National College of Ireland

An efficient Serverless Architecture to support fine-grained coordination and state sharing for Stateful Applications

MSc Research Project
Cloud Computing

Ankit Kumar Singh
Student ID: x19205121

School of Computing
National College of Ireland

Supervisor: Sean Heeney

An efficient Serverless Architecture to support fine-grained coordination and state sharing for Stateful Applications

Ankit Kumar Singh
19205121
MSc in Cloud Computing

20th September 2021

Abstract

In today's cloud driven work culture, serverless infrastructure is widely adopted due to its pay-as-you-go benefits. In serverless resource allocation is fully managed by the cloud provider and the developer can solely concentrate on application development. To take to benefit of serverless infrastructure many organizations are migrating their work on serverless. A range of applications and domains including image, text, and speech processing have led to machine learning (ML) becoming a widely deployed technology across many IT industries. It is difficult to implement ML algorithms workflows on a serverless platform. There are clearly defined user interactions during various steps in ML workflows, such as data preprocessing, data training, and data fine-tuning. The user may execute this frequently and will require low latency and the provision of resources automatically. A serverless platform must be designed efficiently to support a stateful application workflow. There is a need for distributed shared memory layer to manage synchronization and fine-grained coordination between intermediate shared data. The proposed architecture provides an efficient workflow for stateful application with low network latency and the ability to share the mutable object with fine grained coordination and synchronization. I have deployed the architecture on Amazon Web Services lambda functions. I have also validated the architecture using micro-benchmarks like latency, throughput, and parallelism. Further to state the fine-grained state management in ML application on serverless I have compared its performance with AWS Spark Clusters for k-means clustering algorithm using 30 GB of data generated using spark-pref. According to the results, it was found that the proposed architecture optimizes ML application's performance on serverless in terms of time.

1 Introduction

Cloud computing has emerged as a key building component for expanding businesses in the digital era. The benefits of cloud technology, such as scalability and elasticity, have prompted widespread adoption. Traditional infrastructure is much more expensive, difficult to administer, and unavailable without an initial investment. Whereas, Cloud services are significantly less expensive, easier to operate, and available without an initial investment. As businesses move towards containerized and microservices based applications as a result serverless computing is gaining traction as a new platform for deploying applications. TO shift the focus from servers to application development, serverless computing comes in to picture. Serverless has supplanted the monolithic method to application deployment. As indicated by Google Trends' frequent searches for "serverless", the popularity of serverless computing is expanding. The market is anticipated to reach $7.72 billion by 2021 Castro et al. (2019). Serverless Computing is already available on the major cloud provider platforms, including Amazon, Microsoft, IBM, Google, and others.

1.1 Motivation and Background

Serverless Computing is offering high level abstractions from servers and developers or programmers can mainly focus on application development. For example, serverless provides the cloud functions to program their application using the stateless programming languages like Python or JavaScript and also specify how these function should run in response to web requests or triggered events. These cloud functions get invoked when an event is triggered such as an HTTP request to upload a file to an object store, at that time machine spins up a container with specified resources and executes the desired operation written in the function inside that container. Devlopers can also use serverless object storage, key-value store database, message queues, and many more as a backend as a service to develop their application. The qualities of serverless which has boosted its usage in a cloud environment are, high level abstraction to hide backend complexity and servers from the client, only charge for the utilized resources (no charge for an idle machine) that is "pay as you go" service and serverless has also capability to scale up and down and automate the resources allocation to match the resource demand Schleier-Smith et al. (2021).

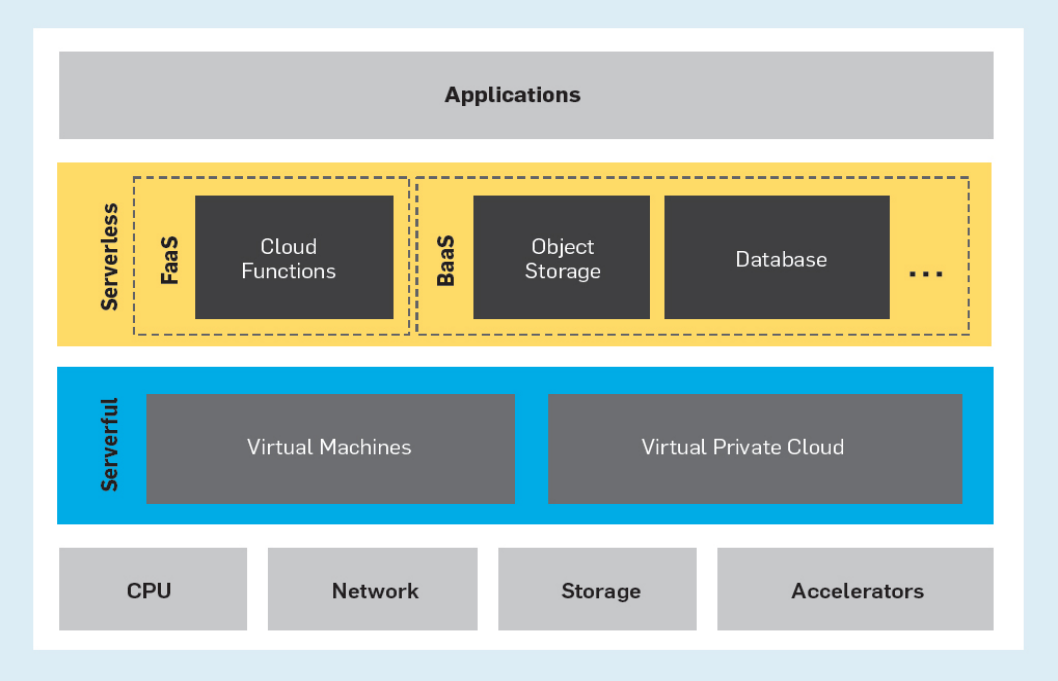

Figure 1: Serverless Architecture, Schleier-Smith et al. (2021)

With the growing popularity of serverless, there are often research and development are going on in this field. Serverless is ideally suited for small event driven applications which can scale separately and do not need to share their intermediate state. The model has limitations such as memory, size,

and runtime. The serverless function has a limit of maximum size and memory of a function which is 3GB and 5 minutes of maximum runtime for each function. Although now maximum runtime is upto 15 minutes and a maximum memory limit of 10,240 MB AWS (2018). The researcher is trying to pull out these limitations of serverless to provide support to run stateful applications on serverless. As mentioned in Ghosh et al. (2020), the time to access the database is almost 14 times more in serverless architecture than in a virtual machine based cloud environment setup. During the development of an application such as machine learning or natural language processing, the code must be updated constantly, and one function must wait for another until it can read its state from a slow object store. This creates overheads and requires additional computing power to run. Furthermore, as a result of cold start delays, application's performance cannot be predicted or should not perform as expected Xu et al. (2019).

It is crucial to implement ML algorithms workflows on serverless infrastructure since the algorithms are quite complex to implement. Many IT industries have implemented machine learning (ML) because of its use in a range of domains, including image, text, and speech processing. A typical ML workflow includes a variety of user interactions, including preprocessing, training, and fine tuning. These stages are frequently executed by users and require low latency and appropriate resource provisioning. ML users currently encounter several challenges that automatically impede their effectiveness and productivity. For example, users often must configure several parameters at the system level, such as physical topology, memory allocation, number of processors, etc. Furthermore, they have to specify parameters like network structure, learning algorithms to interact with the system. Due to the heterogeneity of resource requirements at different stages, ML users are often burdened with over provisioning resources. ML frameworks that run on coarse grained Virtual machine based clusters lack the flexibility needed to support performance-critical applications Carreira et al. (2018).

1.2 Project Specification

An efficient architecture is necessary for supporting stateful applications on a serverless platform. There is a need for distributed shared memory for state sharing and coordination to satisfy the need for fine-grained state sharing and synchronization. As well, replication of the machine is vital to ensuring durability for applications that require long in-memory state sharing. In this research, we are examining the use of function as a service platform for optimizing applications that need high data transactions.

1.2.1 Research Question

An investigation has been conducted to find a solution to the following research question to improve the efficiency of complex stateful applications on the serverless platform.

"Can the serverless architecture workflow for stateful applications be enhanced using distributed shared memory to reduce network latency for fine-grained coordination and state management between functions on different serverless platforms e.g. AWS Lambda or Openstack Qinling?"

1.2.2 Research Objective

This research focuses on providing an efficient serverless architecture to support fine grained coordination and state sharing in stateful applications. As the resources allocated in serverless are unaware of each other I have introduced the "Distributed shared memory" layer and "Monitoring" in the present architecture to provide coordination and fault tolerance. By adding Distributed shared memory layer, functions can share and synchronize the state more finely. Along with that to minimize the latency for reading and writing, the replication is maintained for the application that requires persistent access to shared objects. If any error occurs during the execution of functions, the monitoring function will read execution logs, to make sure that fault tolerance occurs, and the recovery function will execute to ensure that the function has been executed successfully. An application's error handling process can determine how the recovery function should be configured to decide recovery steps in case of failure. As a result of this approach, stateful applications can be programmed more easily using serverless computing environments, providing a bridge between technical environments and current cloud environments.

1.3 Report structure

Multiple studies related to the research topic are reviewed in the proposed research. As further evidence of the proposed approach's success, it has been implemented. After evaluating the proposed

architecture and comparing the results, the best method is identified. The following is the outline of the research article: various research papers and resources related to the proposed work have been carried out in section 2. In section 3 I have discussed the various tools and approaches used in the proposed methodology. Further, In section 4 the proposed architecture design specification are itemized. Section 5 carries out the implementation of architecture and In section 6 the implemented architecture is evaluated to validate the proposed architecture and the results are compared. Lastly, In section 7 I have concluded the research work and discussed the planned future work and shortcomings.

2 Related Work

The purpose of this section is to choose a suitable platform for stateful applications to be executed and monitored by serverless computing. The subsection 2.1 speak for an overview on serverless infrastructure, subsection 2.2 represents the review on coordination between functions in serverless, subsection 2.3 shows the Review on mutable data in serverless, subsection 2.4 represents the review on fault tolerance, subsection 2.5 shows the review on the stateful application in serverless and subsection 2.6 represents a summary of the related work.

