Authors

Jordi Canosa Casellas

Pau Balanzà Malagelada

Directed by

Pedro García López

BACHELOR'S THESIS

Benchmarking and Comparison of Cluster vs Serverless Technologies

ENGINYERIA INFORMÀTICA

UNIVERSITAT ROVIRA i VIRGILI
Escola Tècnica
Superior d'Enginyeria

Tarragona, 2023

Abstract

This thesis embarks on a thorough comparative exploration of the two main execution models of cloud computing: Cluster and Serverless. The first one refers to a computing environment in which multiple interconnected computers or servers, often referred to as nodes, work together to perform tasks or provide services. On the other hand, in the Serverless model, it is the provider who is responsible for provisioning, managing, and scaling the servers. Users create and execute applications without interacting with the underlying infrastructure. To carry out the comparison, cluster technologies have been evaluated using Dask and Ray frameworks, while the serverless approach has been assessed using Lithops. The central goal is to unravel the distinct advantages and limitations inherent in each approach, providing valuable insights for their optimal selection based on varying scenarios. Through hands-on experimentation and benchmarking, this study diligently assesses the performance aspects of Dask, Ray, and Lithops. By closely analyzing factors such as resource efficiency, scalability, and deployment, the research aims to offer a deep comprehension of their operational behaviors. Venturing beyond traditional analysis boundaries, the study introduces a hybrid architecture, aimed at optimizing data ingestion. It presents an implementation that effectively combines both studied architectures. In essence, this study underscores the performance differences between cluster and serverless technologies, highlighting the unique capabilities of each, as well as proposing a hybrid architecture that suggests a promising unified approach.

Resum

Aquesta tesi s'embarca en una exploració comparativa dels dos grans models d'execució de computació en el núvol: Clúster i Serverless. El primer fa referència a un entorn de computació en el qual múltiples servidors interconnectats, sovint referits com a nodes, treballen de forma conjunta per executar tasques o proporcionar serveis. D'altra banda, en el model sense servidor és el proveïdor el que s'encarrega de subministrar, gestionar i escalar els servidors. Els usuaris creen i executen aplicacions sense entrar en contacte amb la infraestructura subjacent. Per dur a terme la comparativa s'han avaluat les tecnologies clúster mitjançant l'ús dels frameworks Dask i Ray, mentre que serverless s'ha avaluat amb l'ús de Lithops. L'objectiu central és descobrir els avantatges i les limitacions distintives inherents a cada enfocament, proporcionant idees valuoses per a la seva selecció òptima en funció de diversos escenaris. Mitjançant un procés de benchmarking per a l'experimentació pràctica i l'avaluació diligent, aquest estudi avalua minuciosament els aspectes de rendiment de Dask, Ray i Lithops. Analitzant de prop factors com l'eficiència dels recursos, l'escalabilitat i la implementació, la investigació té com a objectiu oferir una comprensió profunda dels seus comportaments operatius. Anant més enllà dels límits de l'anàlisi tradicional, l'estudi introdueix una arquitectura híbrida amb l'objectiu d'optimitzar la

ingestió de dades, presentant una implementació que combina de forma efectiva ambdues arquitectures estudiades. En essència, aquest estudi subratlla les diferències de rendiment entre les tecnologies de cluster i serverless, posant de manifest les capacitats úniques de cadascuna, a més de la proposta d'una arquitectura híbrida que suggereix un enfocament unificat prometedor.

Resumen

Esta tesis se embarca en una exploración comparativa de los dos grandes modelos de ejecución de computación en la nube: Clúster y Serverless. El primero se refiere a un entorno de computación en el cual múltiples servidores interconectados, a menudo referidos como nodos, trabajan de manera conjunta para ejecutar tareas o proporcionar servicios. Por otro lado, en el modelo Serverless es el proveedor el que se encarga de suministrar, gestionar y escalar los servidores. Los usuarios crean y ejecutan aplicaciones sin entrar en contacto con la infraestructura subyacente. Para llevar a cabo la comparativa se han evaluado las tecnologías de clúster mediante el uso de los frameworks Dask y Ray, mientras que el modelo serverless se ha evaluado con el uso de Lithops. El objetivo central es descubrir las ventajas y las limitaciones distintivas inherentes a cada enfoque, proporcionando ideas valiosas para su selección óptima según diversos escenarios. A través de un proceso de benchmarking para la experimentación práctica y la evaluación diligente, este estudio evalúa minuciosamente los aspectos de rendimiento de Dask, Ray y Lithops. Analizando de cerca factores como la eficiencia de los recursos, la escalabilidad y la implementación, la investigación tiene como objetivo ofrecer una comprensión profunda de sus comportamientos operativos. Yendo más allá de los límites del análisis tradicional, el estudio introduce una arquitectura híbrida con el propósito de optimizar la ingestión de datos, presentando una implementación que combina de manera efectiva ambas arquitecturas estudiadas. En esencia, esta tesis resalta las diferencias de rendimiento entre las tecnologías de clúster y serverless, poniendo de manifiesto las capacidades únicas de cada una, además de proponer una arquitectura híbrida' que sugiere un enfoque unificado prometedor.

