universitat Rovira i Virgili
PIERRE SUTRA, Télécom SudParis
MARC SÁNCHEZ-ARTIGAS and GERARD PARÍS, Universitat Rovira i Virgili
PEDRO GARCÍA-LÓPEZ, Universitat Rovira i Virgili, and IBM T.J. Watson Research Center

Serverless computing greatly simplifies the use of cloud resources. In particular, Function-as-a-Service (FaaS) platforms enable programmers to develop applications as individual functions that can run and scale independently. Unfortunately, applications that require fine-grained support for mutable state and synchronization, such as machine learning (ML) and scientific computing, are notoriously hard to build with this new paradigm. In this work, we aim at bridging this gap. We present Crucial, a system to program highly-parallel stateful serverless applications. Crucial retains the simplicity of serverless computing. It is built upon the key insight that FaaS resembles to concurrent programming at the scale of a datacenter. Accordingly, a distributed shared memory layer is the natural answer to the needs for fine-grained state management and synchronization. Crucial allows to port effortlessly a multi-threaded code base to serverless, where it can benefit from the scalability and pay-per-use model of FaaS platforms. We validate Crucial with the help of micro-benchmarks and by considering various stateful applications. Beyond classical parallel tasks (e.g., a Monte Carlo simulation), these applications include representative ML algorithms such as k-means and logistic regression. Our evaluation shows that Crucial obtains superior or comparable performance to Apache Spark at similar cost (18%–40% faster). We also use Crucial to port (part of) a state-of-the-art multi-threaded ML library to serverless. The ported application is up to 30% faster than with a dedicated high-end server. Finally, we attest that Crucial can rival in performance with a single-machine, multi-threaded implementation of a complex coordination problem. Overall, Crucial delivers all these benefits with less than 6% of changes in the code bases of the evaluated applications.

CCS Concepts: • **Theory of computation → Distributed computing models**; • **Computer systems organization → Cloud computing;**

Additional Key Words and Phrases: Serverless, FaaS, in-memory, stateful, synchronization

An initial version of this article appeared in the proceedings of the 20th ACM/IFIP International Middleware conference [17]. This extended version includes new programming abstractions, a more thorough evaluation and new use cases.
This work has been partially supported by the EU Horizon 2020 programme (CloudButton, grant 825184) and by the Spanish Government (PID2019-106774RB-C22). Daniel Barcelona-Pons's work is financed by a Martí i Franquès programme grant. Marc Sánchez-Artigas is a Serra Húnter Fellow.
Authors' addresses: D. Barcelona-Pons, M. Sánchez-Artigas, and G. París, Universitat Rovira i Virgili Departament d'Enginyeria Informàtica i Matemàtiques Avinguda Països Catalans, 26 43007 Tarragona, Spain; emails: {daniel.barcelona, marc.sanchez, gerard.paris}@urv.cat; P. Sutra, Télécom SudParis 19 Place Marguerite Perey, F-91120 Palaiseau, France; email: pierre.sutra@telecomsudparis.eu; P. García-López, Universitat Rovira i Virgili Departament d'Enginyeria Informàtica i Matemàtiques Avinguda Països Catalans, 26 43007 Tarragona, Spain and IBM T J Watson Research Center 1101 Kitchawan Road Yorktown Heights, NY 10598; email: pedro.garcia@urv.cat.

1049-331X/2022/03-ART39
https://doi.org/10.1145/3490386

ACM Reference format:
Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 39 (March 2022), 38 pages.
https://doi.org/10.1145/3490386

1 INTRODUCTION

By abstracting away the provisioning of compute resources, serverless computing removes much of the complexity to use the cloud. This fairly recent paradigm was started by services such as Google BigQuery [76] and AWS Glue [15], and it has evolved into **Function-as-a-Service** (**FaaS**) computing platforms, such as AWS Lambda and Apache OpenWhisk. These platforms allow to deploy a user-defined *cloud function* and its dependencies. Once deployed, the function is fully managed by the platform provider that executes it on demand and at scale in a datacenter. Cloud functions are billed at sub-second granularity and only the time they execute is charged to the user.

Current practices show that serverless computing works well for applications that require a small amount of storage and memory due to the operational limits set by the providers (see, e.g., AWS Lambda [12]). However, there are more limitations. While cloud functions can initiate outgoing network connections, they cannot directly communicate between each other, and have little bandwidth compared to a regular virtual machine [24, 101]. This is because the model was originally designed to execute event-driven functions in response to user actions or changes in the storage tier (e.g., uploading a file to Amazon S3 [11]). Despite these constraints, serverless computing applies to many areas. Some recent works show that this paradigm allows to process big data [55, 80, 84], encode videos [35], and perform linear algebra [88] and Monte Carlo simulations [52].

Challenges. All these pioneering works prove that serverless computing can escape its initial area of usage and expand to traditional computing applications. However, programming some of these tasks still face fundamental challenges. Although the list is too long to recount here, convincing cases of these ill-suited applications are distributed stateful computations such as **machine learning** (**ML**) algorithms. Just an imperative implementation of k-means [66] raises several issues: first, the need to efficiently handle a globally-shared state at fine granularity (the cluster centroids); second, the problem to globally synchronize the cloud functions, so that the algorithm can correctly proceed to the next iteration; and finally, the prerogative that the shared state survives system failures.

No serverless system currently addresses all these issues effectively. First, due to the impossibility of function-to-function communication, the prevalent practice for sharing state across functions is to use remote storage. For instance, serverless frameworks, such as PyWren and NumPyWren [88], use highly-scalable object storage services to transfer state between cloud functions. Since object storage is too slow to share short-lived intermediate state in serverless applications [62], some recent works use faster storage solutions. This has been the path taken by Locus [80], which proposes to combine fast, in-memory storage instances with slow storage to scale shuffling operations in MapReduce. However, with all the shared state transiting through storage, one of the major limitations of current serverless systems is the lack of support to handle mutable state at a fine granularity (e.g., to efficiently aggregate small granules of updates). Such a concern has been recognized in various works [24, 56], but this type of fast, enriched storage layer for serverless computing is not available today in the cloud, leaving fine-grained state sharing as an open issue.

Similarly, FaaS orchestration services (such as AWS Step Functions [14] or OpenWhisk Composer [36]) offer limited capabilities to synchronize cloud functions [39, 56]. They have no abstraction to signal a function when a condition is fulfilled, or for multiple functions to coordinate, e.g., in order to guarantee data consistency, or to ensure joint progress to the next stage of computation. Of course, such fine-grained synchronization should be also low-latency to not significantly slow down the application. Existing stand-alone notification services, such as AWS- SNS [20] and AWS-SQS [40], add significant latency, sometimes hundreds of milliseconds. This lack of efficient synchronization mechanisms means that each serverless framework needs to develop its own solutions. For instance, PyWren enforces the coordination of the map and reduce phases through object storage, while ExCamera has built a notification system using a long-running VM-based rendezvous server. As of today, there is no general way to let multiple functions coordinate via abstractions hand-crafted by users, so that fine-grained synchronization can be truly achieved.

Contributions. To overcome the aforementioned issues, we propose Crucial, a framework for the development of stateful distributed applications in serverless environments. The base abstraction of Crucial is the *cloud thread,* which maps a thread to the invocation of a cloud function. Cloud threads manipulate global shared state stored in the ***distributed shared objects*** (**DSO**) layer. To ease data sharing between cloud functions, DSO provides out-of-the-box strong consistency guarantees. The layer also implements fine-grained synchronization, such as collectives, to coordinate the functions. Objects stored in DSO can be either ephemeral or persistent, in which case they are passivated on durable storage. DSO is implemented with the help of state machine replication and executes atop an efficient disaggregated in-memory data store. Cloud threads can run atop any standard FaaS platform.

The programming model of Crucial is quite simple, offering conventional multi-threaded abstractions to the programmer. With the help of a few annotations and constructs, single-machine multi-threaded stateful programs can execute as cloud functions. In particular, since the global state is manipulated as remote shared objects, the interface for mutable state management becomes virtually unlimited, only constrained by the expressiveness of the programming language—Java in our case.

Our evaluation shows that Crucial can scale traditional parallel jobs, such as Monte Carlo simulations, to hundreds of workers using basic code abstractions. For applications that require fine-grained updates, like ML tasks, our system can rival, and even outperform, Apache Spark running on a dedicated cluster. We also establish that an application ported to serverless with Crucial achieves similar performance to a multi-threaded solution running on a dedicated high-end server.

We claim the following novel contributions:

— We provide the first concrete evidence that stateful applications with needs for fine-grained data sharing and synchronization can be efficiently built using stateless cloud functions and a disaggregated shared objects layer.

— We design Crucial, a system for the development and execution of stateful serverless applications. Its simple interface offers fine-grained semantics for both mutable state and synchronization. Crucial is open source and freely available online [6].

— We show that Crucial is a convenient tool to write serverless-native applications, or port legacy ones. In particular, we describe a methodology to port traditional single-machine applications to serverless.

— Crucial is suited for many applications such as traditional parallel computations, ML algorithms, and complex concurrency tasks. We show that a Monte Carlo simulation and a Mandelbrot computation can easily scale over on-demand serverless resources. Using an extensive evaluation of the k-means and logistic regression algorithms over a 100 GB dataset, we show that Crucial can lead to an 18%–40% performance improvement over Apache Spark running on dedicated instances at comparable cost. Crucial is also within 8% of the completion time of the Santa Claus problem running on a local machine.

— Using Crucial, we port to serverless part of Smile [65], a state-of-the-art multi-threaded ML library. The portage impacts less than 4% of the original code base. It brings elasticity and on-demand capabilities to a traditional single-machine program. With 200 cloud threads, the random forest classification algorithm ported to serverless is up to 30% faster than a 4-CPU 160-threads dedicated server solution.

The remaining of the article is structured as follows: Section 2 provides a general background and motivation for this work. We explain Crucial's programming model in Section 3, and describe its design in Section 4. Section 5 covers implementation details. The evaluation is presented in Section 6, where we validate the effectiveness of Crucial for fine-grained state management and synchronization in serverless environments. We review related work in Section 7 before closing in Section 8.

2 BACKGROUND

This section introduces serverless computing and the challenges addressed in this article. We first contextualize this programming model with a description of AWS Lambda, although other platforms are equally well-suited for this purpose (e.g., Google Cloud Functions, Azure Functions, or Apache OpenWhisk). Furthermore, we focus on the dilemma of storing and sharing data across functions, then provide a high-level overview of the solution proposed in Crucial.

2.1 FaaS Computing: Value Under Restraint

AWS Lambda is a cloud service designed to run user-supplied functions, called *cloud functions*, in response to events (e.g., file uploads, message arrivals), or explicit API calls (via HTTP requests). A cloud function can be written in different target languages.[1] Before being usable, the code of the function and its dependencies are uploaded to the FaaS platform. Once deployed, the function is managed by AWS Lambda, that executes it on demand and at scale. Functions are stateless, that is, they do not keep a trace of execution from one invocation to another.

AWS Lambda, as other FaaS computing platforms, gives the advantages of rapid provisioning, high elasticity, and just-right cost: containers used to deploy a function can be launched within a few hundreds of milliseconds; they can quickly scale to match demand; and the service charges at sub-second granularity the duration of their execution. All these properties make possible to run various workloads in the cloud with minimal overhead [55, 80, 84, 88].

However, due to their lightweight nature, cloud functions are also subject to stringent resource restrictions. For instance, AWS Lambda [12] imposes a 15-minute limit per function invocation and caps memory usage to a few GBs. Similar limits are applied by other FaaS providers. In addition, while a user can execute functions concurrently, direct communication is impossible between them. As a consequence, the linear scalability in function execution is in practice only achievable for embarrassingly parallel tasks [44, 56].

Function invocations can also fail for different reasons (e.g., the function raises an exception, times out or runs out of memory). When an error occurs, AWS Lambda may automatically retry the invocation [13]. However, this requires the programmer to carefully consider such a behavior when designing the application, e.g., by ensuring call idempotence.

[1] AWS Lambda supports many languages directly (e.g., Java, Python), and any other by providing a custom runtime.

2.2 The Dilemma of Shared Data

Cloud functions are not addressable in the FaaS platform. This means that they may initiate a connection with a remote node (e.g., to fetch a web page), but they cannot listen for incoming requests. FaaS platforms currently do not support cross-functions communication. Another restriction is that cloud functions are *stateless*, that is they do not keep trace of a call from one invocation to another. These properties greatly simplify scheduling and scalability for the platform. However, they require the programmer to rely on external services for the application state [56, 102].

So far, the prevalent choice for storing data has been to rely on a disaggregated object storage such as Amazon S3. Typically, object stores have high access latency (>10 milliseconds) and deliver either limited or costly I/O performance [56, 102]. Consequently, most serverless frameworks, like PyWren [55], only allow coarse-grained operations on shared data. To alleviate this problem, some recent works [74, 80, 88] use their own in-memory storage instances. Although these systems offer low latency, they do not provide durability, nor convenient abstractions to synchronize cloud functions.

Another recurring problem is the need to ship data to code [44]. Existing serverless frameworks access data using storage services that either offer a CRUD interface or provide a limited set of data types. As a consequence, data is repeatedly transported back and forth between the cloud functions and the storage layer. This negatively impacts performance (especially for large objects) and restrains concurrency on shared data.

2.3 An Overview of Crucial

Crucial offers a simplified view of FaaS computing where cloud functions are seen as a set of cloud threads that communicate through shared state. To achieve this, the framework organizes mutable shared data in a layer of DSO. Cloud functions remotely call the methods of the objects to read/update them at fine granularity.

The DSO layer is implemented within a low-latency in-memory data store and deployed jointly with the serverless application. It delivers sub-millisecond latency—like other in-memory systems such as Redis (see Table 2)—and achieves even better throughput for complex, CPU-bound, concurrent operations (see Figure 2). Both properties, low latency and high throughput, make it an excellent substrate for mutable shared state and synchronization. Crucial also permits data to persist after the computation, ensuring their durability through replication and passivation to stable storage.

Although the idea of distributed objects is not novel, to the best of our knowledge, it has never been applied to serverless computing. Such an approach simplifies the programming of stateful applications atop serverless architectures and further closes the gap between cloud and conventional computing. The next sections describe the programming model of Crucial and its internals.