2.1 Overview on Serverless

Serverless Computing, the new paradigm in cloud computing, helps developers develop applications more efficiently. To examine how well serverless functions performed on factors such as memory allocation and obstruction by other users on a cloud platform, Kelly et al. (2020) evaluated servers such as AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions. A python script is used to observe the serverless functions, the functions get triggered via API calls and use external MongoDB database to store response time and logs about the request. Two scenarios are presented in the script: first, two sequential services are invoked and the cold-start and hot-start functions are measured, and second, fifty concurrent services will be invoked as the platform load increases. In terms of resolution of cold start latency and scaling, AWS offers unique functionality container reuse. scaling. Additionally, they indicate that serverless features, such as complete feature isolation, can be modified for edge case applications.

The scheduling of workflows is one of the most challenging aspects of the distributed environment, which is characterized by the need to meet the various quality of service (QoS) requirements. An in-depth study of scheduling strategies in cloud environments was presented by Adhikari et al. (2019). Three types of cloud models were studied, originally the traditional cloud model, the serverless cloud model, as well as the fog cloud model. Workflow models are generally categorized into two groups, synthetic workflows, and scientific workflows. User applications are often designed using synthetic workflows, and datasets are tested using synthetic workflows. A scientific design, on the other hand, is based on a scientific problem and application. There are four kinds of workflows in Serverless workflow infrastructure: queue model, bridge model, direct executor model, and decentralized model. They talked about how a serverless environment, cloud, and fog computing environment offer superior scheduling strategies.

Lee et al. (2018) tested concurrent invocation of functions on major cloud platforms using serverless computing. According to the researchers, AWS Lambda shows more CPU utilization, higher network bandwidth, and faster I/O throughput than other vendors. The authors assert that serverless computing scales relatively well for distributed data processing if the job is divided into small enough tasks, which require 1.5GB to 3GB of memory and take between five to ten minutes to execute. Additionally, they pointed out that serverless computing would be more cost-effective since it can start up instances almost instantly and would only charge for the time spent running the application and not for idle time.

A paper published by Xu et al. (2019) evaluated the performance of adaptive warm-up strategies (AWU) and adaptive container pool scaling (ACP) used in reducing cold start delay. Time series prediction underlies both techniques. According to AWU, a time series prediction model has been created based on function-invoking rules. Once the model has been established, the functions are preheated according to the rules. For ACP, pre-launched containers are placed in a container pool in the event that AWU fails. The ACP backups AWU, while the pool of containers can scale dynamically based on the frequency of function calls. As well as improving the prediction accuracy of the container pool, the researchers tried to reduce resource wastage and reduce resource wastage from the container pool.

Serverless computing consists of servers that do not have any physical memory but can execute independently on a serverless platform that provides elasticity per request immediately. A high degree of flexibility comes at the price of an increased chance of "cold starts". During the provisioning of the

runtime container, the functions are delayed by a certain amount. In their recent article, Vahidinia et al. (2020) examine recent advances in cold start delay mitigation and the state of the art. In addition, several ways were explored for investigating AWS Lambda's behavior as a platform when it comes to cold start latency. Two benchmarks were used, Tensorflow I/O-Intensive and Fibonacci CPU-Intensive, to analyze the average AWS Lambda cold start delay. These findings indicate that Cold Starts for I/O-intensive functions are longer than for CPU-intensive functions. Furthermore, the author explained that cold start delays would only affect a handful of functions if the execution time is short. when the previous instance completes its execution instantiate another instance immediately to avoid cold start delay.

2.2 Review on coordination in serverless

A system named ExCamera was presented by Fouladi et al. (2017) for editing, encoding, and transforming 4K and VR video. Through a coordinator, the system can establish communication between thousands of threads within seconds. In addition to encoding the video in two parts, ExCamera has the capability of dividing it into the slow and fast parts: during the slow part, ExCamera can encode it parallel while the fast part requires serial processing. On a virtual machine in Amazon's EC2, the coordinator operates as a server to execute the job. A worker is assigned to every function that connects it to the coordinator. Each coordinator event is triggered by an HTTP API call. The coordinator utilizes multiple parallel TCP connections to reduce the overhead of communication. Through the TLS connection, the coordination server maintains fine-grained coordination between the participants and updates their state. A message received from the worker is sent to the coordinator, which then processes the request and performs the state transition logic. The compressed video has good quality with low communication overhead and fine-grained parallelism.

An extensible coordination model for applications was described by Distler et al. (2015). An application's state is updated by updating the data objects via a set of operations that maintain coordination. Operation extensions and event extensions are categorized into two types. The client can specifically request operation extension for coordination while event extension can be assigned to the application in response to state changes. As part of their strategy, they work to ensure the extension has the least impact on the system. To accomplish this, firstly the expansion must be verified during registration time, which eliminates overhead during execution, secondly, it must be limited in size in order to speed up verification, and thirdly, it must include methods that ensure that the extension will not negatively impact performance. Four methods are available for efficient coordination of tasks: distributed barrier, shared queue, shared counter, and leader election. They used DepSpace and Zookeeper models to implement and evaluate their results of extensible versions. According to the researchers, they can increase throughput more than tenfold without expanding the kernel size.

IBM-PyWren was a system built upon the PyWren model used by Sampé et al. (2018) that brings new capabilities and supports the execution of globally distributed map-reduce jobs. In addition to the Python notebook integration, data partitioning, data discovery, and dynamic composition are new features. MapReduce use cases were used to test their model. In the findings, serverless platforms showed speedups of more than 100X in distributed computing, and some new evidence for serverless platforms' ability to perform distributed computing.

The distributed computing framework in serverless computing was evaluated by Jonas et al. (2017). In a current computing environment, the model is similar to bulk-synchronous processing. Using AWS Lambda, they deployed PyWren alongside a separate server called parameter server for management of coordination between functions. To store data on parameter server kay-value pair is used like in Redis and RAMCloud. As an important feature of serverless model PyWern, parameter servers act as a bridge for stateless functions to communicate. The key-value stores should be able to support server-side scripting to maintain flexible consistency and synchronized updates. Their final conclusion was that serverless platforms can provide abstraction and elasticity from backend services for distributed applications.

In a paper titled numpywren, Shankar et al. (2018) proposed a serverless linear algebra system. Their system consists of mainly five components, task queue, runtime state store, global task scheduler, serverless computing, and object store of distributed data. The first stage for any request is task queue, They have to wait for the task scheduler to pick and execute them. After completion of the task, immediately the runtime state of the task has to be updated in the state store. For state sharing at runtime, they used Redis in-memory database to support atomic writes and fast access. The executor will add a child's task to the task queue if they are "ready" to perform a completed task. The state store is designed to be atomic for every child. They also deploy linear algorithms like Cholesky decomposition,

matrix multiply, and value decomposition. Results generated show that numpywren takes 33 % longer to execute than Scalapack, and requires 240 % more CPU power because of the fault tolerance and elasticity of serverless. They showed that how minute changes in the infrastructure enable complex stateful computations and improve workflow.

2.3 Review on mutable data in serverless

Among the many serverless functions proposed by Pu et al. (2019), they achieved low latency and minimized operational costs using a serverless analytics system. An analytics application, for instance, makes use of serverless technology to achieve elasticity. However, implementing such an application is a challenge because it requires a large amount of data to be shared between functions. To address latency issues caused by storage, the "Locus" model combines fast and slow storage together. Storage is selected on the basis of bandwidth, throughput, and the number of Input/output operations per second. They have a major impact on storage performance. This feature enhances machine applications by reducing the access to data uniformly in AllReduce and MapReduce. In addition to producing good results, the model is also cost-effective. In the model, intermediate data is partitioned and stored in Redis, while results are combined in S3 and stored in S3. They used big data benchmark, cloudSort, and TPC-DS to evaluate their model. According to the result, it is evident that Locus is achieved comparable performance to Apache spark. Though, their model is 2X slower as compared to Redshift.

In Sutra et al. (2017), a design CRESON was introduced, which facilitates the use of shared objects that can be callable and replicated over a NoSQL database. By allowing objects to be shared and persistent, applications become simpler to design. When multiple clients access the same object repeatedly, the object has to be converted between serialized and in-memory format. This leads to increased replication costs and decreased performance. Objects created using the CreSON design are directly instantiated and mapped to storage nodes to address this problem. In order to create and manage the replication of objects, state machine replication has been used. Using a key-value store with listenable data abstraction in the NoSQL database, their approach supports strong consistency as well as access and location of objects using their reference point reference. With disjoint access parallelism, multiple concurrent requests can be made on the shared objects. Objects are mapped between replicas using consistent hashing. Implementing the StackSync application has allowed them to compare CRESON's performance with PostgreSQL. To test the performance, they have used real and synthetic workloads. Compared to conventional models, Their model offers a simpler and faster way to implement with less SLOC to program. Their model also outperforms PostgreSQL in terms of scalability and average time to access the data.

Lambdata is a design created by Tang & Yang (2020) that can make an explicit call to data for serverless computing. There are limits to what can be cached in the existing serverless system because it treats the cloud as a black box, and it cannot determine whose functions are reading and writing the data. Several key design elements of Lambdata enable temporary data locality, data pipelining, and spatial data locality to store intermediate results so that they can be passed from one function to another. Developers are able to explicitly specify whether data will be read and written using the system. Several functions can be concomitantly written to handle mutable data. If multiple functions are run simultaneously, the cache of the first executed functions will be deleted using the eviction policy after each function has been completed. As part of Lambdata's effort to improve performance, lambdata prefetches and caches data locally while it also allows for the scheduling of functions based on local data and code. As a result of Lambdata's evaluation, it was found that it reduced execution costs by 16 percent and increased speedups by 1.51 times.