Index

Figures Index

Code Index

Tables index

Plots Index

1. Introduction

In today's rapidly evolving technological landscape, the demand for efficient and scalable solutions has driven the rise of distributed systems. These systems, which involve multiple interconnected devices or computers working together, offer a powerful approach to tackling complex tasks and handling vast amounts of data. As our digital world becomes increasingly interconnected and data-driven, distributed systems have become essential tools for achieving high performance and reliability.

To harness the potential of distributed systems with data analysis, organizations often turn to Cluster technologies like Apache Hadoop [1], Spark [2], Ray [3] or Dask [4]. These frameworks provide the means to distribute workloads across multiple nodes, enabling parallel processing and faster execution of tasks. Ray, for instance, offers the ability to seamlessly distribute tasks, functions, and data across a cluster of machines, effectively harnessing the collective computational power for a wide range of applications. Dask, on the other hand, enables distributed computing using familiar programming paradigms like Pandas, making it easier to scale data processing tasks while maintaining code compatibility.

In parallel with Cluster technologies, Serverless computing has emerged as a paradigm that abstracts away the complexities of infrastructure management. Serverless platforms, like the one provided by Lithops [5], allow developers to focus solely on code and function logic, leaving the underlying infrastructure management to the platform itself. This approach proves especially beneficial for tasks that require quick and elastic scalability, as resources are allocated dynamically based on demand.

Integrating serverless technologies with distributed systems offers a unique combination of scalability, flexibility, and efficiency for data-intensive applications. As we explore the capabilities of these technologies, we delve deeper into the world of distributed systems, uncovering innovative ways to meet the challenges of our interconnected world.

1.1. Objectives and motivation of the study

The primary motivation behind this project stems from the limited research available on the comparison between cluster and serverless technologies. Nowadays, companies that utilize cluster technologies are uncertain about transitioning to serverless, as they are not completely sure whether Serverless can replace Cluster. In order to solve this problem, we based our thesis on three main goals:

1. Perform a comparative analysis between technologies to understand the strengths and limitations they have.
2. Understand when it is more appropriate to use each technology.
3. Determine if it is possible to combine the two technologies in a single implementation and in which situations would that be useful.

This thesis has been developed under the supervision of the CloudLab [6] research group at the Rovira i Virgili University (URV). We deeply want to express our gratitude to CloudLab for the provided resources, without which carrying out this project would have been impossible. These resources involve access to Amazon AWS and IBM clouds, as well as the group members who have been helpful to us at various instances in resolving the challenges we have encountered.

1.2. Methodology

This is primarily a benchmarking project, where the practical part has consisted in measuring the times taken by the three studied technologies (Ray, Dask and Lithops) to perform specific tasks. These tasks include data ingestion, analysing cluster startup times, and evaluating auto-scaling capabilities. The obtained times are presented in several plots to make it easier to reach conclusions.

The project also includes a development section where a simple hybrid model combining both cluster technologies with Lithops is implemented. To accomplish this task, a waterfall model has been followed, dividing the development into several sequential phases. Each phase is built upon the output and deliverables of the preceding one, ensuring a well-organized and structured development of the final prototype.

Working with these frameworks without any previous experience is a task that requires investing a significant amount of time and effort. It would not have been possible to achieve our objectives without an efficient organization and distribution of work.

Each of us has specialized in a cluster framework, which includes becoming familiar with its architecture and API. Pau has handled all tasks related to Ray, while Jordi has taken care of those involving Dask. Regarding Lithops, it has been approached in a collaborative way were both of us have involved ourselves in carrying out its tests.