3 USING CRUCIAL

This section details the programming interface of Crucial and illustrates it with several applications. We also present a methodology to port a conventional single-machine application to serverless with the help of our framework.

3.1 Programming Model

The programming model of Crucial is object-based and can be integrated with any object-oriented programming language. As Java is the language supported in our implementation, the following description considers its jargon.

Table 1. Programming Abstractions

Abstraction	Description
`CloudThread`	Cloud functions are invoked like threads.
`ServerlessExecutorService`	A simple executor service for task groups and distributed parallel *for*s.
Shared objects	Linearizable (wait-free) distributed objects (e.g., `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, and `Map`).
Synchronization objects	Shared objects providing primitives for thread synchronization (e.g., `Future`, `Semaphore`, and `CyclicBarrier`).
`@Shared`	User-defined shared objects. Object methods run on the DSO servers, allowing fine-grained updates and aggregates (e.g., `.add()`, `.update()`, and `.merge()`).
Data persistence	Long-lived shared objects are replicated. Persistence may be activated with `@Shared(persistence=true)`.

Overall, a Crucial program is strongly similar to a regular multi-threaded, object-oriented Java one, besides some additional annotations and constructs. Table 1 summarizes the key abstractions available to the programmer that are detailed hereafter.

Cloud threads. A `CloudThread` is the smallest unit of computation in Crucial. Semantically, this class is similar to a `Thread` in conventional concurrent computing. To write an application, each task is defined as a `Runnable` and passed to a `CloudThread` that executes it. The `CloudThread` class hides from the programmer the execution details of accessing the underlying FaaS platform. This enables access transparency to remote resources [38].

Serverless executor service. The `ServerlessExecutorService` class may be used to execute both `Runnable` and `Callable` instances in the cloud. This class implements the `ExecutorService` interface, allowing the submission of individual tasks and fork-join parallel constructs (`invokeAll`). The full expressiveness of the original JDK interface is retained. In addition, this executor also includes a distributed parallel *for* to run n iterations of a loop across m workers. To use this feature, the user specifies the in-loop code (through a functional interface), the boundaries for the iteration index, and the number of workers m.

State handling. Crucial includes a library of base shared objects to support mutable shared data across cloud threads. The library consists of common objects such as integers, counters, maps, lists, and arrays. These objects are *wait-free* and *linearizable* [67]. This means that each method invocation terminates after a finite number of steps (despite concurrent accesses), and that concurrent method invocations behave as if they were executed by a single thread. Crucial also gives programmers the ability to craft their own custom shared objects by decorating a field declaration with the `@Shared` annotation. Annotated objects become globally accessible by any thread. Crucial refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`. Our framework supports distributed references, permitting a reference to cross the boundaries of a cloud thread. This feature helps to preserve the simplicity of multi-threaded programming in Crucial.

Data Persistence. Shared objects in Crucial can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and only exist during the application lifetime. Once the application finishes, they are discarded. Ephemeral objects can be lost, e.g., in the event of a server failure in the DSO layer, since the cost of making them fault-tolerant outweighs the benefits of their short-term availability [62]. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. Persistent objects outlive the application lifetime and are only removed from storage by an explicit call.

Synchronization. Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or **bulk synchronous parallelism** (**BSP**) [44, 56]. To provide fine-grained coordination of cloud threads, Crucial offers several primitives such as cyclic barriers and semaphores. These coordination primitives are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for cloud functions that is non-existent as of today.

```java
public class PiEstimator implements Runnable {
  private final static long ITERATIONS = 100_000_000;
  private Random rand = new Random();
  @Shared(key="counter")
  AtomicLong counter = new AtomicLong(0);

  public void run() {
    long count = 0;
    double x, y;
    for (long i = 0L; i < ITERATIONS; i++) {
      x = rand.nextDouble();
      y = rand.nextDouble();
      if (x * x + y * y <= 1.0) count++;
    }
    counter.addAndGet(count);
  }
}

List<Thread> threads = new ArrayList<>(N_THREADS);
for (int i = 0; i < N_THREADS; i++) {
  threads.add(new CloudThread(new PiEstimator()));
}
threads.forEach(Thread::start);
threads.forEach(Thread::join);
double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 1. Monte Carlo simulation to approximate π.

```java
ExecutorService se = new ServerlessExecutorService();
List<Callable> tasks = IntStream.range(0, N_THREADS)
      .mapToObj(i -> Executors.callable(new PiEstimator())).collect(Collectors.toList());
se.invokeAll(tasks);
```

Listing 2. Using the `ServerlessExecutorService` to perform a Monte Carlo simulation.

```java
public class Mandelbrot implements Serializable {
  @Shared(key = "mandelbrotImage")
  private MandelbrotImage image = new MandelbrotImage();

  private static int[] computeRow(int row, int width, int height, int maxIters) {...}

  private void compute() {
    image.init(COLUMNS, ROWS);
    ServerlessExecutorService se = new ServerlessExecutorService();
    se.invokeIterativeTask(row -> image.setRowColor(row, computeRow(row, COLUMNS, ROWS, MAX_INTERNAL_ITERS)), N_TASKS, 0, ROWS);
    se.shutdown();
  }
}
```

Listing 3. Mandelbrot set computation in a distributed parallel *for*.

3.2 Sample Applications

Listing 1 presents an application implemented with Crucial. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. It draws a large number of random points and computes how many of them fall in the circle enclosed by the unit square. The ratio of points falling in the circle converges with the number of trials toward $\pi/4$ (line 25).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1 – 17). To parallelize its execution, lines 23 and 24 run a fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 5). This counter maintains the total number of points falling into the circle, which serves to approximate π. It is updated by the threads concurrently using the `addAndGet` method (line 15).

The previous fork-join pattern can also be implemented using the `ServerlessExecutor Service`. In this case, we simply replace lines 19 – 24 in Listing 1 with the content of Listing 2.

A second application is shown in Listing 3. This program outputs an image approximating the Mandelbrot set (a subset of $\mathbb{C}$) with a gradient of colors. The output image is stored in a Crucial shared object (line 3). To create the image, the application computes the color of each pixel (line 5). The color indicates when the pixel escaped from the Mandelbrot set (after a bounded number of iterations). The rows of the image are processed in parallel, using the `invokeIterativeTask` method of the `ServerlessExecutorService` class. As seen at line 10, this method takes as input a functional interface (`IterativeTask`) and three integers. The interface defines the function to apply on the index of the *for* loop. The integers define respectively the number of tasks among which to distribute the iterations, and the boundaries of these iterations (`fromInclusive`, `toExclusive`).

This second example illustrates the expressiveness and convenience of our framework. In particular, as in multi-threaded programming, Crucial allows to define concurrent tasks with lambda expressions and pass them shared variables defined in the encompassing class.

3.3 Portage to Serverless

The previous sections detail the programming interface of Crucial and illustrate it with base applications. In this section, we turn our attention to the problem of porting existing applications to serverless. We first explain the benefits an application may have from a port to serverless and a methodology to achieve it. Furthermore, we present the limitations of this methodology and how the programmer can overcome them. Section 6.4 evaluates the successful application of this methodology to port Smile [65], a state-of-the-art machine learning library.

Benefits & Target applications. Crucial can be used not only to program serverless-native applications, but also to port existing single-machine applications to serverless. Successfully porting an application comes with several incentives; namely the ability to *(i)* access on-demand computing resources; *(ii)* scale these resources dynamically; and *(iii)* benefit from a fine-grained pricing for their usage. To match the programming model of Crucial, Java applications that can benefit from a portage should be multi-threaded. Moreover, as with other parallel programming frameworks (e.g., MPI [91] or MapReduce [29]), they should be inherently parallel.

Methodology. Crucial allows to port an existing Java multi-threaded application to serverless with low effort. To this end, the following steps should be taken: (1) The programmer replaces the `ExecutorService` or `Thread` instances with their Crucial counterparts, as listed in Table 1. (2) The programmer makes `Serializable` each immutable object passed between cloud threads. (3) The programmer substitutes the concurrent mutable objects shared by threads with the equivalent ones provided by the DSO layer. For example, an instance of `java.util.atomic.Atomic Boolean` is replaced with `org.crucial.dso.AtomicBoolean`. (4) Regarding synchronization primitives, the programmer must transform them into distributed objects. As an example, a cyclic barrier can be replaced with `org.crucial.dso.CyclicBarrier`, an implementation based internally on a monitor. Another option is `org.crucial.dso.ScalableCyclicBarrier`, that implements the collective described in [46]. (5) If the application uses the `synchronized` keyword, some rewriting is necessary. Recall that this keyword is specific to the Java language and allows to use any (non-primitive) object as a monitor [47]. CRUCIAL does not support the `synchronized` keyword out of the box since it would require modifying the JVM. Two solutions are offered to the programmer: *(i)* create a monitor object in DSO and use it where appropriate; or *(ii)* create a method for the object used as a monitor that contains all the code in the `synchronized{..}` block. Then, this object is annotated as `@Shared` in the application, and the method called where appropriate. The first solution is simple, but it might not be the most efficient since it requires to move data back and forth the cloud threads that use the monitor. The second solution needs rewriting part of the original application. However, it is more in line with the object-oriented approach in CRUCIAL, where an operation updating a shared object is accessible through a (linearizable) method, and it may perform better.