The author Klimovic et al. (2018) has addressed, issues related to the intermediate exchange of data in applications like machine learning and data analytics. During their research, they identified the difficulty of meeting the I/O needs associated with ephemeral data storage. Irrespective of the size of the object the data store must deliver high IOPS, high bandwidth, and low latency. The authors proposed a small and elastic data store, named Pocket, which would meet the selected performance requirements with minimal cost and flexible scalability. With the pocket's dynamic storage resource rightsizing, you can achieve high I/O and high performance with low cost to store resources such as CPU cores, storage capacity, and network bandwidth. There are three basic principles in the design: placement of data, separate responsibilities, and tracking data between nodes. They can scale up and down easily, and store only data, not metadata, with a response time of sub seconds. Finally, the storage system provides multi-tiered storage in order to meet input/output demand without affecting the cost. In addition, they compared the performance and cost of the Pocket storage with that of the S3 and Redis in-memory

stores. At approximately 60% of Redis' cost, Pocket completes execution within almost the same time frame as Redis.

2.4 Review on fault tolerance

According to Sreekanti et al. (2020), a serverless application can be supported by atomic fault tolerance (AFT). Using AFT, function as a service platform and remote storage can perform atomic reads and writes. In addition to atomic reads with garbage collection for reducing overhead for atomic reads, it supports new protocols in serverless to ensure isolation for atomic reads. By providing read-your-writes and repeatable read property, the model prevents dirty and fractured reads. The write order protocol provides fault tolerance by detecting potential AFT failures. An aborted transaction, discarded updates, will result when a function fails. The function restarts with the same transaction ID. There are two cases in which AFT has failed. The client must re-perform the entire transaction if the transaction fails before completion. The committed transaction metadata is retrieved from the data store if the transaction is committed. Besides guaranteeing the liveness of transactions, the ordering protocol is also guaranteed. When a replica commits a transaction and then does not broadcast the commit to other replicas, the client will receive an acknowledgment. Since the other nodes are unaware of the commit, it appears as if no transaction was ever made on the client side. Their model has the ability to handle thousands of requests per second with minimum overhead. In the case of S3, the AFT imposes an overhead of 25%, while if it is used with DynamoDB it increases the performance by 18%.

Several systematic studies have been conducted on fault tolerance by Hasan & Goraya (2018). Providing trusted cloud services is a major challenge, and fault tolerance is an important aspect. Cloud computing is difficult to implement because of its dynamic service infrastructure, complex configurations, and diverse interdependencies. It is not only necessary to have a thorough understanding of cloud applications but also to analyze various prevalent methods to implement a fault tolerance policy. There are two main types of faults. Failures resulting from the abnormal operation or a failure of several components of the system are known as crash faults, whereas failures causing unpredictable conditions with ambiguity are known as byzantine faults. Researchers use proactive and reactive approaches to implement fault tolerance. The interruptions like errors, failures, or faults can be handled proactively. There are three ways to achieve a proactive approach: self healing capability, migrate pre emptive computations, and backup data periodically. When a fault is encountered, the reactive approach responds rather than anticipates it. In contrast, the proactive approach examines system behavior. The reactive approach can be also achieved in three ways: by restarting the checkpoint, migrate jobs when resources crash, and replicate the instance. The reactive approach is mostly used for fault tolerance as it handles both byzantine faults and crashes.

A trigger-based orchestration workflow was presented by López et al. (2020) in a serverless cloud environment. Kubernetes and k native eventing technologies are used to build the architecture. Event-condition actions are used to implement Trigger services. There are different sources of events in the cloud, such as Kafka, times, object storage, and etc. A Container, VM, or Function can be configured for the execution of a specific action. TriggerService provides a shared repository for persistent contexts with fault tolerance and durability. This design provides fault tolerance by incorporating context components. As part of the execution cycle, it stores trigger states in a form of key value fault tolerant data store. Triggers can also be activated or deactivated dynamically, based on dynamic changes of trigger states. Message delivery and detection of unrecoverable errors are ensured "at least once" by fault tolerance. Worker state is restored congruently if a TF-worker fails. Fork-joint triggers can be used with the persistent context to detect and count composite events, which are also detected and counted in the persistent context. Their model can optimize the performance of the scientific application and can scale upon demand to meet the resource needs.

For serverless computing functions, Kaffes et al. (2019) developed a centralized scheduler to manage clusters. The queue imbalances can be managed to reduce interference in the cluster. A serverless computing model should share resources with other workloads so that cloud computing can be cost-effective. Serverless functions and cloud workloads must have separate cores, and effective mechanisms need to be put in place to partition them. System design does not provide fault tolerance. In addition to the scheduler, a framework runs over it that implements fault tolerance. In event of failure, the scheduler has to check and recover only the idle workers. This approach can be implemented by exchanging meta data for all worker servers with each other. Centralized scheduler approach is beneficial for the application those are sensitive to throughput and latency.

2.5 Review on complex stateful application in serverless

Fouladi et al. (2017) describe ExCamera, a data analytics system that is capable of editing, transforming, and encoding low-latency 4K and Virtual reality videos. Cloud-based ExCamera enables interactive video applications built around video processing in a massively parallel manner. To render any video the user has to just select the video and start the query. The goal of the application is to render the video in less time saves the result in any cloud editor like google docs. The Excamera proposes two modifications: parallelly execute cloud function with inter thread communication between thousands of threads. The billing for this is on incremental usage per second. The second step is that video encoders split videos parallelly into two parts. There are two parts, one slow, that can be encoded parallelly, and one quick, that needs to be run serially. The coordinator server is hosted on EC2 and it triggers events for each function using an HTTP API call, and workers connect the functions to the coordinator. By requesting multiple parallel TCP connections, the coordinator reduces communication overhead. TLS prevents unauthorized access to the coordination server so it can keep track of fine-grain coordination. when the client request to the coordinator, the request is served by the remote workers with transition logic. ExCamera has been tested for its ability to efficiently encode videos and they found that it can invoke 3,600 cores in 2.5 seconds, which is approx 9 teraflops. To encode animated 4K movies to the VP8 format the video compression thas bout 15 minutes. In addition, they also minimized the overhead of communication without affecting the compression quality of the video.

PyWren, the distributed computing framework presented by Jonas et al. (2017). The design focuses on simplicity and elasticity to provide support for component blocks and access to remote storage. End-users will be able to program and deploy stateless functions easily. Users provide inputs, and the design submits functions that are single-threaded and can fulfill data dependencies at runtime. During the function execution, containers are created and can improve the performance if reuse. For these types of designs, outputs and inputs must be stored in a remote location. APIs have been proposed, to execute the tasks in parallel using existing libraries. In order to manage the coordination between AWS lambda functions, they used Aws lambda with parameter servers. The server deployed has the same data store structure as RAMCloud and Redis. Consistency is maintained between the functions executing parallelly by updating their key-value instantly. A PyWren application can develop complex abstractions using the following feathers: parameter server, map reduce, and monolithic reduce. A balance of resources, distributed data store, scalable scheduling, and pricing were considered when designing high-performance data processing. The benchmark they have used is matrix multiplication, read/write on data store using key-value pair, and response time form S3. As the number of workers increases, linear scaling is observed.

To address flexibility and performance issues in Reinforcement learning Moritz et al. (2018) has proposed a distributed system named Ray. Artificial intelligence applications in the future will be adaptive and constantly learning from their surroundings. Due to the new and exhausting system requirements, the applications require high performance as well as flexibility. Ray supports both actor based computations and task level parallelism with a single execution engine. Ray provides load balancing, failure handling on remote workers with object locality. Stateful computations executed in a serial fashion are referred to as actors. Load balancing at a coarser scale is maintained. Using distributed planning, fault tolerance, and a supplier, Ray's system will meet the performance requirements of his system. They proposed a distributed scheduling method based on bottom up metadata storage to achieve fault tolerance and scalability. Taking into account that AI workloads require a flexible programming environment, high outputs, and low latency, this architecture is especially relevant in workloads that differ significantly in terms of resources, durations, and functionality. As a result of our evaluation, 1.8 million tasks can be executed linearly within a second, fault tolerance is transparent, and the performance is significantly enhanced with a variety of contemporary workloads. In addition to its flexible and easy-to-use nature, Ray provides a high-performance and intuitive AI application model.

Cirrus, presented by Carreira et al. (2019), enables end-to-end workflows for machine learning. The model uses S3 and AWS lambda to provide scalability with minimum user efforts. ML models generally consist of three steps: preprocessing data, training the model, and tuning hyperparameters. A client and a server are the components of the design. There is API support to execute machine learning tasks and functions to allocate tasks on the client server. The server is responsible for training the data and storing the intermediate processed data in an attached data store. The following refinements are done in cirrus design: adaptive resource allocation to overcome over provisioning of resources, detached backend to work efficiently in failure, and restarts the executing when resources are up again. The client-side backend is responsible for mapping and managing computing units to execute a function. A serverless API that runs preprocessing and tuning through an end-to-end process. Due to the high computation intensity of Machine learning workflows, they require more computation power to execute. Citrus supports the

concurrent execution of thousands of workers with high scalability. According to their evaluation, cirrus performance is better than other specialized frameworks while training data. The model completes the execution in less time and utilizes less money compared to other frameworks. AS compare to Tensorflow it is 100x faster and 3.75x to Bosen framework.

The framework Ripple, presented by Joyner et al. (2020), allows processes to be executed on single computers while utilizing parallelism in tasks. Ripple has a user friendly interface that can handle any application, proteomics, genomics, analytic, and etc. Its fault tolerance is achieved by detecting tasks with failure, automatically supplying resources, and meeting user defined QoS goals. The data flow between functions is managed using eight primitive functions. Function's basic jobs are to combine files, split files, map keywords or each item in a file to match, partition files, and sort files. There are multiple ways to use each function during different phases of execution. AWS interprets the JSON file generated by Ripple and uploads it to AWS for further processing after it complies with the user's input. Their design consists of separate phases, a multiphase job, and a single phase job designed to address resource provisioning. Ripple uses two steps to determine concurrency levels. It allows developers to choose First in first out, Round-robin, and Priority-based scheduling policies. In parallel jobs, fault tolerance can be achieved by reading logs from Lambda to determine which job to run next based on their recent write in S3 and re-starting the job. If any thread got terminated or did not able to access S3 at his execution time, then also it re-starts the execution. A few applications were developed using Ripple and PyWren Jonas et al. (2017), including Proteomics, SpaceNet, and DNA Compression. Ripple maintains the concurrency and meets the performance need for any application with an overall 80x improvement in performance at the same cost.