2. State of the art

This section is focused on describing the advanced technologies key to our final bachelor's thesis. Within this, we explore the intricate relationship among cloud computing, cluster tools like Dask and Ray, and the streamlined serverless architecture of Lithops. Firstly, there is an introduction to cloud computing's transformative impact, enabling scalable and efficient research through abundant computational resources, and streamlined data processing. Going further, there is a description of the two cluster technologies used in the project, which enhance computational capabilities and empower parallel processing for complex tasks. Finally, there is a deeper exploration into Lithops, a serverless technology that simplifies infrastructure management, allowing us to focus on optimizing applications, describing its execution configuration, integration with other services, function structure, dependency packaging, storage, and communication.

2.1. Cloud Computing

Cloud computing refers to the provisioning of computing services, ranging from storage, processing power, databases, networking, software, and beyond, via the internet, often referred to as the Cloud. Instead of owning and managing physical hardware and software, users can access and use these resources on a pay-as-you-go basis from a remote provider.

In its entirety, cloud computing has brought about a paradigm shift in how enterprises and individuals access and utilize computational resources, presenting numerous advantages encompassing adaptability, efficacy, scalability, and innovation. It has become a fundamental technology underpinning the digital transformation of various industries.

Currently, there are several well-regarded cloud service providers, such as Amazon Web Services (AWS) [7], IBM Cloud [8], and Google Cloud (GCP) [9].

Figure 1. Cloud providers logos: AWS, IBM Cloud & Google Cloud

These providers offer an extensive array of services and tools tailored for the processing of vast volumes of data. Their offerings encompass a wide spectrum of capabilities, ranging from data storage and retrieval, robust data analysis, and real-time data streaming to advanced machine learning frameworks and scalable computing resources.

2.1.1. ***Virtual Machines***

One of the many services that cloud providers offer is the possibility of using Virtual Machines (VMs). These are software-based emulations of physical computers or servers. They allow users to run multiple independent operating systems and applications on a single physical machine, known as the host system. Each virtual machine operates as if it were a separate and dedicated computer (whether on-premises or not), with its own virtualized hardware components such as CPU, memory, storage, and network interfaces.

They offer isolation, resource sharing, and controlled environments for testing and development. VMs also support legacy software, enhance security through sandboxing, and aid in disaster recovery. They allow flexible resource allocation, have educational value, and are integral to cloud computing. In addition, VMs optimize server usage, reduce costs, isolate applications, and simplify operations.

AWS's cloud offers this service as AWS EC2 [10], IBM Cloud offers it as IBM Cloud Virtual Servers [11], and Google Cloud provides it through Compute Engine [12]. It is important to comprehend how this service operates as it has been used frequently in the thesis.

2.1.2. ***Object storage***

Another key service offered by cloud providers is Object Storage. It is a method for storing and retrieving data as discrete objects, each accompanied by metadata and a unique identifier. These objects are stored in a flat address space, eliminating the need for traditional file hierarchy. Objects stored in this type of storage are organized in logical containers called *buckets*. It is ideal for large amounts of data and unstructured data like images, videos, and documents.

The service offers scalability, durability, high availability, and accessibility through APIs. Cloud providers offer object storage solutions such as Amazon S3 [13] in AWS, IBM Cloud in IBM Cloud Object Storage [14], and Google Cloud Storage [15] in GCP. These services not only allow users to store and retrieve data efficiently, but also take advantage of additional features such as data encryption and replication across multiple geographic regions to ensure data durability and availability.

*2.1.3. **Function-as-a-Service (FaaS)***

FaaS is a type of cloud computing service that enables developers to design, execute, and manage sets of applications as functions without having to deal with the maintenance of their own infrastructure. Developers can write and deploy individual functions that are executed in response to specific events, making it easier to create event-driven applications and process real-time data.

It is an execution model based on events and runs in stateless containers. Functions handle the logic and state of servers by utilizing the services of a FaaS provider. An example of FaaS platform is AWS Lambda [16] or Google Cloud Functions [17]. AWS Lambda has been the platform used when working with Lithops, as it will be seen later on.

*2.1.4. **Kubernetes***

Kubernetes [18] (also known as K8s) is an open-source container orchestration platform used for automating the deployment, scaling, and management of applications in containers. It provides a framework for efficiently managing complex, distributed systems by automating tasks such as application deployment, load balancing, scaling, and resource allocation.