```
 1 public class WordCount {
 2   private Document document = new Document(LOCATION);
 3   private String word = "serverless";
 4
 5   private void compute() {
 6     AtomicLong counter = new AtomicLong("wordcount");
 7     ServerlessExecutorService se = new ServerlessExecutorService();
 8     se.invokeIterativeTask(i -> counter.addAndGet(countWords(word, document.split(i))), N_TASKS, 0, N_TASKS);
 9   }
10 }
```

Listing 4. Parallel word count.

Limitations & Solutions. The above methodology works for most applications, yet it has limitations. First, some threading features are not available in the framework—e.g., signaling a cloud thread. Second, CRUCIAL does not natively support arrays (e.g., `T[] tab`). Indeed, recall that the Java language offers native methods to manipulate such data types. For instance, calling `tab[i]=x` assigns the value (or reference) x to `tab[i]`. Transforming a native call is not possible with just annotations.[2] The solution to these two problems is to rewrite the application appropriately, as in the case of `synchronized`.

Another issue is related to data locality. Typically, a multi-threaded application initializes shared data in the main thread and then makes it accessible to other threads for computation. Porting such a programming pattern to FaaS implies that data is initialized at the machine starting up the application, then serialized to be accessible elsewhere; this is very inefficient. Instead, a better approach is to pass a distributed reference that is lazily de-referenced by the thread. To illustrate this point, consider Listing 4, which counts the number of occurrences of the word "`serverless`" in a document. The application first constructs a reference to the document (line 2). Then, the document is split into chunks. For each chunk, the number of occurrences of the word is counted by a cloud thread (line 8). The results are then aggregated in the shared counter "`wordcount`". Reading the document in full at line 2 and serializing it to construct the chunks is inefficient. Instead, the application should send a distributed reference to the cloud threads at line 8. Then, upon calling `split`, the chunks are created on each thread by fetching the content from remote storage.

[2] It is however possible with bytecode manipulation tools (e.g., [22]).

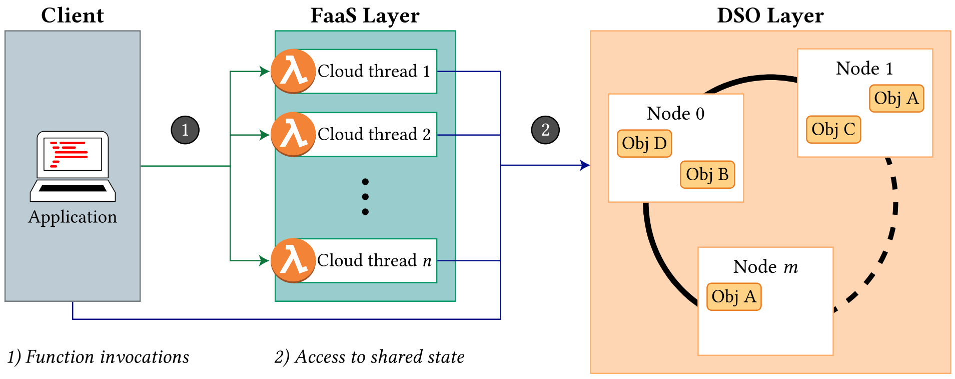

Fig. 1. Crucial's overall architecture. A client application runs a set of cloud threads in the FaaS layer. The cloud threads and the client have access to the shared state stored in the DSO layer.

4 SYSTEM DESIGN

Figure 1 presents the overall architecture of Crucial. In what follows, we detail its components and describe the lifecycle of an application in our system.

Crucial encompasses three main components (from left to right in Figure 1): the client application; the FaaS computing layer that runs the cloud threads; and the DSO layer that stores the shared objects. A client application differs from a regular JVM process in two aspects: threads are executed as cloud functions, and they access shared data using the DSO layer. Moreover, Crucial applications may also rely on external cloud services, such as object storage to fetch input data (not modeled in Figure 1).

4.1 The Distributed Shared Objects Layer

Each object in the DSO layer is uniquely identified by a reference. Fine-grained updates to the shared state are implemented as methods of these objects. Given an object of type T, the reference to this object is (T, k), where k is either the name of the annotated object field or the value of the parameter *key* in the annotation `@Shared(key=k)`. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method.

Crucial constructs the DSO layer using consistent hashing [59], similarly to Cassandra [63]. Each storage node knows the full storage layer membership and thus the mapping from data to node. The location of a shared object o is determined by hashing the reference (T, k) of o. This offers the following usual benefits: (1) No broadcast is necessary to locate an object; (2) Disjoint-access parallelism [53] can be exploited; and (3) Service interruption is minimal in the event of server addition and removal. The latter property is useful for persistent objects, as detailed next.

Persistence. One interesting aspect of Crucial is that it can ensure durability of the shared state. This property is appealing, for instance, to support the different phases of a machine learning workflow (training and inference). Objects marked as persistent are replicated *rf* (replication factor) times in the DSO layer. They reside in memory to ensure sub-millisecond read/write latency and can be passivated to stable storage using standard mechanisms (marshalling). When a cloud thread accesses a shared object, it contacts one of the server nodes. The operation is then forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to the caller. Notice that for ephemeral—non-persistent—objects, *rf* is 1.

Consistency. Crucial provides linearizable objects and programmers can reason about interleaving as in the shared-memory case. This greatly simplifies the writing of stateful serverless applications. For persistent objects, consistency across replicas is maintained with the help of **state machine replication** (**SMR**) [87]. To handle membership changes, the DSO layer relies on a variation of virtual synchrony [27]. Virtual synchrony provides a totally-ordered set of views to the server nodes. In a given view, for some object x, the operations accessing x are sent using total order multicast. The replicas of x deliver these operations in a total order and apply them on their local copy of x according to this order. A distinct replica (primary) is in charge of sending back the result to the caller. When a membership change occurs, the nodes re-balance data according to the new view. Appendix A provides a full pseudo-code of this construction together with a proof of correctness.

4.2 Fast Aggregates Through Remote Procedure Call

As indicated in Section 2, stateful applications aggregate and combine small granules of data (e.g., the training phase of a ML algorithm). Unfortunately, cloud functions are not network-addressable and run separate from data. As a consequence, these applications are routinely left with no other choice but to "ship data to code". This is known as one of the biggest downsides of FaaS platforms [44].

To illustrate this point, consider an `AllReduce` operation where N cloud functions need to aggregate their results by applying some commutative and associative operator f (e.g., a sum). To achieve this, each function first writes its local result in the storage layer. Then, the functions await that their peers do the same, fetch the N results, and apply f sequentially. This algorithm is expensive and entails a communication cost of N^2 messages with the storage layer.

Crucial fully resolves this anti-pattern with minimal efforts from the programmer. Complex computations are implemented as object methods in DSO and called by the cloud functions where appropriate. Going back to the above example, each function simply calls $f(r)$ on the shared object, where r is its local result. This is for instance the case at line 8 in Listing 4 with the method `counter.addAndGet`. With this approach, communication complexity is reduced to $O(N)$ messages with the storage layer.

We exploit this key feature of Crucial in our serverless implementation of several ML algorithms (e.g., k-means, linear regression, and random forest). Its performance benefits are detailed in Section 6.2.

4.3 Lifecycle of an Application

The lifecycle of a Crucial application is similar to that of a standard multi-threaded Java one. Every time a `CloudThread` is started, a Java thread (i.e., an instance of `java.lang.Thread`) is spawned on the client. This thread pushes the `Runnable` code attached to the `CloudThread` to a generic function in the FaaS platform. Then, it waits for the result of the computation before it returns.

Accesses to some shared object of type `T` at cloud threads (or at the client) are mediated by a proxy. This proxy is instantiated when a call to "`new T()`" occurs, and either the newly created object of type `T` belongs to Crucial's library, or it is tagged `@Shared`. As an example, consider the counter used in Listing 1. When an instance of `PiEstimator` is spawned, the field `counter` is created. The "`new`" statement is intercepted and a local proxy for the counter is instantiated to mediate calls to the remote object hosted in the DSO layer. If this object does not exist in the DSO layer, it is instantiated using the constructor defined at line 5. From thereon, any call to `addAndGet` (line 15) is pushed to the DSO layer. These calls are delivered in total order to the object replicas where they are applied before sending back a response value to the caller.

The Java thread remains blocked until the cloud function terminates. Such a behavior gives cloud threads the appearance of conventional threads; minimizing code changes and allowing the use of the `join()` method at the client to establish synchronization points (e.g., fork/join pattern). It must be noted, however, that as cloud functions cannot be canceled or paused, the analogy is not complete. If any failure occurs in a remote cloud function, the error is propagated back to the client application for further processing.

The case of the `ServerlessExecutorService` builds on the same idea as `CloudThread`. A standard Java thread pool is used internally to manage the execution of all tasks. In the case of a callable task, the result is accessible to the caller in a `Future` object.

4.4 Fault Tolerance

Fault tolerance in Crucial is based on the disaggregation of the compute and storage layers. On one hand, writes to DSO can be made durable with the help of data replication. In such a case, Crucial tolerates the joint failure of up to $rf - 1$ servers.[3] On the other hand, Crucial offers the same fault-tolerance semantics in the compute layer as the underlying FaaS platform. In AWS Lambda, this means that any failed cloud thread can be re-started and re-executed with the exact same input. Thanks to the cloud thread abstraction, Crucial allows full control over the retry system. For instance, the user may configure how many retries are allowed and/or the time between them. If retries are permitted, the programmer should ensure that the re-execution is sound (e.g., it is idempotent). Fortunately, atomic writes in the DSO layer make this task easy to achieve. Considering the k-means example depicted in Listing 5 (or another iterative algorithm), it simply consists of sharing an iteration counter (line 6). When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

5 IMPLEMENTATION

The implementation of Crucial is open source and available online [6]. It consists of around 10K SLOC, including scripts to deploy and run Crucial applications in the cloud. The DSO layer is written atop the Infinispan in-memory data grid [69] as a partial rewrite of the Creson project [96].

A Crucial application is written in Java and uses Apache Maven to compile and manage its dependencies. It employs the abstractions listed in Table 1 and has access to scripts that automate its deployment and execution in the cloud.

To run cloud threads, our prototype implementation relies on AWS Lambda. Lambda functions are deployed with the help of a Maven plugin [5] and invoked via the AWS Java SDK. To control the replay mechanism, calls to Lambda are synchronous. The adherence of Crucial to Lambda is limited and the framework can execute atop a different FaaS platform with a few changes. In Section 7.1, we discuss this platform dependency.

The `ServerlessExecutorService` implements the base `ExecutorService` interface. It accepts `Callable` objects and task collections. The invocation of a `Callable` returns a (local) `Future` object. This future is completed once a response from AWS Lambda is received. For `Runnable` tasks, the response is empty unless an error occurs. In that case, the system interprets it and throws an exception at the client machine, referencing the cause.

To create a distributed parallel *for*, the programmer uses an instance of `IterativeTask` (as illustrated at line 10 in Listing 3). This functional interface is similar to `java.util.function.Consumer`, but limited to iteration indexes (i.e., the input parameter must be an integer). Internally, the iterative task creates a collection of `Callable` objects. In our current prototype, the scheduling is static and based on the number of workers and tasks given in parameter.

[3] Synchronization objects (see Table 1) are not replicated. This is not an important issue due to their ephemeral nature.

When an AWS Lambda function is invoked, it receives a user-defined `Runnable` (or `Callable`) object. The object and its content are marshalled and shipped to the remote machine, where they are re-created. Initialization parameters can be given to the constructor. As pointed out in Section 3.1, a distributed reference is sent in lieu of a shared object.

Proxies for the shared objects are waved into the code of the client application using AspectJ [60]. In the case of user-defined objects, the aspects are applied to the annotated fields (see Section 3.1). Such objects must be serializable and they should contain an empty constructor (similarly to a JavaBean). The `jar` archive containing the definition of the objects is uploaded to the DSO servers where it is dynamically loaded.

Synchronization objects (e.g., barriers, semaphores, and futures) follow the structure of their Java counterparts. They rely internally on Java monitors. When a client performs a call to a remote object, it remains blocked until the request responds. The server processes the operation with a designated thread. During the method invocation, that thread may suspend itself through a `wait` call on the object until another thread awakes it.

State machine replication (**SMR**) is implemented using Infinispan's interceptor API. This API enables the execution of custom code during the processing of a data store operation. It follows the visitor pattern as commonly found in storage systems. Infinispan [69] relies on JGroups [43] for total order multicast. The current implementation uses Skeen's algorithm [19].

In our prototype, the deployment of the storage layer is explicitly managed (like, e.g., AWS ElastiCache). Automatic provisioning of storage resources for serverless computing remains an open issue [24, 56], with just a couple works appearing very recently in this area [62, 80].

6 EVALUATION

This section evaluates the performance of Crucial and its usability to program stateful serverless applications.

Outline. We first evaluate the runtime of Crucial with a series of micro-benchmarks (Section 6.1). Then, we focus on fine-grained updates to shared mutable data (Section 6.2) and fine-grained synchronization (Section 6.3). Furthermore, we detail the (partial) portage to serverless of the Smile library [65] (Section 6.4). Finally, we analyze the usability of our framework when writing (or porting) applications (Section 6.5).

Goal & scope. The core objective of this evaluation is to understand the benefits of Crucial to program applications for serverless. To this end, we distinguish two types of applications: serverless-native and ported applications. Serverless-native applications are those written from scratch for a FaaS infrastructure. Ported applications are the ones that were initially single-machine applications and were later modified to execute atop FaaS. For both types of applications, our evaluation campaign aims at providing answers to the following questions:

— *How easy is it to program with Crucial?* In addressing this question, we specifically focus on the following applications: ML (Section 6.2.1), data analytics (Section 6.3.1), and synchronization tasks (Section 6.3.2). These applications are parallel and stateful, that is they contain parallel components that need to update a shared state and synchronize to make progress.
— *Do applications programmed with Crucial benefit from the capabilities of serverless (e.g., scalability and on-demand pricing)?* (Section 6.4.2).
— *How efficient is an application programmed with Crucial?* For serverless-native applications, we compare Crucial to PyWren, a state-of-the-art solution for serverless programming (Section 6.2.3). We also make a comparison with Apache Spark, the de facto standard approach to program stateful cluster-based programs (Section 6.2.2). For ported applications, we compare Crucial to a scale-up approach, using a high-end server (Section 6.4).

Table 2. Average Latency Comparison—1 KB Payload

	PUT	GET
S3	34, 868μs	23, 072μs
Redis	232μs	229μs
Infinispan	228μs	207μs
Crucial	231μs	229μs
Crucial ($rf = 2$)	512μs	505μs

— *How costly is Crucial with respect to other solutions?* (Section 6.5.3) Here we are interested both in the programming effort to code a serverless application and its monetary cost when running atop a FaaS platform. Again, answers are provided for both serverless-native and ported applications.

Experimental setup. All the experiments are conducted in **Amazon Web Services** (**AWS**), within a **Virtual Private Cloud** (**VPC**) located in the `us-east-1` region. Unless otherwise specified, we use `r5.2xlarge` EC2 instances for the DSO layer and 3 GB AWS Lambda functions. Experiments with concurrency over 300 cloud threads are run outside the VPC due to service limitations.

The code of the experiments presented in this section is available online [6].

6.1 Micro-benchmarks

As depicted in Figure 1, the runtime of Crucial consists of two components: an FaaS platform and the DSO layer. In this section, we evaluate the performance of this runtime across several micro-benchmarks.

6.1.1 Latency. Table 2 compares the latency to access a 1 KB object sequentially in Crucial (DSO), Redis, Infinispan, and S3. We chose Redis because it is a popular key-value store available on almost all cloud platforms, and it has been extensively used as storage substrate in prior serverless systems [55, 62, 80]. Each function performs 30 K operations and we report the average access latency. In Table 2, Crucial exhibits a performance similar to other in-memory systems. In particular, it is an order of magnitude faster than S3. This table also depicts the effect of object replication. When data is replicated, SMR adds an extra round-trip, doubling the latency perceived at a client. The number of replicas does not affect this behavior, as shown in the next experiment.

6.1.2 Throughput. We measure the throughput of Crucial and compare it against Redis. For an accurate picture, replication is enabled in both systems to capture their performance under scenarios of high data availability and durability.

In this experiment, 200 cloud threads access 800 shared objects during 30 seconds. The objects are chosen at random. Each object stores an integer offering basic arithmetic operations. We consider simple and complex operations. The simple operation is a multiplication. The complex one is the sequential execution of 10 K multiplications. In Redis, these operations require several commands which run as Lua scripts for both consistency and performance.

To replicate data, Redis uses a master-based mechanism. By default, replication is *asynchronous*, so the master does not wait for a command to be processed by the replicas. Consequently, clients can observe stale data. In our experiment, to minimize inconsistencies and offer guarantees closer to Crucial, functions issue a `WAIT` command after each write [82]. This command flushes the pending updates to the replicas before it returns.

We compare the average throughput of the two systems when the **replication factor** (rf) of a datum varies as follows: ($rf = 1$) Both Crucial and Redis (2 shards with no replicas) are deployed

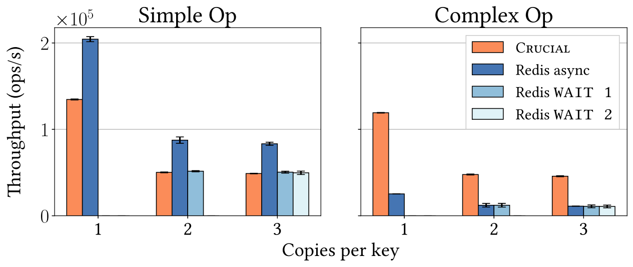

Fig. 2. Operations per second performed in Crucial and Redis (with and without replication). Cloud threads access uniformly at random 800 different keys/objects.

over a 2-node cluster; ($rf = 2$) In the same 2-node cluster, Redis nows uses one master and one replica; ($rf = 3$) We add a third node to the cluster and Redis employs one master and two replicas. In Figure 2, "Redis `WAIT` r" indicates that r is the number of synchronously replicated copies of shared objects.

The experimental results reported in Figure 2 show that Crucial is not sensitive to the complexity of operations. Redis is 50% faster for simple operations because its implementation is optimized and written in C. However, for complex operations, Crucial is almost five times better than Redis. Again, implementation-specific details are responsible for this behavior: While Redis is single-threaded, and thus concurrent calls to the Lua scripts run sequentially, Crucial benefits from disjoint-access parallelism [53]. When objects are replicated, the comparison is similar. In particular, Figure 2 shows that Crucial and Redis have close performance when Redis operates in synchronous mode.

This experiment also verifies that the performance of Crucial is not sensitive to the number of replicas. Indeed, the throughput in Figure 2 is roughly equivalent for all values of $rf \geq 2$. This comes from the fact that Crucial requires a single RTT to propagate an operation to the replicas.

6.1.3 Parallelism. We first evaluate our framework with the Monte Carlo simulation presented in Listing 1. This algorithm is embarrassingly parallel, relying on a single shared object (a counter). The simulation runs with 1–800 cloud threads and we track the total number of points computed per second. The results, presented in Figure 3(a), show that our system scales linearly and that it exhibits a 512× speedup with 800 threads.

We further evaluate the parallelism of Crucial with the code in Listing 3. This second experiment computes a 30K×30K projection of the Mandelbrot set, with (at most) $1,000$ iterations per pixel. As shown in Figure 3(b), the completion time decreases from 150 seconds with 10 threads to 14.5 seconds with 200 threads: a speedup factor of 10.2× over the 10-thread execution. This super-linear speedup is due to the skew in the coarse-grained row partitioning of the image. It also underlines a key benefit of Crucial. If this task is run on a cluster, the cluster is billed for the entire job duration, even if some of its resources are idle. Running atop serverless resources, this implementation ensures instead that row-dependent tasks are billed for their exact duration.

Takeaways. The DSO layer of Crucial is on par with existing in-memory data stores in terms of latency and throughput. For complex operations, it significantly outperforms Redis due to data access parallelism. Crucial scales linearly to hundreds of cloud threads. Applications written with the framework benefit from the serverless provisioning and billing model to match irregularities in parallel tasks.

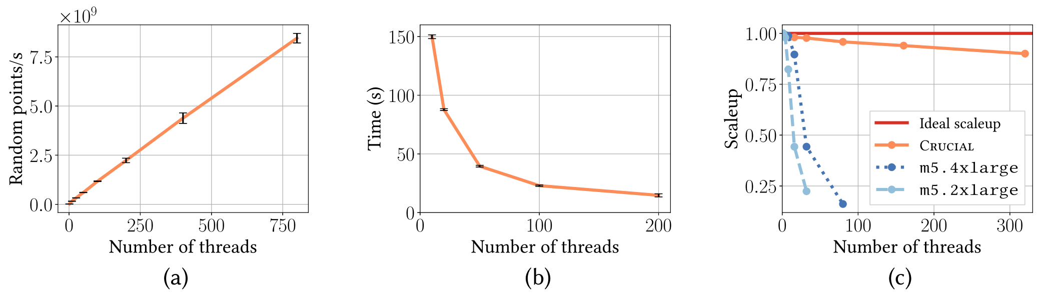

Fig. 3. (a) Scalability of a Monte Carlo simulation to approximate π. Crucial reaches 8.4 billion random points per second with 800 threads. (b) Scalability of a Mandelbrot computation with Crucial. (c) Scalability of the k-means clustering algorithm with Crucial versus single-machine multi-threading.

6.2 Fine-grained State Management

This section shows that Crucial is efficient for parallel applications that access shared state at fine granularity. We detail the implementation of two ML algorithms in the framework. These algorithms are evaluated against a single-machine solution, as well as two state-of-the-art frameworks for cluster computing (Apache Spark) and FaaS-based computation (PyWren).

6.2.1 A Serverless k-means. Listing 5 details the code of a k-means clustering algorithm written with Crucial. This program computes k clusters from a set of points across a fixed number of iterations, or until some convergence criterion is met (line 21). The algorithm is iterative, with recurring synchronization points (line 19), and it uses a small mutable shared state. Listing 5 relies on shared objects for the convergence criterion (line 4), the centroids (line 8), and a synchronization object to coordinate the iterations (line 2). At each iteration, the algorithm needs to update both the centroids and the criterion. The corresponding method calls (lines 14, 17, and 18) are executed remotely in DSO.