2.6 Summary of literature review

Reviewed Studies	Area	State Store	Replication
P. Sutra et. al. (2017)	CRESON: callable and replicated shared object over NoSQL	Listenable key-value store, NoSQL storage	Yes
Jonas et. al. (2017)	Pywren: Distributed computing framework for data processing	S3	No
Fouladi et.al. (2017)	ExCamera: Video processing with low latency using threads	Finite-state-machine, EC2	No
Moritz et. al. (2018)	Ray: Distributed framework for AI application (performance and flexibility)	Global Control Store (Key value store)	Yes
Carreira et. al. (2019)	Cirrus: End-to-end ML workflow to minimise cost	Parameter server, EC2	No
Joyner et. al. (2020)	Ripple: Task level parallelism with single machine	S3	No
Proposed Approach	Serverless architecture with distributed shared object layer	Distributed Shared Memory	Yes

Figure 2: Summary of Related Work

In serverless computing, researchers are working to address fine-grained coordination and to share intermediate states with minimum latency. Serverless is widely adopted in cloud industries because of its resource abstraction and developer can focus solely on development. Many authors have evaluated the cloud providers performance and most of them found AWS outperforms other providers as mentioned in Kelly et al. (2020) and Lee et al. (2018). To manage coordination, parameter server has been deployed in EC2, NoSQL, S3 and in many more as discussed in Distler et al. (2015), Fouladi et al. (2017), Jonas et al. (2017), Sampé et al. (2018), Shankar et al. (2018). With coordination their is a need to share

state of intermediate data is mentioned by the authors Tang & Yang (2020), Pu et al. (2019), Klimovic et al. (2018) and Sutra et al. (2017). Further, I have talked about the fault tolerance as shown in papers Hasan & Goraya (2018), Sreekanti et al. (2020), Kaffes et al. (2019), López et al. (2020).

3 Methodology

3.1 Tools and Techniques used

Before configuring lambda, create the Identity and Access management role with the following permissions as shown in figure 3.

- AWSLambdaFullAccess
- AmazonS3FullAccess
- CloudWatchLogsFullAccess
- AWSLambdaVPCAccessExecutionRole
- CloudWatchEventsFullAccess
- AmazonEC2FULLAccess

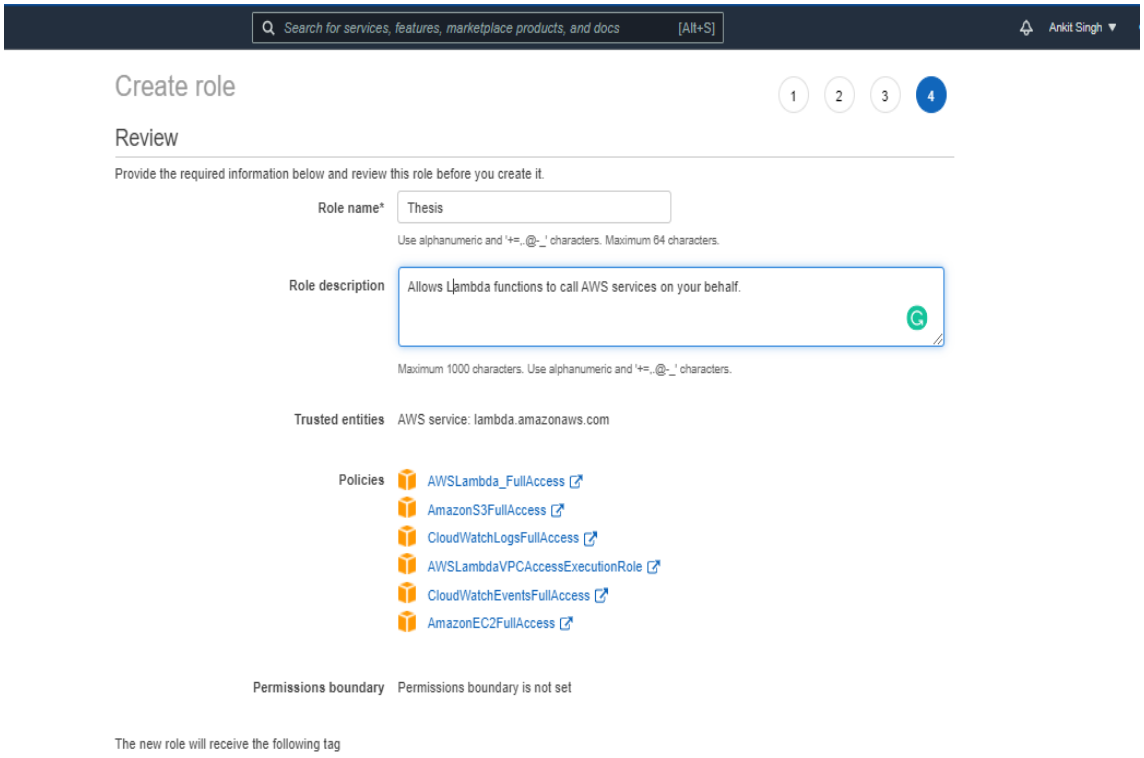

Figure 3: IAM role created to access the lambda and other services.

Then Create VPC group with several subnets, Ec2 instance of r5.2xlarge, S3 bucket and lambda function with access to s3 bucket.
In Ec2 install docker and java to deploy distributed shared object layer.

3.2 Proposed approach overall Structure

In the proposed approach FaaS is seen as a unified set of tasks consisting of a set of shared "cloud threads" communicating through a shared state. To achieve the shared object state, distributed shared object layer is implemented atop Infinispan. It is an in-memory data store with low latency to deploy distributed serverless applications. The High-throughput and low latency of DSO is the best substrate

for shared objects to share and synchronize their state. Further, it ensures durability through replicating the shared objects.

3.2.1 Programming model

Abstraction	Description
CloudThread	Lambda functions are treated as threads
Shared objects	Wait free distributed objects (AtomicInt, Atomic Long, List, Map , etc)
Synchronization objects	Provide primitive for thread synchronization - CyclicBarrier
@shared	User defined Shared objects (.add(), .merge(), .update(), etc)
Dara persistance	Object that needs to resides for long time in the memory (@shared(persistence=true))

Figure 4: Programming model abstraction

The proposed architecture follows an object based programming model and we can integrate it with any object oriented programming language. I have used Java to implement this approach. 4 showed the jargons used in the architecture. It summarises the abstraction used in the programming to develop the DSO layer.

Cloud Thread Programmers can write the multi threaded, object oriented java program in a distributed stateful application using Cloud threads. They only have to do two necessary modifications to execute their java code on top of the serverless model. First, each object should be treated as a cloud thread and second, the mutable shared object should be replaced with counter part of the DSO layer so, that it can be shared between cloud threads.

Fine-grained State Management To support mutable, cloud-based shared data, the proposed distributed architecture provides a library of common objects, such as integers, maps, lists, arrays, and counters. The shared objects can be linearizable, and they are also wait-free across the cloud threads. Despite concurrent access to shared objects, every method-invocation is completed in only limited steps, and all requests are addressed by a single thread. Using the @Shared annotation, programmers can customize shared objects. To refer to a shared object, it uses the shared object's reference to locate the object in the Cloud Storage Object layer. Objects decorated with the annotation @Shared are globally accessible and can be shared between cloud threads. Shared keys are by default associated with the field name, so the programmer can alter this value based on their need by using @shared(key=k).

Data Persistence There is the option to have persistent or ephemeral shared objects in the architecture. By default, ephemeral objects are shared and they exist till the lifetime of an application. Once execution finishes these objects are discarded. In the event of failure with DSO layer failure, ephemeral objects are also lost. Making ephemeral objects fault-tolerant will out-cost its benefits of short term availability. To make it cost-effective we can make the objects persistent by denoting them with @shared(persistent=true). When the application completes its execution these objects can be destroyed explicitly.

Fine-grained synchronization Serverless supports only bulk synchronous parallelism that is uncoordinated and stateless. Several primitives, such as future, semaphores, and cyclic barriers, are available for synchronizing cloud threads. Cycle barriers make it possible for threads to wait for each other till they reach a common barrier point and then reuse the barrier after this waiting period is over. Serverless functions can be executed concurrently in this model. The java.util.concurrent.CyclicBarrier *CyclicBarrier class* (2020) is equivalent to cyclic barrier elementary for fine grained coordination. The cyclic barrier follows a breakage model that involves all or none of the synchronization stages. Due to an interruption, timeout, or other unexpected events that cause the thread to leave the barrier point, the BrokenBarri-

erException ensures that other threads will also leave the barrier point if the thread untimely leave the barrier point.

3.3 Sample application

I have implemented a simple multi threaded program of Monte Carlo simulation. The basic work of Monte Carlo simulation is to approximate the value of pi (π). Random points are drawn in large numbers and only the points which fall in the unit square area of the circle are computed. As the number of trials increases toward $\pi/4$, the large number of points will fall in the circle. This application is implemented with the runnable class to estimate the value of π as shown in figure 5.