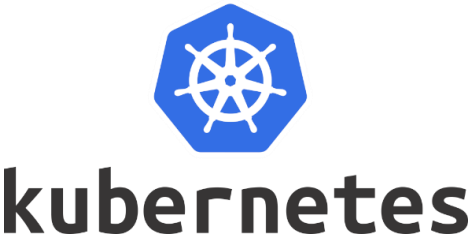


Figure 2. Kubernetes logo

Kubernetes achieves this through a combination of various components and mechanisms that collaborate to maintain the desired state of applications while adapting to changes and failures. Here is a high-level overview of how Kubernetes works:

1. **Master and Nodes Architecture**:
    a. *Master Node (Control Plane):* The control plane of Kubernetes, consisting of several key components, including the API server, etcd (persistence store), scheduler, and controller manager. The master node is responsible for receiving and processing commands, managing the cluster's state, and making global decisions about the cluster.
    b. *Worker Nodes:* These are the machines where containers are deployed and run. Each worker node runs a container runtime (e.g., Docker), as well as two critical Kubernetes components: the kubelet (manages containers on the node) and kube-proxy (handles network routing).

2. **Pods and Containers:** The basic deployment unit in Kubernetes is a pod, which can contain one or more containers. Containers within a pod share the same network namespace, IP address, and storage volumes. This simplifies communication between containers and allows them to work together seamlessly.

3. **Desired State and Declarative Configuration**: Users and administrators interact with Kubernetes through its API server, submitting declarative configuration files or commands specifying the desired state of the application or resources. Kubernetes continuously monitors the actual state of resources and compares it against the desired state. If there are any discrepancies, Kubernetes takes corrective actions to bring the actual state in line with the desired state.

4. **Deployments and Replication Controllers:** Deployments and ReplicaSets ensure the desired number of identical pods is running and can handle scaling and rolling updates. Deployments enable easy application updates and rollbacks without downtime.

5. **Services and Load Balancing**: Services provide a consistent IP address and DNS name for a group of pods, enabling load balancing and discovery. They allow

applications to communicate with each other within the cluster and expose services externally.

6. **Scaling:** Kubernetes offers Horizontal Pod Autoscaling (HPA), which adjusts the number of pod replicas based on resource utilization or custom metrics. This ensures optimal performance under varying loads.

7. **Self-Healing**: Kubernetes monitors the health of pods and nodes. If a pod fails or a node becomes unavailable, Kubernetes automatically replaces the failed components on healthy nodes, ensuring high availability and resilience.

8. **Networking:** Kubernetes provides a variety of networking solutions, including creating a virtual network for pods, managing IP addresses, and handling network routing.

9. **Persistent Storage:** Kubernetes manages persistent storage for applications through PersistentVolume (PV) and PersistentVolumeClaim (PVC) resources, allowing data to be preserved even when pods are rescheduled or replaced.