```java
1  public class KMeans implements Runnable {
2    private CyclicBarrier barrier = new CyclicBarrier();
3    @Shared(key = "delta")
4    private GlobalDelta globalDelta = new GlobalDelta();
5    @Shared(key = "iterations")
6    private AtomicInteger globalIterCount = new AtomicInteger();
7    // Wraps a list of @Shared centroids
8    private GlobalCentroids centroids = new GlobalCentroids();
9
10   public void run() {
11     loadDatasetFragment();
12     int iterCount = globalIterCount.intValue();
13     do {
14       correctCentroids = globalCentroids.getCorrectCoordinates();
15       resetLocalStructures();
16       localDelta = computeClusters();
17       globalDelta.update(localDelta);
18       centroids.update(localCentroids, localSizes);
19       barrier.await();
20       globalIterCount.compareAndSet(iterCount, iterCount++);
21     } while (iterCount < maxIterations && !endCondition());
22   }
23 }
```

Listing 5. k-means implementation with Crucial.

Figure 3(c) compares the scalability of Crucial against two EC2 instances: `m5.2xlarge` and `m5.4xlarge`, with 8 and 16 vCPUs, respectively. In this experiment, the input increases proportionally to the number of threads. We measure the *scale-up* computed with respect to that fact:

scale-up = T_1/T_n, where T_1 is the execution time of Listing 5 with one thread, and T_n when using n threads.[4] Accordingly, *scale-up* = 1 means a perfect linear scale-up, i.e., the increase in the number of threads keeps up with the increase in the workload size (top line in Figure 3(c)). The scale-up is sub-linear when *scale-up* < 1. As expected, the single-machine solution quickly degrades when the number of threads exceeds the number of cores. The solution using Crucial is within 10% of the optimum. For instance, with 160 threads, the scale-up factor is approximately 0.94. This lowers to 0.9 for 320 threads due to the overhead of creating the cloud threads.

6.2.2 Comparison with Spark. Apache Spark [104] is a state-of-the-art solution for distributed computation in a cluster. As such, it is extensively used to scale many kinds of applications in the cloud. One of them is ML training, as enabled by Spark's MLlib [71] library. Most ML algorithms are iterative and share a modest amount of state that requires per-iteration updates. Consequently, they are a perfect fit to assess the efficiency of fine-grained updates in Crucial against a state-of-the-art solution. This is the case of logistic regression and k-means clustering, which we use in this section to compare Crucial and Spark.

Setup. For this comparison, we provide equivalent CPU resources to all competitors. In detail, Crucial experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1, 792 MB and 2, 048 MB of memory for logistic regression and k-means, respectively. These values are chosen to have the optimal performance at the lowest cost (see Section 6.5.3).[5] The DSO layer runs on an `r5.2xlarge` EC2 instance. Spark experiments are run in Amazon EMR with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR terminology), each having 8 vCPUs. Spark executors are configured to utilize the maximum resources possible on each node of the cluster. To improve the fairness of our comparison, the time spent in loading the dataset from S3 and parsing it is not considered for both solutions. For Spark, the time to provision the cluster is not counted. Regarding Crucial, FaaS cold starts are also excluded from measurements due to a global barrier before starting the computation.

Dataset. The input is a 100 GB dataset generated with spark-perf [28] that contains 55.6M elements. For logistic regression, each element is labeled and contains 100 numeric features. For k-means, each element corresponds to a 100-dimensional point. The dataset has been split into 80 equal-size partitions to ensure that all partitions are small enough to fit into the function memory. Each partition has been stored as an independent file in Amazon S3.

Logistic regression. We evaluate a Crucial implementation of logistic regression against its counterpart available in Spark's MLlib [71]: `LogisticRegressionWithSGD`. A key difference between the two implementations is the management of the shared state. Each iteration, Spark broadcasts the current weight coefficients, computes, and finally aggregates the sub-gradients in a MapReduce phase. In Crucial, the weight coefficients are shared objects. Each iteration, a cloud thread retrieves the current weights, computes the sub-gradients, updates the shared objects, and synchronizes with the other threads. Once all the partial results are uploaded to the DSO layer, the weights are recomputed, and the threads proceed to the next iteration.

In Figure 4(a) and (b), we measure the running time of 100 iterations of the algorithm and the logistic loss after each iteration. Results show that the iterative phase is 18% faster in Crucial (62.3 seconds) than with Spark (75.9 seconds), and thus the algorithm converges faster. This gain

[4] In Figure 3(c), threads are AWS Lambda functions for Crucial, and standard Java threads for the EC2 instances.
[5]Starting with a configuration of 1, 792 MB, an AWS Lambda function has the equivalent to 1 full vCPU (https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html). Also, with this assigned memory, the function uses a full **Elastic Network Interface** (**ENI**) in the VPC.

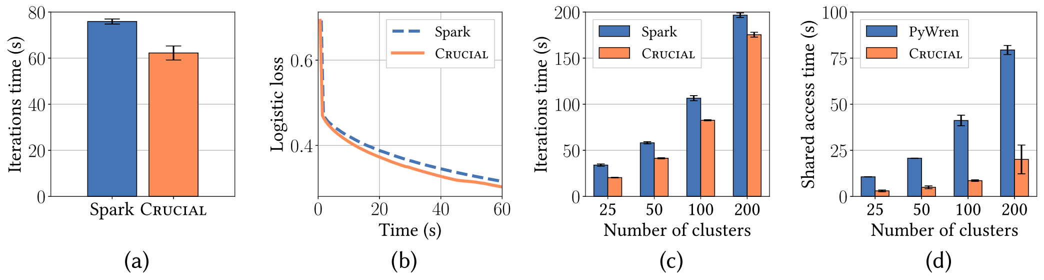

Fig. 4. Comparison of Crucial and the state-of-the-art. (a) Average logistic regression iterative phase completion time (100 iterations). (b) Logistic regression performance. (c) Average k-means iterative phase completion time (10 iterations) with varying number of clusters. (d) Average k-means shared state access time.

is explained by the fact that Crucial aggregates and combines the sub-gradients in the storage layer. On the contrary, each iteration in Spark requires a reduce phase that is costly both in terms of communication and synchronization.

k-means. We compare the k-means implementation described in Section 6.2.1 to the one in MLlib. For both systems, the centroids are initially at random positions and the input data is evenly distributed among tasks. Figure 4(c) shows the completion time of 10 iterations of the clustering algorithm. In this figure, we consider different values of k to assess the effectiveness of our solution when the size of the shared state varies. With $k = 25$, Crucial completes the 10 iterations 40% faster (20.4 seconds) than Spark (34 seconds). The time gap is less noticeable with more clusters because the time spent synchronizing functions is less representative. In other words, the iteration time becomes increasingly dominated by computation. As in the logistic regression experiment, Crucial benefits from computing centroids in the DSO layer, while Spark requires an expensive reduce phase at each iteration.

6.2.3 Comparison with PyWren. We close this section by comparing Crucial to a serverless-native state-of-the-art solution. To date, the most evaluated framework to program stateful serverless applications is PyWren [55]. Its primitives, such as `call_async` and `map` are comparable to Crucial's cloud thread and serverless executor abstractions. Our evaluation employs Lithops [7], a recent and improved version of PyWren. PyWren is a MapReduce framework. Thus, it does not natively provide advanced features for state sharing and synchronization. Therefore, following the recommendations by Jonas et al. [55], we use Redis for this task.

Setup. We employ the same application, dataset, and configuration as in the previous experiment. The two frameworks use AWS Lambda for execution. A single `r5.2xlarge` EC2 instance runs DSO for Crucial, or Redis for PyWren.

k-means. Implementing k-means above PyWren requires to store the shared state in Redis, that is the centroids and the convergence criterion. Following Jonas et al. [55], we use a Lua script to achieve this. At the end of each iteration, every function updates (atomically) the shared state by calling the script. This approach is the best solution in terms of performance. In particular, it is more efficient than using distributed locking due to the large number of commands needed for the updates. To synchronize across iterations, we use the Redis barrier covered in Section 6.3.2.

The Crucial and PyWren k-means applications are written in different languages (Java and Python, respectively). Consequently, the time spent in computation for the two applications is dissimilar. For that reason, and contrary to the comparison against Spark, Figure 4(d) does not report the completion time. Instead, this figure depicts the average time spent in accessing the shared state during the k-means execution for both Crucial and PyWren. This corresponds to the time spent inside the loop in Listing 5 (excluding line 16).

In Figure 4(d), we observe that the solution combining PyWren and Redis is always slower than Crucial. This comes from the fact that Crucial allows efficient fine-grained updates to the shared state. Such results are in line with the ones presented in Section 6.1.2.

Takeaways. The DSO layer of Crucial offers abstractions to program stateful serverless applications. DSO is not only convenient but, as our evaluation confirms, efficient. For two common machine learning tasks, Crucial is up to 40% faster than Spark, a state-of-the-art cluster-based approach, at comparable resource usage. It is also faster than a solution using jointly PyWren, a well-known serverless framework, and the Redis data store.

6.3 Fine-grained Synchronization

This section analyzes the capabilities of Crucial to coordinate cloud functions. We evaluate the synchronization primitives available in the framework and compare them to state-of-the-art solutions. We then demonstrate the use of Crucial to solve complex coordination tasks by considering a traditional concurrent programming problem.

6.3.1 Synchronizing a Map Phase. Many algorithms require synchronization at various stages. In MapReduce [29], this happens between the map and reduce phases, and it is known as shuffle. Shuffling ensures that the reduce phase starts when all the appropriate data was output in the preceding map phase. Shuffling the map output is a costly operation in MapReduce, even if the reduce phase is short. For that reason, when data is small and the reduction operation simple, it is better to skip the reduce phase and instead aggregate the map output directly in the storage layer [30]. Crucial allows to easily implement this approach.

In what follows, we compare different techniques to synchronize cloud functions at the end of a map. Namely, we compare (1) the original solution in PyWren, based on polling S3; (2) the same mechanism but using the Infinispan in-memory key-value data store; (3) the use of Amazon SQS, as proposed in some recent works (e.g., [61]); and (4) two techniques based on the `Future` object available in Crucial. The first solution outputs a future object per function, then runs the reduce phase. The second aggregates all the results directly in the DSO layer (called auto-reduce).

We compare the above five techniques by running back-to-back the Monte Carlo simulation in Listing 1. The experiment employs 100 cloud functions, each doing 100 M iterations. During a run, we measure the time spent in synchronizing the functions. On average, this accounts for 23% of the total time.

Figure 5(a) presents the results of our comparison. Using Amazon S3 proves to be slow, and it exhibits high variability—some experiments being far slower than others. This is explained by the combination of high access latency, eventual consistency, and the polling-based mechanism. The results improve with Infinispan, but being still based on polling, the approach induces a noticeable overhead. Using Amazon SQS is the slowest approach of all. It needs a polling mechanism that actively reads messages from the remote queue. The solution based on `Future` objects allows to immediately respond when the results are available. This reduces the number of connections necessary to fetch the result and thus translates into faster synchronization. When the map output is directly aggregated in DSO, Crucial achieves even better performance, being twice as fast as the polling approach atop S3.

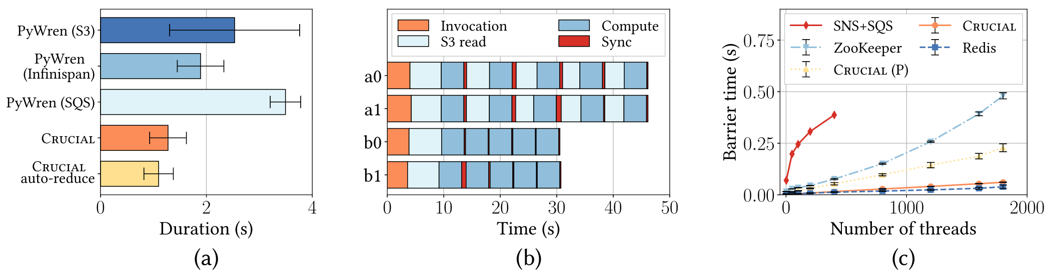

Fig. 5. (a) Synchronizing a map phase in MapReduce with PyWren, Amazon SQS, and Crucial. (b) Performance breakdown of an iterative task using either multiple stages (a0/a1), or a single stage with a Crucial barrier (b0/b1). (c) Average time threads spend waiting on a barrier.

6.3.2 Synchronization Primitives. Cloud functions need to coordinate when executing parallel tasks. This section evaluates some of the synchronization primitives available in Crucial to this end.

For starters, we study the performance of a barrier when executing an iterative task. In Figure 5(b), we depict a breakdown of the time spent in the phases of each iteration (Invocation, S3 read, Compute, and Sync). The results are reported for 2 cloud functions out of 10—the other functions behave similarly.

The breakdown in Figure 5(b) considers two approaches. The first one launches a new stage of functions (a0 and a1) at each iteration that do not use the barrier primitive. The second launches a single stage of functions (b0 and b1) that run all the iterations and use the barrier primitive to synchronize. In the first case, data must be fetched from storage at each iteration, while in the second approach it is only fetched once. Overall, Figure 5(b) shows that this latter mechanism is clearly faster. In particular, the total time spent in coordinating the functions is lower when the barrier is used (Sync).

Figure 5(c) draws a comparison between two barrier objects available in Crucial and several state-of-the-art solutions. More precisely, the figure reports the performance of the following approaches: (1) a pure cloud-based barrier, which combines Amazon SNS and SQS services to notify the functions; (2) a ZooKeeper cyclic barrier based on the official double barrier [37] in a 3-node cluster; (3) a non-resilient barrier using the Redis `BLPOP` command ("blocking left pop") on a single server; (4) the default cyclic barrier available in Crucial, with a single server instance; and (5) a resilient, poll-based (P) barrier implementing the algorithm in [46] on a 3-node cluster with replication.

To draw this comparison, we measure the time needed to exit 1, 000 barriers back-to-back for each approach. An experiment is run 10 times. Figure 5(c) reports the average time to cross a single barrier for a varying number of cloud functions.