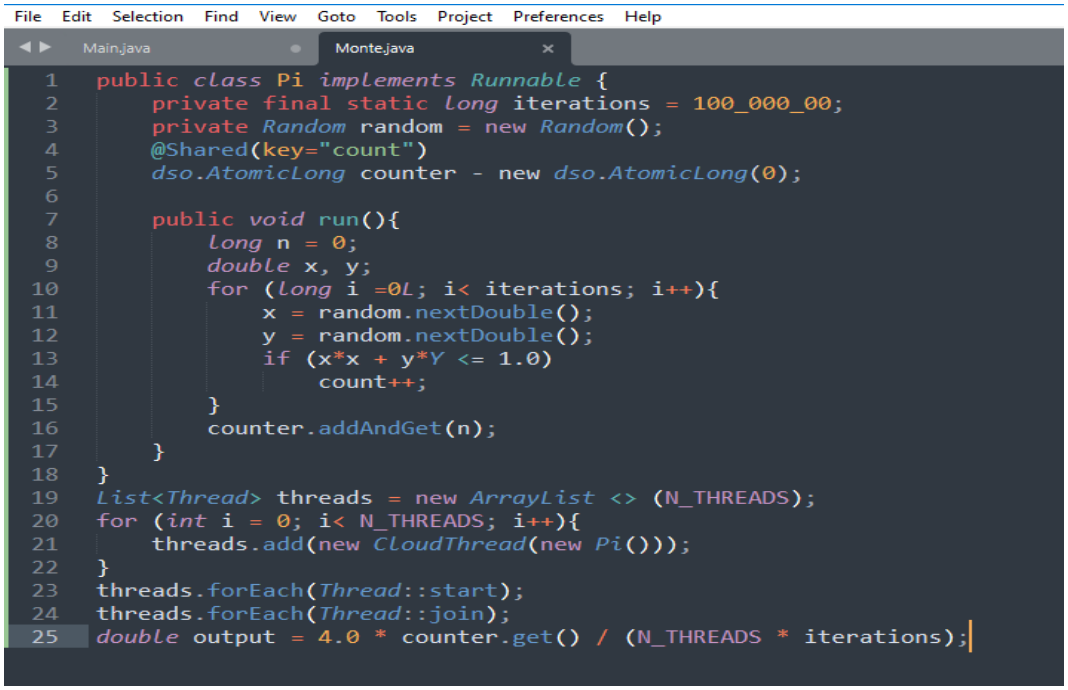

Figure 5: Monte Carlo simulation on serverless

4 Design Specification

Figure 6 illustrates how a distributed serverless application utilizes a distributed shared object architecture on serverless. There are several stages and layers in the architecture. Four main components make up the architecture:

1 Client application

2 Serverless computing FaaS layer

3 Distributed shared object

4 Recovery function (fault tolerance)

4.1 Distributed Shared Object layer

Serverless distributed applications are deployed together with the DSO layer. The fine-grained coordination and state management are enabled by an in-memory, low-latency data store. The fine-grained coordination and state management are enabled by an in-memory, low latency data store. Coherence across serverless threads is simplified with this layer, which guarantees strong coherence and simplified the global state semantics. Due to its ability to manipulate the global state of the remote object, it limits mutable state management to a virtually infinite extent, however only the language's flexibility dictates

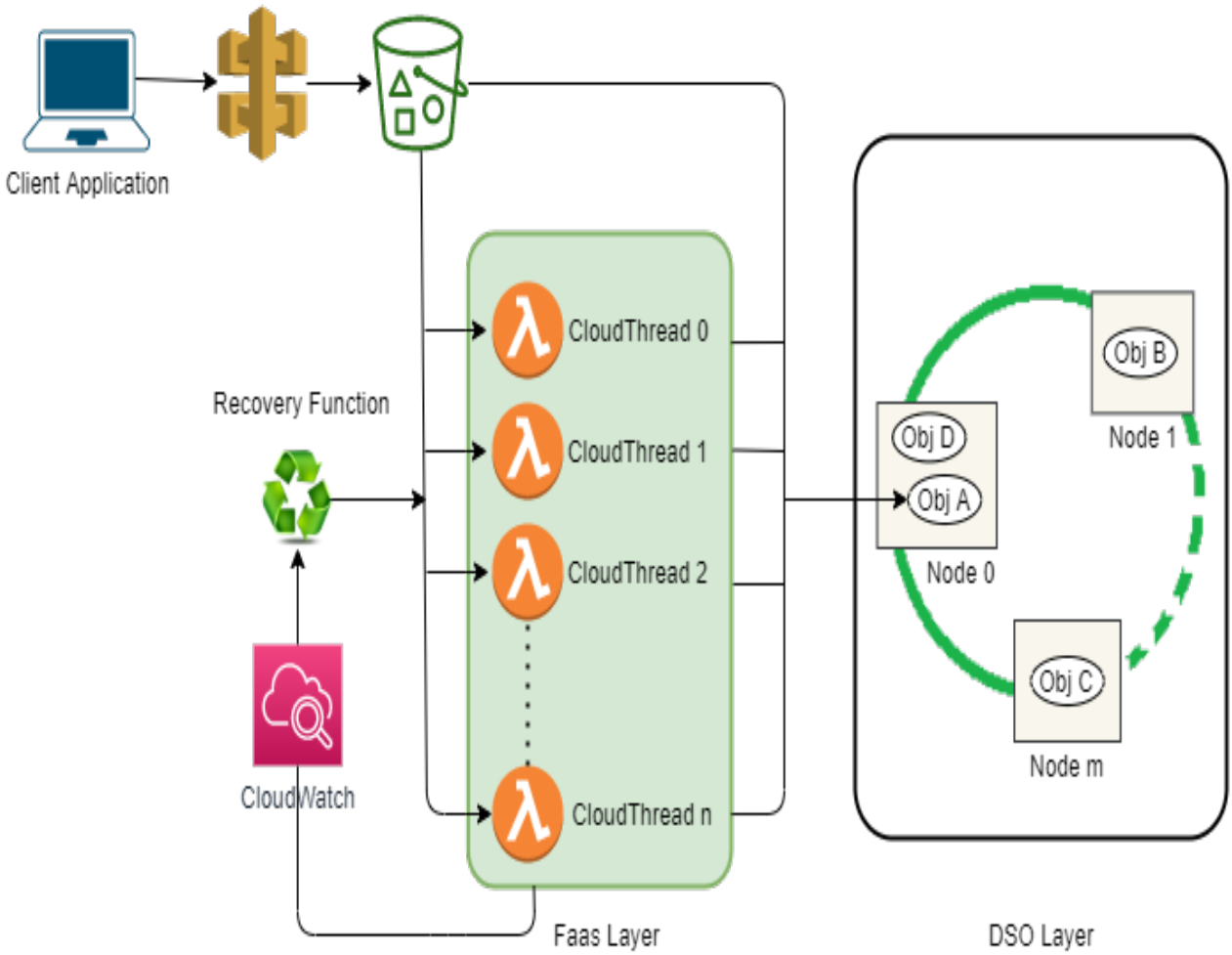

Figure 6: Proposed Machine learning application architecture on serverless

how far it goes. DSO is a layered database containing integers, maps, arrays, counters, and lists of shared objects. By invoking methods on shared objects, updates to shared objects can be made. References can be used to uniquely identify objects within DSO. The object of type P can be referred to with (P,k). k is the shared key with annotation @shared(key=k). The cloud thread invokes shared objects using its object reference to invoke their appropriate methods remotely. Similar to the decentralized structured storage of Cassandra Zhou et al. (2018), consistent hashing shall be used for the DSO layer. Since every storage node is an individual member of the storage layer, mapping from data to nodes can be done easily. The shared object can also be located this way. A hash of reference point (P, k) can be used to determine the location of object O. By using this approach, architecture benefits in many ways, such as the elimination of broadcasting to find an object's location, securing disjoint access parallelism, and minimal interruption of service upon server failure or when an additional server is added.

Persistence The DSO ensures the durability of objects that are part of workflows that require persistent access to the shared data. In the DSO layer, replicas of persistent objects are created. The replication factor (rf) is used to determine how replications are created. Using standard mechanisms like marshaling to move to stable storage, you can delete the replicas once they are stale. They reside in memory to reduce read and write latency and minimize storage costs. Every time it needs to access a shared object, the cloud thread calls the server node. The replicas in the DSO layer execute the requested operation, compute all the results and send it back to the requested source.

Consistency DSO layers enable stateful distributed applications in serverless environments by combining shared memory and linearizable shared objects. Replication of state machines (SMR) Alchieri et al. (2018), is a method for keeping persistent objects consistent across replicas. Using virtual synchronization Dolev et al. (2018), the DSO layer provides server nodes with an ordered view of the replication

state. Objects are multicasted to access them, and multiple replicas are created to reflect changes. The caller function will receive the results after the changes are re-balanced in the nodes.

4.2 Fast aggregate through method call

Functions run separately from data in the Fass architecture and applications must send data from the data store to the execution code. By implementing methods for multiple-precision arithmetic, developers can overcome this downside of the FaaS platform, with minimal effort. Applications where small data granules must be aggregated and combined, such as machine learning pipelines, benefit greatly from this ability. Methods of the shared objects are executed remotely, over the distributed memory layer, as a means of minimising communication overhead. Due to the remote execution of object methods, the overhead and resources of communication are minimized. At present serverless architectures have to pull data from S3 or any other slow storage and then perform an operation on them like AllReduce. These requests to pull data, add communication overhead and cost in N^2 messages, where N is the number of functions. In the proposed architecture the DSO layer will aggregate the data remotely and apply the operation on them and send back only the result to skip reduce phase. This has reduced the complexity of O(N) messages.

4.3 Recovery function

Both FaaS and distributed storage layers in the proposed architecture are fault tolerant. Persistent access to shared objects is enabled in DSO by replicating them. A system that can handle rf - 1 server failures can be configured to handle successive failures. By default, Cloud Watch monitors the AWS cloud threads, and when a cloud thread fails to execute, the recovery function will activate. The latest logs are extracted by cloud watch. It will try to re-start and re-execute the failed threads with the same inputs if any CloudThread fails to execute. The recovery function will then be triggered and the state of the DSO layer will be updated. The developer can explicitly program how many retries to perform and the time interval between each of them in the recovery function. A programmer must also ensure idempotency when reattempting. DSO's atomic writes facilitate this process. For example, in K-means clustering they require to share their iteration counter at every state and if any thread fails to execute, the other threads can extract the state of that thread from the DSO layer.

5 Implementation

The proposed architecture application can be written in any object oriented programming language. I have written the application using JaVA with Maven for compilation and managing the dependencies. Runnable is used to implement cloud threads with abstraction shown in figure 4 while execution. TO compute these cloud threads I have used AWS Lambda serverless computation engine. Lambda-maven-plugin3 is used to deploy the AWS lambda functions and these functions are get invoked through Java SDK provided by AWS. I have used Synchronous requests and responses to control invocations of the request. Whenever the Lambda function receives the request, it also receives the Runnable name and parameters to be initialized in the payload. To provide initialization values and instantiate classes in the FaaS layer Java reflection API is used. Before executing the application code the FaaS layer establishes the connection with the DSO layer to store and execute remote data. The return payload received from the DSO layer is empty as the proposed architecture accepts Runnable only. In the event of failure or any error may occur the received payload has the logs that can be interpreted and exceptions are thrown.