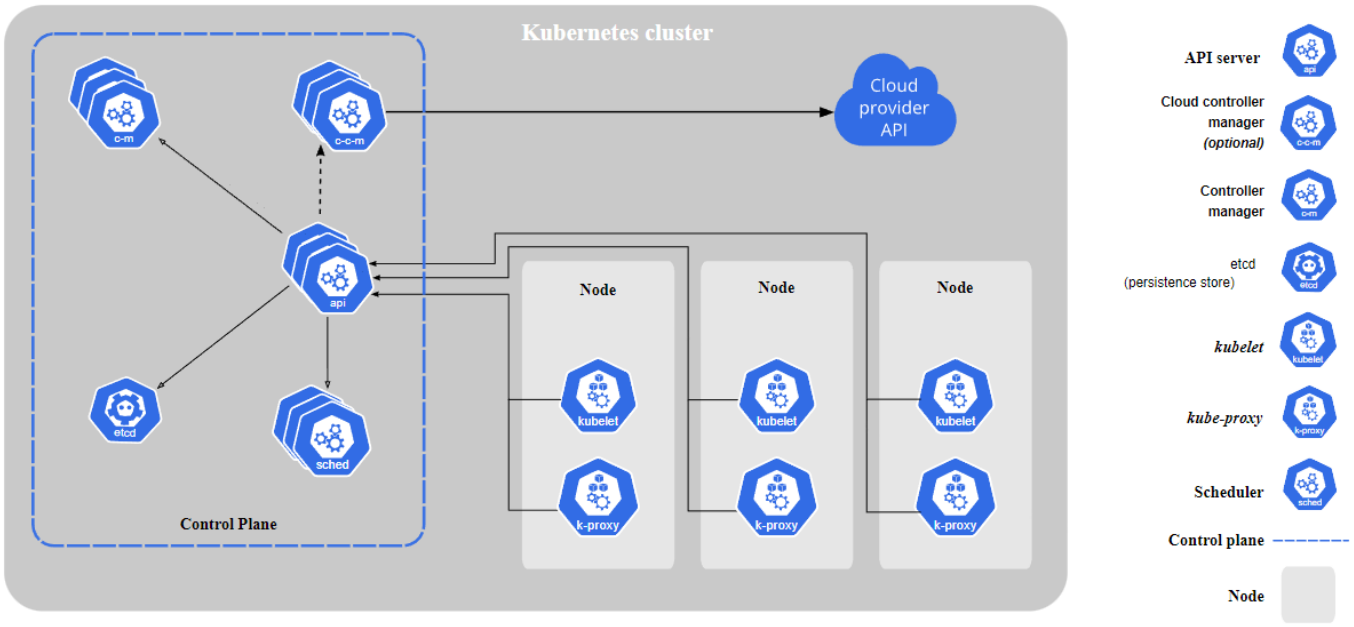


Figure 3. Kubernetes architecture [19]

Kubernetes abstracts the underlying infrastructure and allows developers to define how their applications should run, ensuring high availability, fault tolerance, and scalability. It simplifies the process of managing and deploying containerized applications across various environments, from local development to production clusters.

Many cloud providers provide support to Kubernetes: AWS with Amazon Elastic Kubernetes Service (Amazon EKS), GCP with Google Kubernetes Engine (GKE) and IBM Cloud with IBM Kubernetes Service (IKS). In the upcoming sections GKE and IKS will be implemented and used.

2.2. Dask

Dask is a Python library designed for parallel computing. It is made up of two main parts. The first part is all about dynamic task scheduling optimized for performing computations. It is built to work especially well for interactive computational tasks.

Figure 4. Dask logo

The second part is focused on "Big Data" collections. These are like parallel versions of arrays, dataframes, and lists from libraries like NumPy, Pandas, and Python iterators. These collections are designed to work in environments where data is larger than what can fit in memory, or in distributed setups. These collections operate on top of the dynamic task scheduler as shown in Figure 5.

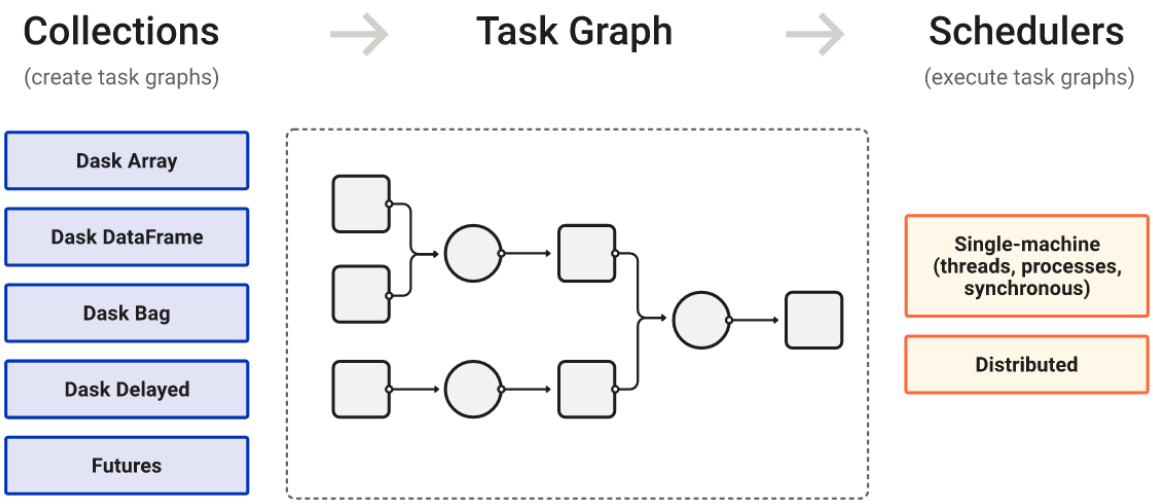


Figure 5. Dask High level architecture

Dask facilitates the manipulation of datasets that are larger than memory. It can also scale to clusters of many machines. It is designed to be resilient, meaning it can handle failures without crashing, and it is also elastic, so it can adapt to changes in the workload. It is particularly good for working with data that's on your local machine, which helps with performance. It is also responsive, which means it's quick to provide results and diagnostics. This makes it useful for interactive work.