The results in Figure 5(c) show that the single server solutions, namely Crucial and Redis, are the fastest approaches. With 1, 800 threads, these barriers are passed after waiting 68 ms on average. The fault-tolerant barriers (Crucial (P) and ZooKeeper) create more contention, incurring a performance penalty when the level of parallelism increases. With the same number of threads, passing the poll-based barrier of Crucial takes 287 milliseconds on average. ZooKeeper requires twice that time. The solution using Amazon SNS and SQS is an order of magnitude slower than the rest.

It is worth noting the difference between the programming complexity of each barrier. Both barriers implemented in Crucial take around 30 lines of basic Java code. The solution using Redis has the same length, but it requires a proper management of the connections to the data store

Table 3. Santa Claus Problem's Completion Time (in Seconds) on a Single Machine vs. Crucial

	Threads	Threads + DSO	Crucial
p50	20.15	20.91	21.97
p99	21.09	22.03	22.66
Overhead	–	3.8%	9.0%

as well as the manual creation/deletion of shared keys. ZooKeeper substantially increases code complexity, as programmers need to deal with a file-system-like interface and carefully set watches, requiring around 90 lines of code. Finally, the SNS and SQS approach is the most involved technique of all, necessitating 150 lines of code and the use of two complex cloud service APIs.

6.3.3 A Concurrency Problem. Thanks to its coordination capabilities, Crucial can be used to solve complex concurrency problems. To demonstrate this feature, we consider the Santa Claus problem [98]. This problem is a concurrent programming exercise in the vein of the dining philosophers, where processes need to coordinate in order to make progress. Common solutions employ semaphores and barriers, while others, actors [18].

Problem. The Santa Claus problem involves three sets of *entities*: Santa Claus, nine reindeer, and a group of elves. The elves work at the workshop until they encounter an issue that needs Santa's attention. The reindeer are on vacation until Christmas eve, when they gather at the stable. Santa Claus sleeps, and can only be awakened by either a group of three elves to solve a workshop issue, or by the reindeer to go delivering presents. In the first case, Santa solves the issues, and the elves go back to work. In the second, Santa and the reindeer execute the delivery. The reindeer have priority if the two situations above occur concurrently.

Solution. Let us now explain the design of a common solution to this problem [18]. Each entity (Santa, elves, and reindeer) is a thread. They communicate using two types of synchronization primitives: *groups* and *gates*. Elves and reindeer try to join a group when they encounter a problem or Christmas is coming, respectively. When a group is full—either including three elves or nine reindeer—the entities enter a room and notify Santa. A room has two gate objects: one for entering and one for exiting. Gates act like barriers, and all the entities in the group wait for Santa to open the gate. When Santa is notified, he looks whether a group is full (either of reindeer or elves, prioritizing reindeer). He then opens the gate and solves the workshop issues or goes delivering presents. This last operation is repeated until enough deliveries, or *epochs*, have occurred.

We implemented the above solution in three flavors. The first one uses **plain old Java objects** (**POJOs**), where groups and gates are monitors and the entities are threads. Our second variation is a refinement of this base approach, where the synchronization objects are stored in the DSO layer. The conversion is straightforward using the API presented in Section 3. In particular, the code of the objects used in the POJO solution is unchanged. Only adding the `@Shared` annotation is required. The last refinement consists in using Crucial's cloud threads instead of the Java ones.

Evaluation. We consider an instance of the problem with 10 elves, 9 reindeer, and 15 *deliveries* (epochs of the problem). Table 3 presents the completion time for each of the above solutions.

The results in Table 3 show that Crucial is efficient in solving the Santa Claus problem, being at most 9% slower than a single-machine solution. In detail, storing the group and gate objects in Crucial induces an overhead of around 4% on the completion time. When cloud threads are used instead of Java ones, a small extra time is further needed—less than a second. This penalty comes from the necessary remote calls to the FaaS platform to start computation.

Takeaways. The fine-grained synchronization capabilities of Crucial permit cloud functions to coordinate efficiently. The synchronization primitives available in the framework fit iterative tasks well and perform better than state-of-the-art solutions at large scale while being simpler to use. This allows Crucial to solve complex concurrency problems efficiently.

6.4 Smile Library

The previous section presented the portage to serverless of a solution to the Santa Claus problem. In what follows, we further push this logic by considering a complex single-machine program. In detail, we report on the portage to serverless of the random forest classification algorithm available in the Smile library. Smile [65] is a multi-threaded library for ML, similar to Weka [48]. It is widely employed to mine datasets with Java and contains around 165 K SLOC. In what follows, we first describe the steps that were taken to conduct the portage using Crucial. Then, we present performance results against the vanilla version of the library.

6.4.1 *Porting* `smile.classification.RandomForest`. The portage consists of adapting the random forest classification algorithm [21] with the help of our framework. In the training phase, this algorithm takes as input a structured file (commonly, `.csv` or `.arff`), which contains the dataset description. It outputs a random forest, i.e., a set of decision trees. During the classification phase, the forest is used to predict the class of the input items. Each decision tree is calculated by a training task (`Callable`). The tasks are run in parallel on a multi-core machine during the training phase. During this computation, the algorithm also extracts the **out-of-bag** (**OOB**) precision, that is the forest's error rate induced by the training dataset.

To perform the portage, we take the following three steps. First, a proxy is added to stream input files from a remote object store (e.g., Amazon S3). This proxy lazily extracts the content of the file, and it is passed to each training task at the time of its creation. Second, the training tasks are instantiated in the FaaS platform. With Crucial, this transformation simply requires calling a `ServerlessExecutorService` object in lieu of the Java `ExecutorService`. Third, the shared-memory matrix that holds the OOB precision is replaced with a DSO object. This step requires to change the initial programming pattern of the library. Indeed, in the original application, the `RandomForest` class creates a matrix using the metadata available in the input file (e.g., the number of features). If this pattern is kept, the application must load the input file to kick off the parallel computation, which is clearly inefficient. In the portage, we instead use a barrier to synchronize the concurrent tasks. The first task to enter the barrier is in charge of creating the matrix in the DSO layer.[6]

For performance reasons, Smile uses Java arrays (mono or multi-dimensional) and not object-oriented constructs (such as `ArrayList`). As pointed out previously in Section 3.3, it is not possible to build proxies for such objects in Java without changing the bytecode generated during compilation. Thus, the portage necessitates to transform these arrays into high-level objects. These objects are then replaced with their Crucial counterparts.

Overall, the portage modifies 378 SLOC in the Smile library (version 1.5.3). This is less than 4% of the original code base to run the random forest algorithm. We also added scripts to deploy and run the serverless solution in AWS Lambda, and performance benchmarks (see below), for a total of around 1K SLOC. Notice that the portage does not preclude local (in-memory) execution, e.g., for testing purpose. This is possible by switching a flag at runtime.

6.4.2 *Evaluation Results.* In Figure 6, we compare the vanilla version of Smile to our Crucial portage. To this end, we use four datasets: (*soil*) is built using features extracted from satellite

[6] This pattern is reminiscent of a Phaser object in Java.

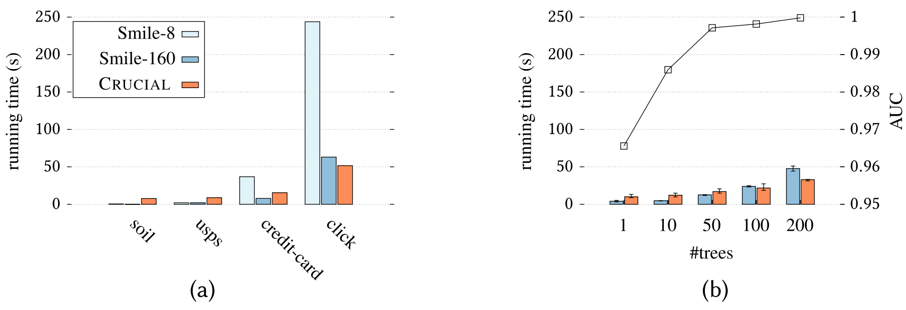

Fig. 6. Smile portage. (a) Performance per dataset using 50 trees. (b) Varying the number of trees for the credit-card dataset [79].

observations to categorize soils [41]; (*usps*) was published by Le Cun et al. [70], and it contains normalized handwritten digits scanned from envelopes by the US Postal Service; (*credit-card*) is a set of both valid and fraudulent credit card transactions [79]; and (*click*) is a 1% balanced subset of the KDD 2012 challenge (Task 2) [51].

We report the performance of each solution during the learning phase. As previously, Crucial is executed atop AWS Lambda. The DSO layer runs with $rf = 2$ in a 3-node (4 vCPU, 16 GB of RAM) Kubernetes cluster. For the vanilla version of Smile, we use two different setups: an hyper-threaded quad-core Intel i7-8550U laptop with 16 GB of memory (tagged Smile-8 in Figure 6), and a quad-Intel CLX 6230 hyperthreaded 80-core server with 740 GB of memory (tagged Smile-160 in Figure 6).[7]

As expected for small datasets (*soil* and *usps*), the cost of invocation out-weights the benefits of running over the serverless infrastructure. For the two large datasets, Figure 6(a) shows that the Crucial portage is up to 5x faster. Interestingly, for the last dataset the performance is 20% faster than with the high-end server.

In Figure 6(b), we scale the number of trees in the random forest, from a single tree to 200. The second y-axis of this figure indicates the **area under the curve** (**AUC**) that captures the algorithm's accuracy. This value is the average obtained after running a 10-fold cross-validation with the training dataset. In Figure 6(b), we observe that the time to compute the random forest triples from around 10–30 seconds. Scaling the number of trees helps improving classification. With 200 trees, the AUC of the computed random forest is 0.9998. This result is in line with prior reported measures [79] and it indicates a strong accuracy of the classifier. Figure 6(b) indicates that training a 200-trees forest takes around 30 seconds with Crucial. This computation is around 50% slower with the 160-threads server. It takes 20 minutes on the laptop test machine (not shown in Figure 6(b)).

Takeaways. Overall, the above results show that the portage is efficient, bringing elasticity and on-demand capabilities to a traditional monolithic multi-threaded library. We focused on the random forest classification algorithm in Smile, but other algorithms in this library can be ported to FaaS with the help of Crucial.

6.5 Usability of Crucial

This section evaluates how Crucial simplifies the writing of stateful serverless applications and their deployment and management in the cloud.

[7] In this last case, the JVM executes with additional flags (+XX:+UseNUMA -XX:+UseG1GC) to leverage the underlying hardware architecture.

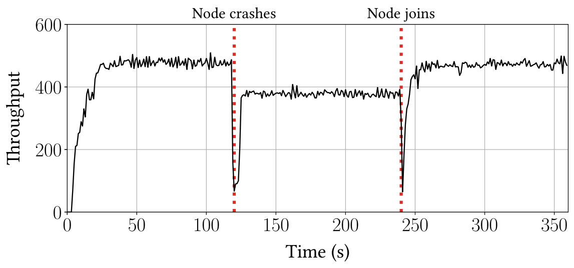

Fig. 7. Inferences per second performed on a k-means model during 6 minutes. Up to 100 concurrent FaaS functions connecting to the shared model on up to 3 DSO nodes with $rf = 2$. Note the FaaS cold start at the beginning.

6.5.1 Data Availability. Our first experiment assesses that Crucial indeed offers high availability to data persisted in the DSO layer. To this end, the membership of DSO is changed during the execution of the serverless k-means. Figure 7 shows a 6-minute run during which inferences are executed with the model trained in Section 6.2.1. The model is stored in a cluster of 3 nodes with $rf = 2$. The inferences are performed using 100 cloud threads. Each inference executes a read of all the objects in the model, i.e., the 200 centroids.

During the experiment, at 120 seconds and 240 seconds, we crash and add, respectively, a storage node to the DSO layer. Figure 7 shows that our system is elastic and resilient to such changes. Indeed, modifications to the membership of the DSO layer affect performance but never block the system. The (abrupt) removal of a node lowers performance by 30%. The initial throughput of the system (490 inferences per second) is restored 20 seconds after a new storage node is added.

Notice that handling catastrophic (or cascading) events is possible by running DSO across several availability zones, or even datacenters. In such cases, SMR can be tailored to accommodate with the increased latency between data replicas [73]. Evaluating these geo-distributed scenarios is however outside of the scope of this article.

6.5.2 Programming Simplicity. Each of the applications used in the evaluation is initially a single-machine program. Table 4 lists the modifications that were necessary to move each program to serverless with Crucial. The differences between the single-machine, parallel code and its serverless counterpart are small. In the case of Smile, as mentioned earlier, they mainly come from the use of low-level non-OOP constructs in the library (e.g., Java arrays). For the other programs, e.g., the logistic regression algorithm detailed in Section 6.2.2, the changes account for less than 3%.

Starting from a conventional OOP program, Crucial requires only a handful of changes to port it to FaaS. We believe that this smooth transitioning can help everyday programmers to start harvesting the benefits of serverless computing.

6.5.3 Cost Comparison. Although one may argue that the programming simplicity of serverless computing justifies its higher cost [55], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

Table 5 offers a cost comparison between Spark and Crucial based on the experiments in Section 6.2.2. The first two columns list the time and cost of the entire experiments, including the time spent in loading and parsing input data, but not the resource provisioning time. The last column lists the cost that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we only consider the pricing for on-demand instances.

Table 4. Lines of Code Changed in Each Application to Move it to FaaS with Crucial

Application	Total lines	Changed lines	