DSO layer is built atop of Infinispan Marchioni & Surtani (2012) in-memory database. The Source line of code (SLOC)of server and client code on the DSO layer weighs 9.2k and 2.5k respectively. The client application has a library of shared objects that can be accessed through cloud functions. The library on the client application is compiled using AspectJ Kiczales et al. (2001). The user defined instantiated shared object should be serialized with an empty constructor to serve marshaling needs. Synchronization of the object is done using Cyclic Barrier and also relies on counterparts of Java like method call to block, monitor, or using a combination of notify()/wait(). In a cyclic barrier, the process waits till other threads reach the barrier point and synchronize them. The implementation of state machine replication is done on top of infinispan using its interceptors API Marchioni & Surtani (2012). SMR uses the visitor pattern of the storage system to manage the replicas. In the proposed architecture the storage layer is explicitly managed while the automatic resource provisioning for storage in serverless is an open concern Jonas et al. (2019).

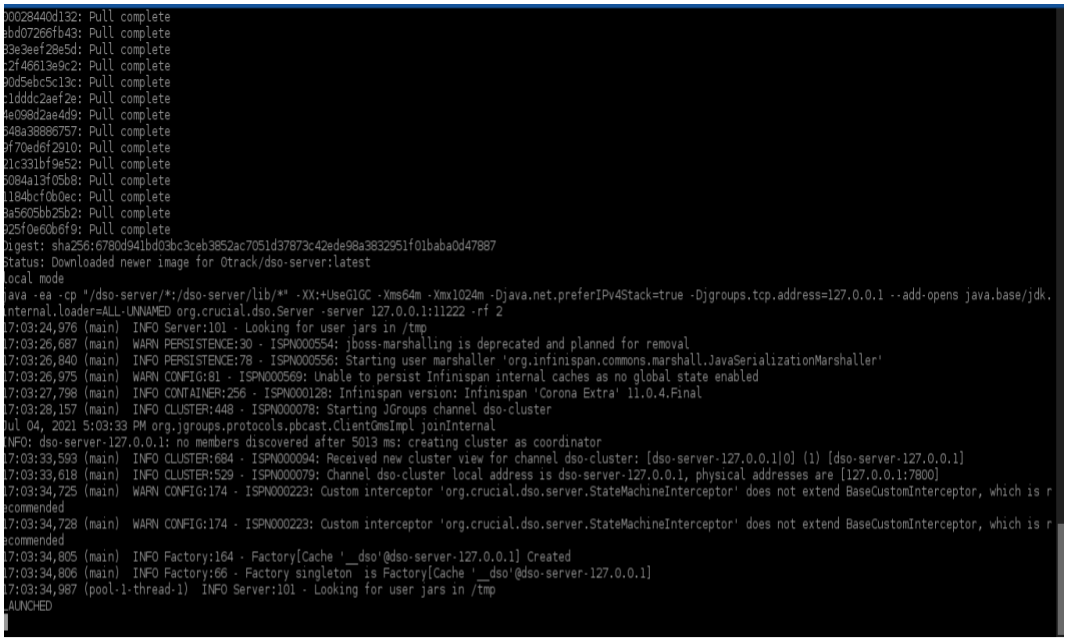

Figure 7: k-means, machine learning implementation on proposed architecture

6 Evaluation

This section shows the experimental results. Firstly I, have validated the proposed approach on micro-benchmarks like latency, throughput, and parallelism. After that, I have implemented the k-means machine learning algorithm to evaluate state management and later I, have compared the proposed architecture with AWS spark.

Evaluation setup Amazon web services are used to run all the experiments. Distributed shared object layer is deployed within a virtual private cloud on r5.2xlarge elastic compute cloud instance. Machine learning application, k-means is executed on AWS Lambda with maximum available resources. Figure 7 shows that the proposed architecture DSO layer is successfully configured and launched to perform experiments.

6.1 Experiment 1

In experiment 1 I, have evaluated the proposed approach on micro-benchmarks latency, throughput, and parallelism.

6.1.1 Latency

Table 1 shows the comparison of average latency to access 1 KiloByte of payload in Distributed shared object, Amazon simple storage service, and Redis in-memory database. I have compared the proposed DSO layer with Redis because it is the most popular key-value data store. I have experimented 7 times with 20k operations for each function executed and calculated the average latency to access the data store. As shown in Table 1, it is evident that DSO is better than AWS simple storage service and has similar performance as Redis database.

Type	PUT	GET
DSO	221 µs	209 µs
S3	33,900 µs	22,100 µs
Redis	222 µs	207 µs

Table 1: Average latency

6.1.2 Throughput

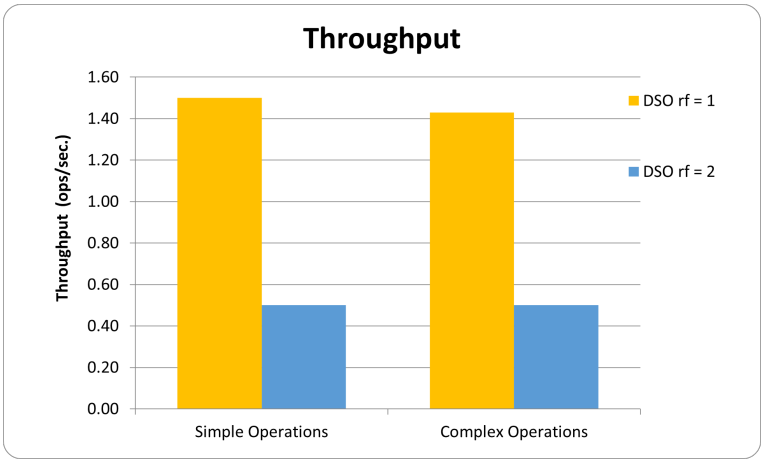

Figure 8: Throughput of the proposed architecture with and without replication (ops/sec in 10^5)

Figure 8 shows the comparison of throughput for distributed shared object layer with replication and without replication. This experiment carries out throughput for simple operations and complex operations performed per second. I have executed 200 cloud threads that access 500 remote objects at a random basis in a loop. The remote object contains an integer and performs basic calculus operations. simple multiplication is done to calculate throughput in simple operation while in complex operation I, have executed 1k multiplication sequentially using cloud thread. It is clear from the figure 8 that the performance of DSO is not affected by the complexity of operations performed. It only depends on whether the replication factor is enabled or not.

6.1.3 Parallelism

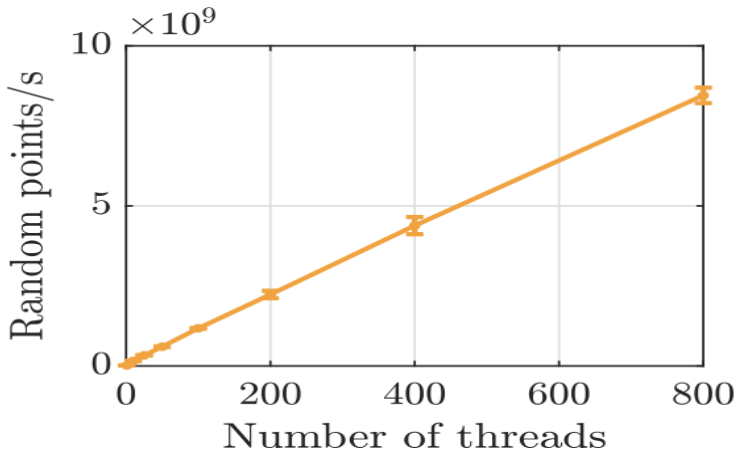

Figure 9: parallelism of application using Monte Carlo Simulation

I have implemented Monte Carlo Simulation to approximate π Yavoruk (2020) to show how well proposed architecture scales with an increase in cloud thread from 1 to n. The base algorithm of Monte Carlo is fully parallel as shown in 5. I have executed the experiment to 1 to 700 threads and noted down the number of random points it computes in a second. All the readings are in 10^9 random points per second. The proposed architecture reaches 7.6 billion points in a second with 700 threads. It is evident from the figure 9 that the system linearly scales with an increase in cloud threads and demonstrates 380x speedup when cloud thread reaches 700.

6.2 Experiment 2

Figure 10 and 11 shown the implementation of k-means clustering algorithm on proposed architecture. The k-means clustering algorithm first initializes the centroid by shuffling the user dataset and then selects the k point for centroid randomly without replacement. The data points are computed and the sum of the squared distance is taken till the centroid stop changing its position. In the proposed architecture k-means uses mutable shared objects to store centroid points and k points. The synchronization of objects for coordinating the iterations is done using a cyclic barrier on line number 15. All the method calls to perform the remote operation in distributed shared object layer are done using global method calls.