*2.2.1. **Parallel Data Structures***

Dask's parallel data structures, such as *dask.array* and *dask.dataframe*, are designed to handle datasets that exceed the memory capacity of a single machine. The first one enables parallelized operations on large arrays, mimicking NumPy arrays by breaking them into smaller chunks that can be processed independently as described in Figure 6.

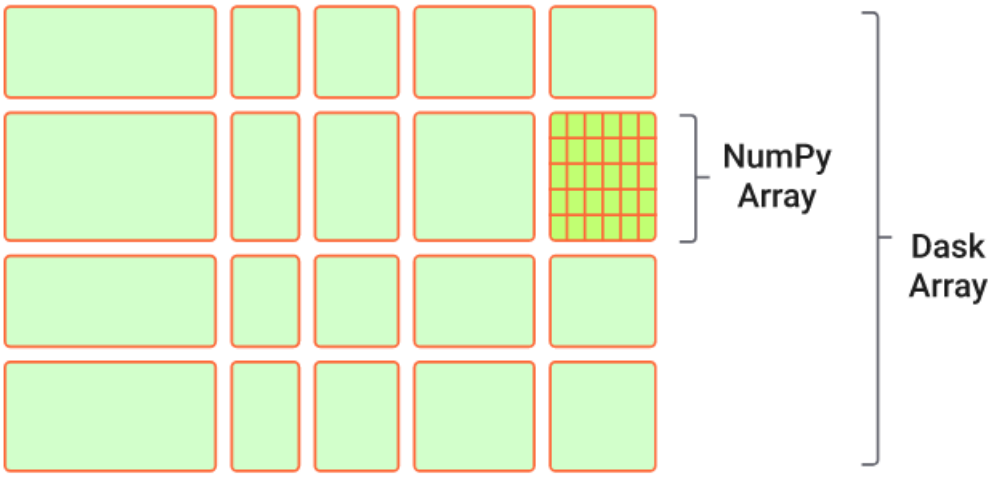


Figure 6. Dask Array vs NumPy Array

Similarly, *dask.dataframe* parallels the functionality of Pandas DataFrames, partitioning data into manageable pieces that can be processed in parallel. This partitioning allows for distributed computing across cores or machines, efficiently utilizing available resources. This library is used for the Data Ingestion experiment. This library has methods that allow the user to read data and transform it into a Dask Dataframe. The data can be stored in different formats as CSV or Parquet, and it can read it from local and from different cloud providers storage.

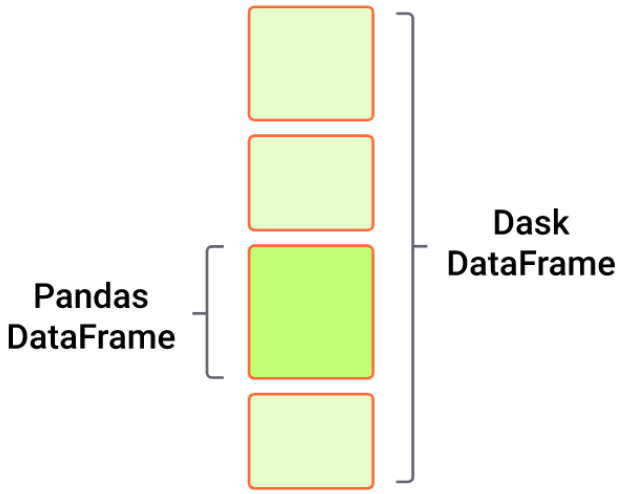


Figure 7. Dask Dataframe vs Pandas Dataframe

Moreover, Dask also has Dask Bag which implements operations like *map*, *filter*, *fold*, and *groupby* on collections of generic Python objects. It operates with Python iterators that work

in parallel with a small memory footprint. Dask bags basically coordinate many Python iterators or lists, each one being a partition of a larger collection.

Dask uses out-of-core computation ability, which is crucial when dealing with datasets that can't fit entirely in memory. It divides the data into smaller chunks that can be processed sequentially, loading only the required chunks into memory at a time. This mitigates memory limitations and minimizes the risk of memory errors, enabling efficient computation on large datasets.