Monte Carlo	44	2	(4.5%)
Mandelbrot	88	3	(3.4%)
Logistic regression	430	10	(2.3%)
k-means	329	8	(2.4%)
Santa Claus problem	255	15	(5.9%)
Random Forest[8]	9, 882	378	(3.8%)

Table 5. Monetary Costs of the Experiments

		Total time (s)	Total cost ($)	Iterations cost ($)
Logistic regression	Spark	192	0.282	0.111
	Crucial	122	0.302	0.154
k-means ($k = 25$)	Spark	168	0.246	0.050
	Crucial	87	0.244	0.057
k-means ($k = 200$)	Spark	330	0.484	0.288
	Crucial	234	0.657	0.492

With the current pricing policy of AWS [12], the cost per second of Crucial is always higher than Spark: 0.25 and 0.28 cents per second for 1, 792 MB and 2, 048 MB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in k-means clustering with $k = 200$, the cost of using Crucial is logically higher. This difference disappears when Crucial is substantially faster than Spark (e.g., $k = 25$).

To give a complete picture of this cost comparison, there are two additional remarks to make here. First, the solution provided with Crucial using 80 concurrent AWS Lambda functions employs a larger aggregated bandwidth from S3 than the solution with Spark. This reduces the cost difference between the two approaches. Second, as pointed in Section 6.1.3, Crucial users only need to pay for the execution time of each function, and not the time the cluster remains active. This includes bootstrapping the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [100].[9]

Takeaways. The programming model of Crucial allows to easily write conventional object-oriented applications for serverless. Starting from a single-machine code, the changes are minimal. In particular, the DSO layer offers the exact same semantics for state sharing and synchronization as a conventional multi-threaded library (e.g., *java.util.concurrent*). Being serverless, applications written with Crucial are scalable. Moreover, they execute at a comparable cost than cluster-based solutions without high upfront investment.

7 RELATED WORK

Serverless computing has gained traction recently and many works have been proposed in this area. In what follows, we survey runtimes (Section 7.1), programming frameworks (Section 7.2), and storage systems (Section 7.3). The bottom of this section (Section 7.4) focuses on (serverless and non-serverless) solutions to the problem of stateful distributed computation. A summary of our findings appears in Table 6, where we compare Crucial to other serverless solutions that address the problems of state sharing and coordination. The comparison is made along five dimensions:

[8]Transitive closure of the dependencies of `smile.classification.RandomForest` in the Smile library.

[9]Provisioning the 11-machine EMR cluster takes 2 minutes (not billed) and bootstrapping requires an extra 4 minutes. A DSO node starts in 30 seconds.

Table 6. Serverless Solutions for State Sharing and Coordination

System	Storage	Mutability	Synchronization	Durability	Consistency
PyWren	object store	—	coarse-grained	replication	weak
ExCamera	rendezvous server	—	rendezvous server	—	—
Ripple	object store	—	high-level dataflow	—	weak
Beldi	key-value store	transactions	—	replication	strong
Pocket/Crail	multi-tiered	— (Pocket) / append-only (Crail)	coarse-grained	ephemeral	—
Cloudburst	FaaS + cache	lattice data structures	coarse-grained	replication	weak: repeatable read and causal
FAASM	hierarchical: local shared memory + global tier (Redis)	byte-array level	coarse-grained	—	variable; strong via explicit locking
CRUCIAL	DSO	fine-grained	fine-grained	replication	linearizability

— *Storage.* This category describes what storage media/service the system uses to keep the intermediate state of an application. The storage type determines the access latency for I/O operations. It ranges from object stores, which exhibit high latency, to in-memory storage designed for fast access and high throughput.
— *Mutability.* This category indicates how the system handles updates to the shared state (e.g., fine-grained updates to arbitrary mutable data, append-only semantics).
— *Synchronization.* Here, we detail how coordination between multiple functions is achieved and, most importantly, at which granularity. For instance, fine-grained coordination allows functions to coordinate between each other with well-known synchronization primitives. On the contrary, coarse-grained coordination, such as in the BSP model, only allows functions to progress in lock step.
— *Durability.* Some systems, such as CRUCIAL, enables the shared state to survive system failures. This dimension categorizes the methods employed to achieve such a property.
— *Consistency.* Since concurrent accesses to the mutable shared state can hit stale data, the system should provide a consistency criterion for the programmer. Here, we list the existing guarantees offered by each system.

7.1 Serverless Runtimes

Serverless computing has appealing characteristics, based on simplicity, high scalability, and fine-grained execution. It has seduced both industry [12, 72, 77] and academia [45]. This enthusiasm has also led to a blossom of open-source systems, e.g., [1–4, 45], to cite a few.

At core, a serverless runtime is in charge of maintaining the user-defined functions, executing them upon request. It must ensure strong isolation between function instances and deliver fast startup times to enhance the critical path of function execution. Many works propose to tackle these two central challenges.

Micro-kernels [68] offer a solid basis to quickly start a function, even achieving sub-millisecond startup time. Catalyzer [32] introduces the `sfork` system call to reuse the state of a running sandbox. Similarly, Firecracker [9] makes containers more lightweight and faster to spawn. SOCK [75] is a serverless-specialized system that uses a provisioning mechanism to cache and clone function containers. SAND [10] exploits function interaction in FaaS to improve startup time and resource efficiency. The system achieves these properties by relaxing isolation at the application level, enabling functions from the same application to share memory and communicate through a hierarchical message bus. Faasm [90] offers similar guarantees using a language-agnostic runtime. Fast function initialization is achieved thanks to a lightweight execution mechanism built atop the **software-fault isolation** (**SFI**) facilities of WebAssembly. For data sharing between functions, Faasm offers a two-tier architecture: the local tier provides in-memory data sharing for co-located functions, while the global tier supports distributed access across the whole system.

Rather than playing out with isolation guarantees for better performance, the goal of Crucial is to provide an efficient substrate for handling *mutable state* and *coordination* at fine granularity over existing platforms (e.g., AWS Lambda), which the above runtimes do not support in place. For instance, Faasm's global state tier is implemented with a distributed Redis instance, which is inefficient for complex operations as discussed in Section 6.1.2. Also, two recent works [44, 56] coincide with our view that existing runtimes do not support mutable shared state and coordination across cloud functions. These two challenges are fully detailed in Section 2. Hellerstein et al. [44] underline that the model is a data-shipping architecture that imposes indirect communication and hinders coordination. Jonas et al. [56] highlight the lack of adequate storage for fine-grained operations and the inability to coordinate functions at fine granularity.

Crucial is evaluated with AWS Lambda, yet it may run atop any FaaS platform. As indicated in Section 5, this is however not a straightforward task since it necessitates to rewrite entirely the scripts for the deployment and invocation of functions for the new targeted platform. This issue is common to many serverless applications. The dependency to the FaaS platform causes a "vendor lock-in" and reduces code portability. Several projects try to address this concern. The Serverless Framework [8] offers plug-ins to simplify the deployment and execution of serverless functions over multiple clouds and FaaS environments. RADON [25] targets the whole development stack with the goal of providing a cloud-agnostic serverless programming experience. Similarly, Lithops [83] hides under a common interface the deployment and execution of serverless functions for different cloud settings.

7.2 Programming Frameworks

Several works that address the challenges of mutable shared state and coordination confront them from a function composition perspective: a scheduler orchestrates the execution of stateless functions and shares information between them.

Several cloud services support function compositions. AWS allows creating state machines with Step Functions [14]. The Amazon States Language (JSON-based) is, however, ill-suited to express complex workflows. IBM Composer [36] offers a similar solution. In this case, function compositions are written in JavaScript and then transformed into state machines. As before, the expressiveness of IBM Composer is bound to a small set of constructs. Google Cloud Composer [78], built on Apache Airflow, allows to create and run a DAG of tasks. In addition to a poorer expressiveness than state machines, it requires to deploy multiple components in Google Kubernetes Engine before the execution of a workflow, similar to an on-premises deployment. Finally, Azure Durable Functions [72] enables to programmatically coordinate function calls. It is the most complete solution among all, allowing to write imperative code. Asynchronous calls to functions are expressed in C# permitting to explicitly wait prior results. Compared with Crucial, the major downside of the above services is their poor performance for running highly parallel compositions [16, 39].

To sidestep this limitation, PyWren [55] pioneered the idea to use FaaS for BSP computations. The article shows the elasticity and scalability of FaaS and demonstrates with a base Python prototype how to run MapReduce workloads. PyWren uses a client-worker architecture where stateless functions share state through slow cloud storage. IBM-PyWren [84] evolves the PyWren model with new features and a broader support to run fully-fledged MapReduce tasks. Furthermore, Locus [80] extends PyWren to support shuffling with a good cost-performance ratio. Tailored to linear algebra, NumPyWren [88] manages a pool of stateless workers that run small tasks built on the fly as mathematical computation progresses. ExCamera [35] is another system atop FaaS, more focused on video encoding and low latency. Its computing framework (*mu*) is designed to run thousands of threads as an abstraction for cloud functions. It handles inter-thread communication through a rendezvous server. *gg* [34] keeps *mu*'s line for running serverless parallel threads but targeting a broader audience. Finally, Ripple [57] is a programming framework to enable single-machine applications to benefit from the ample parallelism of FaaS platforms. It provides a simple interface of eight functions for programmers to express the dataflow of their applications. Also, it automates resource provisioning and handles fault tolerance by eagerly detecting stragglers. Before the full computation run, the framework performs a series of dry runs to test and find the best resource provisioning for the job.

Jangda et al. [54] propose an operational model for serverless platforms (named λ_λ), a simplified semantics, and an extension for stateful functions. The simplified semantics is equivalent to λ_λ when the cold and warm states of a function produce the same result. The stateful extension adds a (global) transactional key-value store that serverless functions may call. Extending serverless computing with transactions is also the path taken in Beldi [105]. Compared with [54], the model of Beldi does not serialize accesses to the data store through a central lock.

Fault tolerance is a key concern when programming in a serverless environment. The λ_λ model captures the fact that the FaaS platform may start multiple instances to answer a request, yet use a single one to reply. The bisimulation result in [54] indicates when this is equivalent to executing the request exactly once (that is under the simplified semantics). Serverless cloud vendors warn programmers that serverless functions must be idempotent. Yet they do not precise what does this mean, neither what to do when computation is stateful. In [93], the authors introduce a layer that interposes between the FaaS platform and the storage engine to ensure read atomicity when functions access multiple data items. Rifle [64] is a proposal to achieve idempotence in SMR. It could be applied to the DSO layer of Crucial to further simplify programming.

In comparison to Crucial, none of the above frameworks provides a complete solution to the joint problem of fine-grained updates and synchronization (see Table 6). To wit, state sharing in PyWren, and the likes [80, 84], is too coarse-grained and with weaker consistency guarantees than Crucial. Similarly, Ripple shares intermediate results using Amazon S3, which is slow and provides an ill-suited interface for tasks with fine-grained data sharing needs. ExCamera requires a long-lived relay server to share state between workers. For certain operations such as `AllReduce`, the message-passing architecture can become a bottleneck, whereas in Crucial, communication complexity can be kept low through remote procedure call. Analogous concerns can be raised about synchronization in the surveyed systems.

7.3 Storage

Many frameworks focus on cloud function scheduling and coordination, while using disaggregated storage to manage data dependencies. In particular, they opt to write shared data to slow, highly-scalable storage [55, 84, 88]. To hide latency, they perform coarse-grained accesses, resort to in-memory stores, or use a combination of storage tiers [80].

Pocket [62] is a distributed data store that scales out and in on demand to match the storage needs of serverless applications. It leverages multiple storage tiers and right-sizes them offline based on the application requirements. Crail [95] presents the NodeKernel architecture with similar objectives. These two systems are designed for ephemeral data, which are easy to distribute across a cluster. They do not use a distributed hash table that would require data movement when the cluster topology changes, but instead use a central directory. Both systems scale in to zero when computation ends.