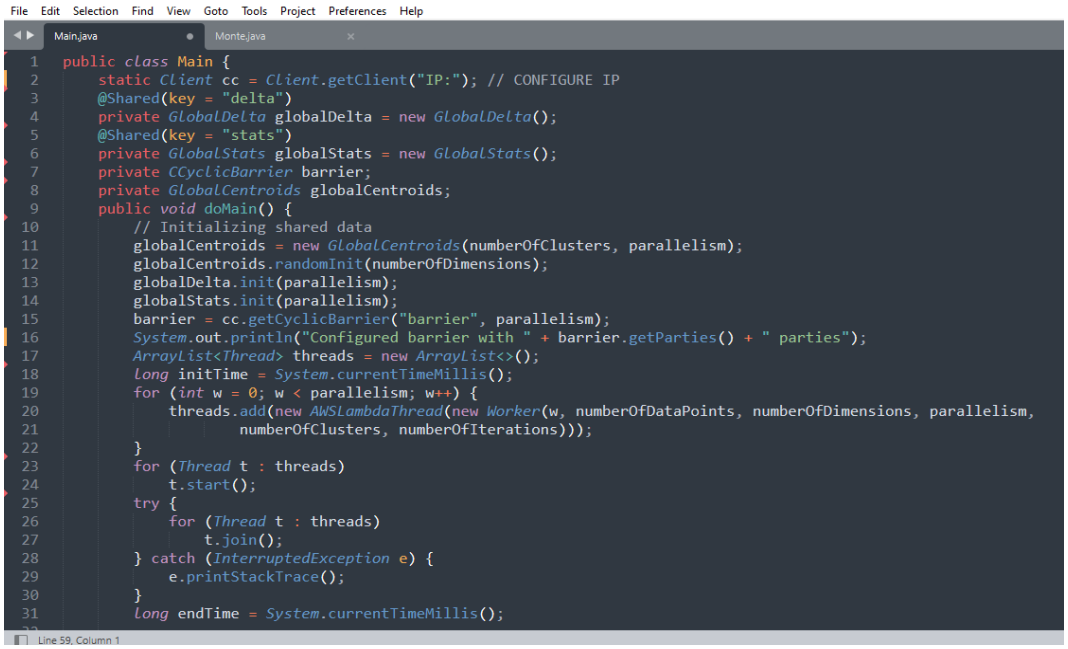

Figure 10: k-means, machine learning implementation on proposed architecture

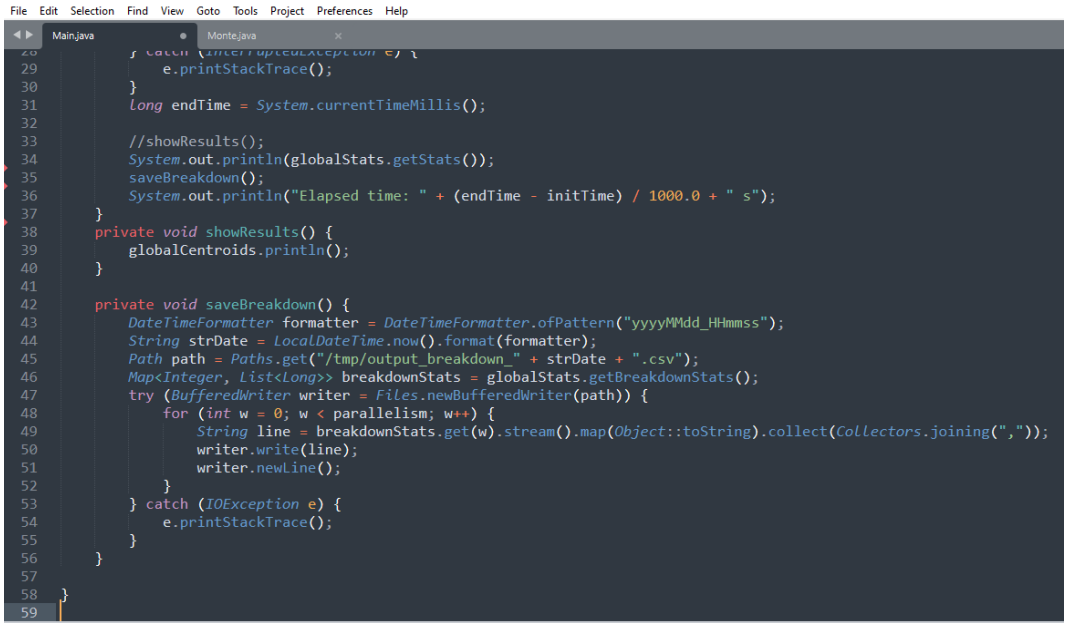

Figure 11: k-means, machine learning implementation on proposed architecture

Figure 13 compares the performance of proposed architecture with elastic compute cloud instance having configuration m5.2xlarge with 8 cores, m5.4xlarge with 16 cores. It appears that input increases

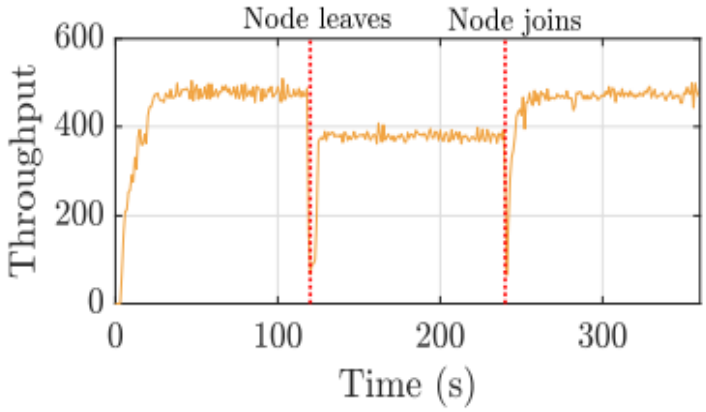

Figure 12: Throughput of the model when k-means algorithm executing.

relative to a thread count in this experiment. I have measured the scale up with the increase in the number of threads using scale up = T1/Tn. Here, T1 is time to complete the k-means execution on proposed architecture as shown in Figure 10 and 11 with 1 thread. Tn, where n is the number of threads. When scale up is one, that means architecture is scaling perfectly. With an increase in workload size, the number of thread also increase parallelly. When scale up is less than 1 the graph is sublinear. Further, when threads in the application are more than the number of cores in a single machine the performance degrades quickly. In comparison to m5.2xlarge and m5.4xlarge, the proposed distributed shared object layer has 10% more optimum performance. When the cloud threads are 160 the scale up is approximately 0.94 and it decreased to 0.9 when cloud threads reach 320. The scalability decreases because of the overhead imposed for creating the threads.

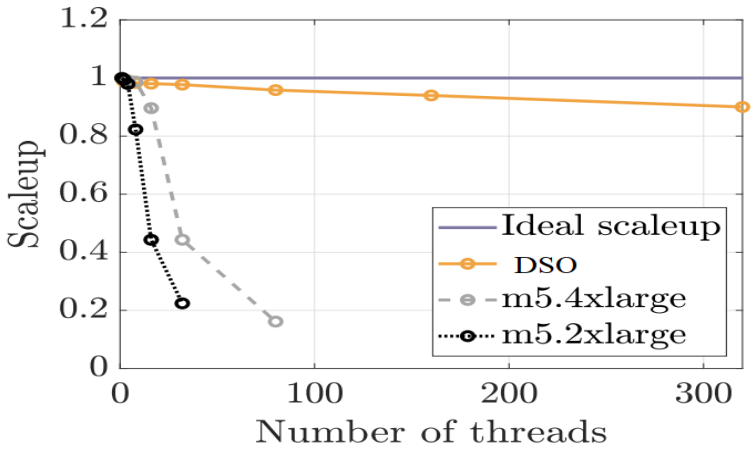

Figure 13: Scalability of architecture compared m5.2xlarge, m5.4xlarge and DSO layer with r5.2xlarge instance

6.3 Experiment 3

In this experiment, I have compared the k-means implementation on the proposed architecture against the AWS spark. K-means algorithm is an iterative algorithm that is a perfect fit to evaluate the state management of the proposed approach.

Setup: For the proposed approach - I have allocated 2048 megabytes of memory to run the k-means algorithm with 80 Lambda functions executing concurrently on one store node on r5.sxlarge elastic compute cloud instance. This configuration is chosen to have good performance at low cost.

For AWS Spark - Amazon elastic MapReduce cluster is configured with m5.2xlarge having 8 cores. The cluster is running with 1 master node and 10 worker nodes.

Dataset: I have used spark-pref *Databricks.* (2014) to generate data set of 30 GB having 55.6M elements.

The k-means has 100-dimensional points to correspond to each element. Further, the data set is split into 80 small partitions of equal size so that it can fit into the amount of memory available on each function. All the partitions are stored independently on Amazon S3.

k-means: In this experiment, I have compared the implementation of k-means as shown in 6.2 to MLlib of AWS spark. Initially, centroids points are at a random position. Figure 14 depicts the average time to complete 10 iterations, for k-means clustering algorithm. To evaluate the effectiveness of the proposed architecture I have taken different values of k. When the value of k is 25 the proposed architecture is 37% faster than the AWS spark. With the increase in the cluster, the completion time gap

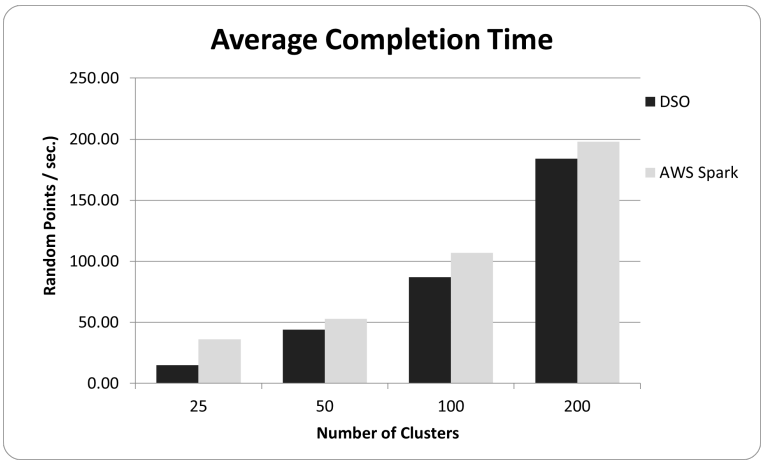

Figure 14: Average time to complete 10 iteration for k-means algorithm with different cluster size

is less as the synchronization at each iteration is decreased. Also, the iteration increasingly dominates the computation with less cost as the proposed approach does not require to reduce locally, reduce is performed at DSO layer remotely while in AWS spark have to perform reduce at each iteration locally.

Types	Proposed Architecture	AWS Spark	Proposed Architecture	AWS Spark
	k = 25		k = 200	
Total time (s)	88	170	239	337
Total Cost ($)	0.258	0.262	0.680	0.495
Iterations Cost ($)	0.058	0.053	0.510	0.300

Figure 15: Monetary cost for running the experiment

Figure 15 show the monetary cost for running the experiment with 10 iterations having the value of k 25 and 200. AWS spark takes comparatively more time to complete the iteration. When the value of k is 25 the total cost for execution is almost the same but when the k is 200, the cost is approximately 30% less in spark as compared to the proposed approach.