InfiniCache [99] is an in-memory cache built atop cloud functions. The system exploits FaaS to store objects in a fleet of ephemeral cloud functions. It uses erasure coding and a background rejuvenation mechanism to maintain data availability despite the continuous fluctuations in the pool of cloud functions as they are reclaimed by the provider. Similar to a traditional distributed in-memory cache, InfiniCache has been designed to provide fast access to read-only objects but not to mutate them as in Crucial.

The above works do not allow fine-grained updates to mutable shared state. Such a feature can be abstracted in various ways. Crucial chooses to represent state as objects and keeps the well-understood semantics of linearizability. This approach is way more in line with the inherent simplicity of serverless computing.

Existing storage systems such as Memcached [33], Redis [81], or Infinispan [69] cannot be readily used as a shared object layer. They either provide too low-level abstractions or require server-side scripting. Coordination kernels such as ZooKeeper [49] can help synchronizing cloud functions. However, their expressiveness is limited, and they do not support partial replication [31, 58]. We show these problems in Section 6.

Crucial borrows the concept of callable objects from Creson [96]. It simplifies its usage (`@Shared` annotation), provides control over data persistence, and offers a broad suite of synchronization primitives. While Crucial implements strong consistency, some systems [89, 92, 97] rely instead on weak consistency, trading ease of programming for performance. Weak consistency has been used to implement distributed stateful computation in FaaS, as detailed in the next section.

7.4 Distributed Stateful Computation

Cloudburst [94] is a stateful serverless computation service. State sharing across cloud functions is built atop Anna [103], an autoscaling key-value store that supports a lattice put/get CRDT data type. Cloudburst offers repeatable read and consistent snapshot consistency guarantees for function composition—something that is not achievable, for instance, when using AWS Lambda in conjunction with S3 (i.e., computing $x + f(x)$ is not possible if x mutates). Contrary to Crucial, Cloudburst necessitates a custom FaaS platform (which was written from scratch).

Cirrus [23] is a ML framework that leverages cloud functions to efficiently use computing resources. It specializes in iterative training tasks and asynchronous stochastic gradient descent. The initial motivation for Cirrus is much in line with Crucial. However, the solution is quite different. Cirrus relies on a distributed data store that does not allow custom shared objects and/or computations. Furthermore, distributed workers cannot coordinate as they do in Crucial.

Besides serverless systems, there exist many frameworks for machine clusters that target stateful distributed computation.

Ray [74] is a recent specialized distributed system mainly targeting AI applications (e.g., Reinforcement Learning). It offers a unified interface for both stateless tasks and stateful actor-based computations. Crucial shares Ray's motivation for the need of a specialized system that combines stateful and stateless computations. However, Ray couples both models in the same system and is built for a provisioned resource environment where stateless tasks and actors live co-located. Crucial is built with serverless in mind, so it separates the two types of computation. That is, Crucial uses the highly scalable capabilities of FaaS platforms for stateless tasks and a layer of shared objects for data sharing and coordination. The programming model is also consequently different: while Ray exposes interfaces to code tasks and actors, Crucial uses a traditional shared-memory model where concurrent tasks are expressed as threads.

Other systems with a focus on stateful computations, such as Dask and PyTorch, usually build on low-level technologies (e.g., MPI) to communicate among nodes. These frameworks rely on clusters with known topology and fail to quickly scale out or in to match demand. Such a design is also at odds with the FaaS model, where functions are forbidden to communicate directly. Specialized distributed big data processing frameworks, such as MapReduce, are available as a service in the cloud (e.g., AWS-EMR). We have explored such alternatives in the evaluation section (Section 6.2.2), where we have compared Crucial against Apache Spark.

8 CLOSING REMARKS

Serverless computing is a recent paradigm to program the cloud. In this paradigm, the quantum of computation is a cloud function. Once deployed in an FaaS platform, the function is executed on-demand, at scale and in a pay-per-use manner. This article details Crucial, a new programming framework for FaaS platforms. In line with recent works [55, 94, 105], Crucial pivots serverless computing on its head. Instead of event-driven stateless computations, FaaS is used to run complex stateful programs. In building and evaluating Crucial, we faced several challenges and opportunities. This section summarizes these observations before closing.

8.1 Lessons Learned

During the development and evaluation of Crucial, we learned the following lessons that we find useful to the community: *(Lesson 1.)* Our evaluation of Crucial was conducted entirely on AWS Lambda, a state-of-the-art platform. There are some inherent difficulties in using public cloud services. As it acts as a black box, a public cloud service makes certain experimental results complex to understand and reproduce [85]. Anomalies can be due to cold starts, function co-location, and intermittent service disruptions. Therefore, during the evaluation of Crucial, we took extra care when examining the results. Fortunately, we did not experience major difficulties besides cold starts, which we could easily overcome. *(Lesson 2.)* Evaluating the cost of a serverless application requires to understand at fine grain the billing procedure of the various cloud services involved. A typical deployment requires the use of many services—e.g., FaaS, object storage, virtual cloud, public IPs, and so on. For our cost comparison in Section 6.5.3, we had no other choice than to manually extract these costs and aggregate them. This can be an intricate and time-consuming task. *(Lesson 3.)* As indicated in Section 5, Crucial can be used atop any FaaS platform, provided a Java runtime is available. Serverless computing platforms currently have incompatible APIs when uploading/calling cloud functions. Thus, the choice of a platform must consider the necessary tools (e.g., scripts) to simplify the deployment and execution of a serverless application. As a result, the tools created to use Crucial atop AWS Lambda would require some adjustments for other platforms. We believe that this situation will improve over time as cloud vendors start homogenizing their services.

8.2 Limitations and Future Work

FaaS platforms ship data to the cloud functions. Crucial partly remedies to this problem by providing a storage layer that may execute complex operations near data. Therefore, the system runs over two different layers: the FaaS platform itself and DSO (see Figure 1). An interesting direction of improvement would be to run jointly these two layers in the same infrastructure, e.g., the same container orchestrator, to improve performance.

In Crucial, cloud threads cannot directly communicate with each other. For instance, signaling between threads is not possible (see Section 3.3). This limitation of our design is intended to allow running Crucial atop any FaaS platform. However, for efficiency reasons, it could be of interest to benefit from direct function-to-function communication, without resorting to an external storage layer.

A cloud thread is a computing abstraction between a light- and a heavy-weight thread. Namely, contrary to the light-weight model, sharing is made explicit in a Crucial program. However, sharing among cloud threads is not restricted to IPC abstractions, as with heavy-weight threads (i.e., processes). We believe that this is an interesting trade-off to further explore. On one hand, implicit sharing simplifies the life of the programmer. On the other, explicitly marking shared data could be of interest for performance, e.g., by scheduling computation near data. In addition, we note that closing the gap between cloud threads and conventional threads (either light or heavy) would simplify the portage to serverless of single-machine programs.

8.3 Conclusion

This article presents Crucial, a system to program highly concurrent stateful serverless applications. Crucial can be used to construct demanding serverless programs that require fine-grained support for mutable shared state and synchronization. We show how to use it to implement applications such as traditional data parallel computations, iterative algorithms, and coordination tasks.

Crucial is built using an efficient disaggregated in-memory data store and an FaaS platform. Contrary to prior works, such as Faasm [90] and Cloudburst [94], Crucial can run on any standard FaaS platform, simply requiring the existence of a Java runtime.

Our evaluation shows that, for two common ML algorithms, Crucial achieves superior or comparable performance to Apache Spark. Crucial is also a good fit for function synchronization, outperforming the ZooKeeper coordination kernel in this task. In particular, it can solve efficiently complex coordination problems despite the inherent costs of its disaggregated design. For data sharing across cloud functions, Crucial compares favorably against storage alternatives such as Redis.

Our framework allows to move traditional single-machine, multi-threaded Java programs to serverless. We use it to port Smile, a state-of-the-art multi-threaded ML library. The portage achieves performance comparable to the one of a dedicated high-end server, while providing elasticity and on-demand capabilities to the application.

Crucial offers conventional multi-threaded abstractions to the programmer. In our evaluation, less than 6% of the application code bases differ from standard solutions using plain old Java objects. We believe that this simplicity can help to broaden the horizon of serverless computing to unexplored domains.

APPENDIX

A PARALLEL VIRTUAL SYNCHRONY

In Section 4, we mention that Crucial relies internally on a variation of virtual synchrony [27]. Below, we define formally such a variation and explain how it is constructed with atomic multicast and an unreliable failure detector [26]. Furthermore, we explain how virtual synchrony is used to implement the DSO layer in Crucial.

A.1 Specification

Virtual synchrony allows processes to deliver messages across a sequence of system views. Parallel virtual synchrony extends this idea to the case where messages are not addressed to all the processes but to a subset of them.

In detail, let $\mathcal{P}$ be a set of processes and $\mathcal{M}$ some set of (applicative) messages. **Parallel virtual synchrony (PVS)** provides an interface that consists of three operations: $send(m)$ sends message m to its destination set ($dst(m) \subseteq \mathcal{P}$); $rcv(m)$ triggers at some process in $dst(m)$ when m is received; and $viewChange(V)$ notifies the local process that a view change occurs ($V \subseteq \mathcal{P}$). When the event $viewChange(V)$ happens at process p, we shall say that p *installs* view V. Initially, the view containing all the processes is installed.

Consider two messages m and m' and a process $p \in dst(m) \cap dst(m')$. Relation $m \overset{p}{\mapsto} m'$ captures the *local delivery order* at process p. This relation holds when at the time p delivers m, p has not delivered m'. The *delivery order* is then defined as $\mapsto = (\bigcup_{p \in \mathcal{P}} \overset{p}{\mapsto})^*$, where R^* denotes the transitive closure of relation R. Similarly, we can define a relation $<$ on the order in which the views are installed. PVS guarantees the following set of properties during a run:

(Precise Membership) the installation of views ensures that *(i)* every faulty process gets eventually excluded from the view, and *(ii)* every correct process is eventually present in the view.
(Primary Component) the installation of views is sequential ($<$ is a total order);
(Virtual Synchrony) processes agree on the messages they receive in a view (if p delivers m in view V, then every correct process $q \in V \cap dst(m)$ delivers m in V).
(Partial Order) processes agree on the order in which they receive the messages ($\mapsto$ is a partial order)
(Disjoint-access Parallelism) messages sent to different destinations are processed in parallel (during a nice run, if a process p takes a computation step, then some message m is sent to p.[10])

A.2 Implementation

PVS can be implemented with the help of genuine atomic multicast [42] and an eventually perfect failure detector [26]. In detail, messages are sent and received through the atomic multicast layer. To build a new view, we first construct it using the failure detector. Then, to install it, e.g., when a process is suspected, the view is multicast to all the processes. This construction is detailed in Algorithm 1. The theorem below states that it is indeed a sound implementation of the PVS abstraction.

Theorem A.1. *Algorithm 1 implements parallel virtual synchrony.*

Proof. For starters, let us recall the definitions of the two building blocks of Algorithm 1. Atomic multicast ensures the following set of properties: (Integrity) for every process p and message m, p delivers m at most once, and only if p belongs to $dst(m)$ and m was previously multicast; (Termination) if a correct process p multicasts a message m or delivers it, eventually every correct process in $dst(m)$ delivers m; and (Ordering) the transitive closure of the delivery relation forms a strict partial order. A protocol is genuine when it ensures the minimality property [42], that is, in every run of the protocol, if some process p takes a step, there exists a message m addressed to p. An eventually perfect detector returns at each process a set of suspects such that (Strong Completeness) every faulty process is eventually suspected; and (Eventually Strong Accuracy) every correct process is eventually not suspected.

We now prove that each property of PVS is satisfied with Algorithm 1. (Precise Membership) Consider that a process p is correct. In that case, since $\Diamond P$ is eventually perfect, p is eventually not in $\Diamond P$ at all the processes. Consider that this is true after time t_0. At any time $t > t_0$, if some

[10]A nice run [50] is a failure-free run during which the failure detector behaves perfectly.

ALGORITHM 1: Parallel virtual synchrony – code at process p

```
1:  Variables:
2:      $\Diamond P$                                   // An eventually perfect failure detector.
3:      $AM$                                           // An atomic muticast protocol.
4:      $view \leftarrow \mathcal{P}$                  // Local variable to hold the view.
5:
6:  when $view \neq (\mathcal{P} \setminus \Diamond P)$
7:      $v \leftarrow (\mathcal{P} \setminus \Diamond P)$
8:      $AM$.multicast($v$, VIEW) to $\mathcal{P}$
9:
10: when $AM$.deliver($m, f$)
11:     if $f$ = VIEW then
12:         $view \leftarrow v$
13:     else
14:         trigger $rcv(m)$
15:
16: when $send(m)$
17:     $AM$.multicast($m$, APP) to $dst(m)$
```

process p executes line 8, then it send a message $(v, \texttt{VIEW})$, with $p \in v$. In addition, if at some correct process q, p is not in variable *view* at t_0, q eventually executes line 8. Thus, eventually p is added to *view* at p. A similar reasoning holds for the case where p is faulty. (Primary Component) This is a straightforward consequence of the Ordering property of atomic multicast. (Virtual Synchrony) Consider that a message m is received by a process p in view V. Assume by contradiction that some correct process $q \in dst(m) \cap V$ does not receive m in V. As q is correct, it eventually joins some view $V' > V$. Let $V \leq V_1 \leq V'$ be the next view in the order $<$ that q joins. Then, we have $(m, \texttt{APP}) \overset{p}{\mapsto} (V_1, \texttt{VIEW})$ and $(V_1, \texttt{VIEW}) \overset{q}{\mapsto} (m, \texttt{APP})$. Contradiction. (Partial Order) This property is a consequence of the Ordering property of atomic multicast. (Disjoint-access Parallelism) This property follows from the minimality of genuine atomic multicast and the fact that during a nice run, no view change occurs. □

A.3 Usage

Crucial uses the PVS abstraction to implement state machine replication [86]. We outline the principles of such a construction in what follows.

For some view V and some object key k, we assume a locally computable function $rep(k, V)$. This function defines a replication mapping, that is the processes in charge of replicating object k in view V. There are at least $f + 1$ replicas per object in a given view, where f is an upper bound on the number of failures that may occur during a run. Function $rep(k, V)$ is also used to elect a primary node (e.g., using the highest process id). The primary is in charge of answering to the application calls and migrating object k across views.

With more details, when the primary receives an application call, it sends the call to the object replicas through the PVS service. Upon receiving a call, all the processes apply it to their local copy and the primary answers back to the client. When a new view V is installed at some process p, p transitions each key k with $p \in rep(k, V)$ to the `migration` state. It also stops applying commands to k. The current state of object k is multicast by the primary replica of the old view using the PVS service. (If this process is slow or crashed, any replica may take over this responsibility.) Process p

transitions to the `running` state for key k, thus resuming execution, when it delivers this message in view V.

The above construction guarantees that replicated objects are linearizable. However, progress only holds when the churn is sufficiently low, requiring a view change to take place only once the keys have all migrated. To improve upon this situation, it is possible to condition the installation of a new view to the fact that the previous one completed successfully.

REFERENCES

[1] 2016. Apache OpenWhisk is a serverless, open source cloud platform. Retrieved September 2021 from https://openwhisk.apache.org/.

[2] 2016. Kubeless. Retrieved September 2021 from https://kubeless.io/.

[3] 2016. OpenFaaS. Retrieved September 2021 from https://www.openfaas.com/.

[4] 2016. Serverless Functions for Kubernetes - Fission. Retrieved September 2021 from https://fission.io/.

[5] 2019. lambda-maven-plugin. Retrieved September 2021 from https://github.com/SeanRoy/lambda-maven-plugin.

[6] 2020. The Crucial Project - GitHub. Retrieved September 2021 from https://github.com/crucial-project.

[7] 2021. Lithops - GitHub. Retrieved September 2021 from https://github.com/lithops-cloud/lithops.

[8] 2021. Serverless Framework. Retrieved September 2021 from https://www.serverless.com/.

[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Santa Clara, CA, 419–434. Retrieved September 2021 from https://www.usenix.org/conference/nsdi20/presentation/agache.

[10] 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 *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, Berkeley, CA, 923–935. Retrieved September 2021 from http://dl.acm.org/citation.cfm?id=3277355.3277444.

[11] Amazon. 2008. AWS Simple Storage Service. Retrieved September 2021 from https://aws.amazon.com/s3.

[12] Amazon. 2014. AWS Lambda. Retrieved September 2021 from https://docs.aws.amazon.com/lambda.

[13] Amazon. 2015. Invoke - AWS Lambda. Retrieved September 2021 from https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html.

[14] Amazon. 2016. AWS Step Functions. Retrieved September 2021 from https://aws.amazon.com/step-functions.

[15] Amazon. 2017. AWS Glue. Retrieved September 2021 from https://aws.amazon.com/glue/.

[16] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*. Association for Computing Machinery, New York, NY, 25–30. DOI: https://doi.org/10.1145/3366623.3368137

[17] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*. ACM, New York, NY, 41–54. DOI: https://doi.org/10.1145/3361525.3361535

[18] Mordechai Ben-Ari. 2001. How to solve the santa claus problem. *Concurrency: Practice and Experience* 10 (2001). DOI: https://doi.org/10.1002/(SICI)1096-9128(199805)10:63.0.CO;2-2

[19] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Transactions on Computers Systems* 5, 1 (Jan. 1987), 47–76. DOI: https://doi.org/10.1145/7351.7478

[20] Stephen Blum. 2014. Amazon SNS vs PubNub: Differences for Pub/Sub. Retrieved September 2021 from https://www.pubnub.com/blog/2014-08-21-amazon-sns-pubnub-differences-pubsub/.

[21] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (Oct. 2001), 5–32. DOI: https://doi.org/10.1023/A:1010933404324

[22] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and extensible component systems*.

[23] 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*. Association for Computing Machinery, New York, NY, 13–24. DOI: https://doi.org/10.1145/3357223.3362711

[24] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew M. Zhang, and Randy Katz. 2018. A case for serverless machine learning. In *Proceedings of the Workshop on Systems for ML and Open Source Software at NeurIPS*.

[25] Giuliano Casale, Matej Artac, Willem-Jan van den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, Satish Narayana Srirama, Damian A. Tamburri, Michael Wurster, and Lulai

Zhu. 2020. RADON: Rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2020), 77–87. DOI: https://doi.org/10.1007/s00450-019-00413-w

[26] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43, 2 (1996), 225–267.

[27] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. 2001. Group communication specifications: A comprehensive study. *ACM Computing Surveys* 33, 4 (2001), 427–469.

[28] Databricks. 2014. spark-perf. Retrieved September 2021 from https://github.com/databricks/spark-perf.

[29] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (Jan. 2008), 107–113. DOI: https://doi.org/10.1145/1327452.1327492

[30] David J. DeWitt and Michael Stonebraker. 2008. MapReduce: A major step backwards. DatabaseColumn Blog. Retrieved September 2021 from http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

[31] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. 2015. Extensible distributed coordination. In *Proceedings of the 10th European Conference on Computer Systems.* ACM, New York, NY, 16 pages. DOI: https://doi.org/10.1145/2741948.2741954

[32] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems.* Association for Computing Machinery, New York, NY, 467–481. DOI: https://doi.org/10.1145/3373376.3378512

[33] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (Aug. 2004), 5–.

[34] 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 *Proceedings of the 2019 USENIX Annual Technical Conference.* USENIX Association, Renton, WA, 475–488. Retrieved September 2021 from https://www.usenix.org/conference/atc19/presentation/fouladi.

[35] 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 *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation.*

[36] The Apache Software Foundation. 2017. OpenWhisk Composer. Retrieved September 2021 from https://github.com/apache/openwhisk-composer.

[37] The Apache Software Foundation. 2019. ZooKeeper barrier recipe. Retrieved September 2021 from https://zookeeper.apache.org/doc/current/recipes.html#sc_recipes_eventHandles.

[38] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless End Game: Disaggregation enabling Transparency. arXiv:2006.01251. Retrieved September 2021 from https://arxiv.org/abs/2006.01251.

[39] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto. 2018. Comparison of FaaS orchestration systems. In *IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companio'18n).* 148–153. DOI: 10.1109/UCC-Companion.2018.00049

[40] Simson L. Garfinkel. 2007. *An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS.* Technical Report TR-08-07. Harvard Computer Science Group. Retrieved September 2021 from http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829568.

[41] Markus Goldstein and Seiichi Uchida. 2016. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLOS ONE* 11, 4 (04 2016), 1–31. DOI: https://doi.org/10.1371/journal.pone.0152173

[42] Rachid Guerraoui and André Schiper. 1997. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms.* Saarbrücken, Germany.

[43] Red Hat. 2015. Reliable group communication with JGroups. Retrieved September 2021 from http://jgroups.org/manual/#TOA.

[44] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless computing: One step forward, two steps back. In *Proceedings of the CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research.* Retrieved September 2021 from http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[45] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing.* USENIX Association, Berkeley, CA, 33–39. Retrieved September 2021 from http://dl.acm.org/citation.cfm?id=3027041.3027047.

[46] Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two algorithms for barrier synchronization. *International Journal of Parallel Programming* 17, 1 (Feb. 1988), 1–17. DOI: https://doi.org/10.1007/BF01379320

[47] C. A. R. Hoare. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (Oct. 1974), 549–557. Retrieved September 2021 from https://doi.org/10.1145/355620.361161

[48] G. Holmes, A. Donkin, and I. H. Witten. 1994. WEKA: A machine learning workbench. In *Proceedings of ANZIIS'94 - Australian New Zealnd Intelligent Information Systems Conference*. 357–361.

[49] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1 pages.

[50] Michel Hurfin and Michel Raynal. 1999. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing* 12, 4 (01 Sep 1999), 209–223. DOI: https://doi.org/10.1007/s004460050067

[51] Tencent Inc. 2014. KDD Cup - 2012. Retrieved September 2021 from https://www.openml.org/d/1220.

[52] V. Ishakian, V. Muthusamy, and A. Slominski. 2018. Serving deep learning models in a serverless platform. In *IEEE International Conference on Cloud Engineering (IC2E'18)*. 257–262. DOI: 10.1109/IC2E.2018.00052

[53] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*. 151–160. DOI: https://doi.org/10.1145/197917.198079

[54] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, (Oct. 2019), 26 pages. DOI: https://doi.org/10.1145/3360575

[55] 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*. DOI: https://doi.org/10.1145/3127479.3128601

[56] 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. Retrieved September 2021 from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[57] Shannon Joyner, Michael MacCoss, Christina Delimitrou, and Hakim Weatherspoon. 2020. Ripple: A Practical Declarative Programming Framework for Serverless Compute. arXiv:2001.00222. Retrieved September 2021 from https://arxiv.org/abs/2001.00222.

[58] Babak Kalantari and André Schiper. 2013. In *Proceedings of the 14th International Conference Distributed Computing and Networking*. Springer Berlin Heidelberg, Chapter Addressing the ZooKeeper Synchronization Inefficiency.

[59] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* .

[60] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*.

[61] Y. Kim and J. Lin. 2018. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 451–455. DOI: 10.1109/CLOUD.2018.00063

[62] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Carlsbad, CA, 427–444. Retrieved September 2021 from https://www.usenix.org/conference/osdi18/presentation/klimovic.

[63] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (April 2010).

[64] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 71–86. DOI: https://doi.org/10.1145/2815400.2815416

[65] Haifeng Li. 2014. Smile. Retrieved from https://haifengl.github.io.

[66] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (March 1982), 129–137. DOI: https://doi.org/10.1109/TIT.1982.1056489

[67] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[68] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 218–233. DOI: https://doi.org/10.1145/3132747.3132763

[69] Francesco Marchioni and Manik Surtani. 2012. *Infinispan Data Grid Platform*. Packt Publishing Ltd.

[70] Ofer Matan, Henry S. Baird, Jane Bromley, Christopher J. C. Burges, John S. Denker, Lawrence D. Jackel, Yann Le Cun, Edwin P. D. Pednault, William D. Satterfield, Charles E. Stenard, and Timothy J. Thompson. 1992. Reading handwritten digits: A zip code recognition system. *Computer* 25, 7 (July 1992), 59–63. DOI: https://doi.org/10.1109/2.144441

[71] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine learning in apache spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. Retrieved September 2021 from http://jmlr.org/papers/v17/15-237.html.

[72] Microsoft. 2016. Azure Durable Functions. Retrieved September 2021 from https://functions.azure.com.

[73] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 358–372.

[74] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 561–577. Retrieved September 2021 from http://dl.acm.org/citation.cfm?id=3291168.3291210.

[75] 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 *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, 57–70. Retrieved September 2021 from https://www.usenix.org/conference/atc18/presentation/oakes.

[76] Google Cloud Platform. 2010. BigQuery. Retrieved September 2021 from https://cloud.google.com/bigquery/.

[77] Google Cloud Platform. 2016. Cloud Functions. Retrieved September 2021 from https://cloud.google.com/functions/.

[78] Google Cloud Platform. 2018. Cloud Composer. Retrieved September 2021 from https://cloud.google.com/composer.

[79] A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. 2015. Calibrating probability with undersampling for unbalanced classification. In *Proceedings of the 2015 IEEE Symposium Series on Computational Intelligence*. 159–166.

[80] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Boston, MA, 193–206. Retrieved September 2021 from https://www.usenix.org/conference/nsdi19/presentation/pu.

[81] Redis. 2009. Retrieved September 2021 from https://redis.io/.

[82] Redis. 2019. Replication. Retrieved September 2021 from https://redis.io/topics/replication.

[83] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. 2021. Toward multicloud access transparency in serverless computing. *IEEE Software* 38, 01 (jan 2021), 68–74. DOI: https://doi.org/10.1109/MS.2020.3029994

[84] 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*. ACM, New York, NY, 1–8. DOI: https://doi.org/10.1145/3284028.3284029

[85] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1–2 (Sept. 2010), 460–471. DOI: https://doi.org/10.14778/1920841.1920902

[86] Fred B. Schneider. 1986. *The State Machine Approach: A Tutorial.* Technical Report 86-800. Revised June 1987.

[87] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22, 4 (1990), 299–319. DOI: https://doi.org/10.1145/98163.98167

[88] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*. Association for Computing Machinery, New York, NY, USA, 281–295. DOI: https://doi.org/10.1145/3419111.3421287

[89] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Convergent and commutative replicated data types. In *Proceedings of the Bulletin of the European Association for Theoretical Computer Science* (June 2011).

[90] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker.

[91] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI-The Complete Reference, Volume 1: The MPI Core* (2nd. (revised) ed.). MIT Press, Cambridge, MA.

[92] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP '11: Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. New York, NY, 385–400. DOI: https://doi.org/10.1145/2043556.2043592

[93] 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 15th European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, 15 pages. DOI: https://doi.org/10.1145/3342195.3387535

[94] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.* 13, 12 (August 2020), 2438–2452. DOI: https://doi.org/10.14778/3407790.3407836

[95] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of temporary storage in the nodekernel architecture. In *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, Renton, WA, 767–782. Retrieved September 2021 from https://www.usenix.org/conference/atc19/presentation/stuedi.

[96] Pierre Sutra, Etienne Riviere, Cristian Cotes, Marc Sánchez-Artigas, Pedro García-López, Emmanuel Bernard, William Burns, and Galder Zamarreno. 2017. CRESON: Callable and replicated shared objects over NoSQL. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. DOI: https://doi.org/10.1109/ICDCS.2017.239

[97] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. ACM SIGOPS, ACM Press, Copper Mountain, CO, 172–182. Retrieved September 2021 from http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[98] John A. Trono. 1994. A new exercise in concurrency. *SIGCSE Bulletin* 26, 3 (1994), 8–10. DOI: https://doi.org/10.1145/187387.187391

[99] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 267–281. Retrieved September 2021 from https://www.usenix.org/conference/fast20/presentation/wang-ao.

[100] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *Proceedings of the IEEE Conference on Computer Communications, INFOCOM 2019*.

[101] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, 133–146. Retrieved September 2021 from https://www.usenix.org/conference/atc18/presentation/wang-liang.

[102] Chenggang Wu. 2019. The State of Serverless Computing. Retrieved from https://www.infoq.com/presentations/state-serverless-computing/. Presentation at QCon New York 2019.

[103] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling tiered cloud storage in anna. *Proceedings of the VLDB Endowment* 12, 6 (Feb. 2019), 624–638. DOI: https://doi.org/10.14778/3311880.3311881

[104] 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 *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, San Jose, CA, 15–28. Retrieved September 2021 from https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[105] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 1187–1204. Retrieved September 2021 from https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

Received July 2020; revised September 2021; accepted October 2021