7 Conclusion and Future Work

In this paper, I have presented an efficient serverless architecture to support fine grained coordination and state sharing for distributed stateful applications. The proposed architecture uses an additional layer of distributed shared objects hosted on r5.2xlarge EC2 instance to provide intermediate state management and fine grained coordination between the cloud threads. Also, it provides replicas of the shared object using state machine replication. Any machine learning application that requires sharing its state and needs coordination between stages can be implemented on the proposed architecture by doing few modifications in the code. To evaluate this approach I have used micro benchmarks, latency,

throughput, and parallelism. The results show that the proposed DSO layer built on top of infinispan has the same access time as the Redis database with high throughput and is linearly scalable. Further, I have implemented a k-means machine-learning algorithm to check the efficiency of state management and also compared the average time of execution with AWS spark cluster using m5.2xlarge instance. The proposed approach finishes the execution in less time when the value of k is high. If the replication factor is enabled in the DSO layer then the total cost for executing the k-means is high in the proposed approach.

Due to time constraints, this research work is implemented only for the k-means machine learning algorithm. In the future, this research work can be further extended for logistic regression, linear regression, random forest, etc. Machine learning applications such as text processing, natural language processing, health prediction, and many more applications can also be deployed on serverless to get accurate and precise results with low latency.

References

Adhikari, M., Amgoth, T. & Srirama, S. N. (2019), 'A survey on scheduling strategies for workflows in cloud environment and emerging trends', *ACM Computing Surveys* **52**(4). CoreRank: A*, Impact Factor: 7.990.
URL: *https://doi.org/10.1145/3325097*

Alchieri, E., Dotti, F. & Pedone, F. (2018), Early scheduling in parallel state machine replication, *in* 'Proceedings of the ACM Symposium on Cloud Computing', SoCC '18, Association for Computing Machinery, New York, NY, USA, p. 82–94.
URL: *https://doi.org/10.1145/3267809.3267825*

AWS (2018), Maximum memory and runtime limit of aws lambda functions. url = https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/.

Carreira, J., Fonseca, P., Tumanov, A., Zhang, A. & Katz, R. (2018), A case for serverless machine learning, *in* 'Workshop on Systems for ML and Open Source Software at NeurIPS', Vol. 2018.

Carreira, J., Fonseca, P., Tumanov, A., Zhang, A. & Katz, R. (2019), Cirrus: A serverless framework for end-to-end ml workflows, *in* 'Proceedings of the ACM Symposium on Cloud Computing', SoCC '19, Association for Computing Machinery, New York, NY, USA, p. 13–24.
URL: *https://doi.org/10.1145/3357223.3362711*

Castro, P., Ishakian, V., Muthusamy, V. & Slominski, A. (2019), 'The rise of serverless computing', *Commun. ACM* **62**(12), 44–54. JCR Impact Factor: 5.86.
URL: *https://doi.org/10.1145/3368454*

CyclicBarrier class (2020). `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html`, Accessed: 2021-07-19.

Databricks. (2014). `https://github.com/databricks/spark-perf`, Accessed: 2021-06-20.

Distler, T., Bahn, C., Bessani, A., Fischer, F. & Junqueira, F. (2015), Extensible distributed coordination, *in* 'Proceedings of the Tenth European Conference on Computer Systems', EuroSys '15, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/2741948.2741954*

Dolev, S., Georgiou, C., Marcoullis, I. & Schiller, E. M. (2018), 'Practically-self-stabilizing virtual synchrony', *Journal of Computer and System Sciences* **96**, 50–73. CoreRank: A*, JCR Impact Factor: 1.494.
URL: *https://www.sciencedirect.com/science/article/pii/S002200001830432X*

Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G. & Winstein, K. (2017), Encoding, fast and slow: Low-latency video processing using thousands of tiny threads, *in* '14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)', USENIX Association, Boston, MA, pp. 363–376. CoreRank: National:USA.
URL: *https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi*

Ghosh, B. C., Addya, S. K., Somy, N. B., Nath, S. B., Chakraborty, S. & Ghosh, S. K. (2020), Caching techniques to improve latency in serverless architectures, *in* '2020 International Conference on COMmunication Systems NETworkS (COMSNETS)', Bengaluru, India, pp. 666–669.

Hasan, M. & Goraya, M. S. (2018), 'Fault tolerance in cloud computing environment: A systematic survey', *Computers in Industry* **99**, 156–172. CoreRank: B, JCR Impact Factor: 3.954.
URL: *https://www.sciencedirect.com/science/article/pii/S0166361517304438*

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I. & Recht, B. (2017), Occupy the cloud: Distributed computing for the 99%, *in* 'Proceedings of the 2017 Symposium on Cloud Computing', SoCC '17, Association for Computing Machinery, New York, NY, USA, p. 445–451.
URL: *https://doi.org/10.1145/3127479.3128601*

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R. A., Stoica, I. & Patterson, D. A. (2019), Cloud programming simplified: A berkeley view on serverless computing, Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.
URL: *http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html*

Joyner, S., MacCoss, M., Delimitrou, C. & Weatherspoon, H. (2020), 'Ripple: A practical declarative programming framework for serverless compute'. CoRR abs/2001.00222, 2020. https://arxiv.org/abs/2001.00222.

Kaffes, K., Yadwadkar, N. J. & Kozyrakis, C. (2019), Centralized core-granular scheduling for serverless functions, *in* 'Proceedings of the ACM Symposium on Cloud Computing', SoCC '19, Association for Computing Machinery, New York, NY, USA, p. 158–164.
URL: *https://doi.org/10.1145/3357223.3362709*

Kelly, D., Glavin, F. & Barrett, E. (2020), Serverless computing: Behind the scenes of major platforms, *in* '2020 IEEE 13th International Conference on Cloud Computing (CLOUD)', Beijing, China, pp. 304–312. CoreRank: B.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001), An overview of aspectj, *in* 'Proceedings of the 15th European Conference on Object-Oriented Programming', ECOOP '01, Springer-Verlag, Berlin, Heidelberg, p. 327–353.

Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J. & Kozyrakis, C. (2018), Pocket: Elastic ephemeral storage for serverless analytics, *in* 'Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation', OSDI'18, USENIX Association, USA, p. 427–444.

Lee, H., Satyam, K. & Fox, G. (2018), Evaluation of production serverless computing environments, *in* '2018 IEEE 11th International Conference on Cloud Computing (CLOUD)', San Francisco, CA, USA, pp. 442–450. CoreRank: B.

López, P. G., Arjona, A., Sampé, J., Slominski, A. & Villard, L. (2020), Triggerflow: Trigger-based orchestration of serverless workflows, *in* 'Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems', DEBS '20, Association for Computing Machinery, New York, NY, USA, p. 3–14.
URL: *https://doi.org/10.1145/3401025.3401731*

Marchioni, F. & Surtani, M. (2012), Infinispan data grid platform, *in* 'Packt Publishing Ltd'. url = https://www.packtpub.com/product/infinispan-data-grid-platform/9781849518222,Accessed: 2021-08-10.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I. & Stoica, I. (2018), Ray: A distributed framework for emerging AI applications, *in* '13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)', USENIX Association, Carlsbad, CA, pp. 561–577. CoreRank: A*.
URL: *https://www.usenix.org/conference/osdi18/presentation/moritz*

Pu, Q., Venkataraman, S. & Stoica, I. (2019), Shuffling, fast and slow: Scalable analytics on serverless infrastructure, *in* 'NSDI', Boston, MA, USA.

Sampé, J., Vernik, G., Sánchez-Artigas, M. & García-López, P. (2018), Serverless data analytics in the ibm cloud, *in* 'Proceedings of the 19th International Middleware Conference Industry', Middleware '18, Association for Computing Machinery, New York, NY, USA, p. 1–8.
URL: *https://doi.org/10.1145/3284028.3284029*

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I. & Patterson, D. A. (2021), 'What serverless computing is and should become: The next phase of cloud computing', *Commun. ACM* **64**(5), 76–84. JCR Impact Factor: 5.86.
URL: *https://doi.org/10.1145/3406011*

Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B. & Ragan-Kelley, J. (2018), 'numpywren: serverless linear algebra'. CoRR abs/1810.09679, 2018. http://arxiv.org/abs/1810.09679.

Sreekanti, V., Wu, C., Chhatrapati, S., Gonzalez, J. E., Hellerstein, J. M. & Faleiro, J. M. (2020), A fault-tolerance shim for serverless computing, *in* 'Proceedings of the Fifteenth European Conference on Computer Systems', EuroSys '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3342195.3387535*

Sutra, P., Riviere, E., Cotes, C., Artigas, M., Lopez, P., Bernard, E., Burns, W. & Zamarreno, G. (2017), Creson: Callable and replicated shared objects over nosql, *in* '2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)', IEEE Computer Society, Los Alamitos, CA, USA, pp. 115–128. CoreRank: B.
URL: *https://doi.ieeecomputersociety.org/10.1109/ICDCS.2017.239*

Tang, Y. & Yang, J. (2020), 'Lambdata: Optimizing serverless computing by making data intents explicit', *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* pp. 294–303. CoreRank: B.

Vahidinia, P., Farahani, B. & Aliee, F. S. (2020), Cold start in serverless computing: Current trends and mitigation strategies, *in* '2020 International Conference on Omni-layer Intelligent Systems (COINS)', Barcelona, Spain, pp. 1–7.

Xu, Z., Zhang, H., Geng, X., Wu, Q. & Ma, H. (2019), Adaptive function launching acceleration in serverless computing platforms, *in* '2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)', Tianjin, China, pp. 9–16. CoreRank: B.

Yavoruk, O. (2020), 'How does the monte carlo method work?'. CoRR abs/1909.13212, https://arxiv.org/abs/1909.13212.

Zhou, J., Chen, Y. & Wang, W. (2018), Atributed consistent hashing for heterogeneous storage systems, *in* 'Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques', PACT '18, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3243176.3243202*