In addition, Dask is flexible and allows you to define custom tasks beyond the provided data structures. This means you can parallelize your specific algorithms, whether they are numerical computations, machine learning models, or any other data processing tasks. This customization empowers you to harness the full potential of Dask's parallel computing capabilities in a tailored manner.

2.2.2. ***Lazy Evaluation***

Dask employs a lazy evaluation model, postponing actual computation until necessary. When operations are performed on Dask data structures, it constructs a directed acyclic graph (DAG) representing the computation steps.

The computation only happens when the result is explicitly requested by using the *compute()* function. This approach optimizes the execution by grouping tasks together, reducing memory overhead and allowing Dask to make better scheduling decisions.

In addition to its core data structures like *dask.array* and *dask.dataframe*, Dask provides the delayed function, which offers a more flexible way to parallelize and defer computations. The delayed function allows you to wrap a normal Python function call with a lazy version of that call. Instead of executing the function immediately, delayed constructs a task graph representing the computation steps required by the function. It can also be used as a decorator.

*2.2.3. **Task scheduling***

Dask's scheduler is responsible for mapping tasks in the DAG to available workers. It intelligently schedules tasks, considering dependencies and available resources. This enables parallel execution of tasks on multiple cores or machines. The scheduler can automatically balance the workload and manage task execution, ensuring optimal utilization of computing power while minimizing idle time.

Dask is primarily composed of two groups of schedulers: the Single-Machine Scheduler and the Distributed Scheduler.

The **Single-Machine Scheduler** offers fundamental functionalities within a local process or a thread pool. As the original and default scheduler, it presents a straightforward and cost-effective approach. However, its usage is limited to a single machine and does not inherently provide scalability.

The **Distributed Scheduler** offers more features. It is more sophisticated and, as a consequence, it requires more effort and knowledge to setup. Moreover, it can be employed both for local computation and across a distributed cluster, providing the potential for more extensive processing capabilities.

*2.2.4. **Custom task creation***

Dask's flexibility extends beyond its pre-defined data structures. With Dask, you have the remarkable capability to design and define custom tasks tailored precisely to your specific needs. This feature is not restricted to any particular domain, it covers a broad spectrum of applications, from complex numerical computations to sophisticated machine learning models, and virtually any type of data processing task.

Consider, for instance, a scenario where you are dealing with a unique data transformation sequence that doesn't fit neatly into Dask's existing frameworks. Instead of adapting your problem to fit within established structures, Dask empowers you to create a custom task, shaping your computation pipeline as you envision it. This adaptability is particularly invaluable in research, innovation, or any area where off-the-shelf solutions might fall short.

Whether you are distributing complex mathematical operations, training intricate machine learning models, or implementing your proprietary data manipulation algorithms, Dask stands ready to efficiently allocate resources and execute these tasks in parallel. This finely-tuned control over parallelization lets you maximize computational efficiency and optimize performance, all while preserving the integrity of your unique approach.

2.2.5. ***Scaling on Distributed Systems***

Dask's compatibility with distributed computing systems, including Hadoop, Kubernetes, and cloud-based clusters, facilitates scaling computations across multiple nodes. As the computational workload increases, Dask automatically expands the resources by adding more worker nodes to the cluster, enabling the processing of vast datasets on a larger scale.

In addition, Dask Distributed incorporates mechanisms for fault tolerance, a critical feature when working with distributed systems. If a worker node fails or encounters errors, Dask automatically detects and redistributes the affected tasks to healthy nodes. This ensures that the computation continues smoothly and reliably even in the presence of failures.

Dask is compatible with these cloud providers:

- AWS
- Digital Ocean
- Google Cloud Platform
- Microsoft Azure
- Hetzner

We will use the *EC2Cluster* [20] functionality for the VMs analysis and *KubeCluster* [21] for Kubernetes analysis. In Dask, clusters have an *adapt()* method that allows them to autoscale up and down just adjusting the *max_workers* and *min_workers*.

Dask EC2Cluster

The *EC2Cluster()* simplifies the process of establishing and supervising Dask clusters on Amazon Web Services' (AWS) Elastic Compute Cloud (EC2) infrastructure. It facilitates the creation of distributed computing clusters that can be conveniently adjusted in size to match

the specific computational requirements. This attribute proves especially advantageous for executing demanding computations, such as data processing, machine learning, and scientific simulations, within a cloud-based cluster environment.

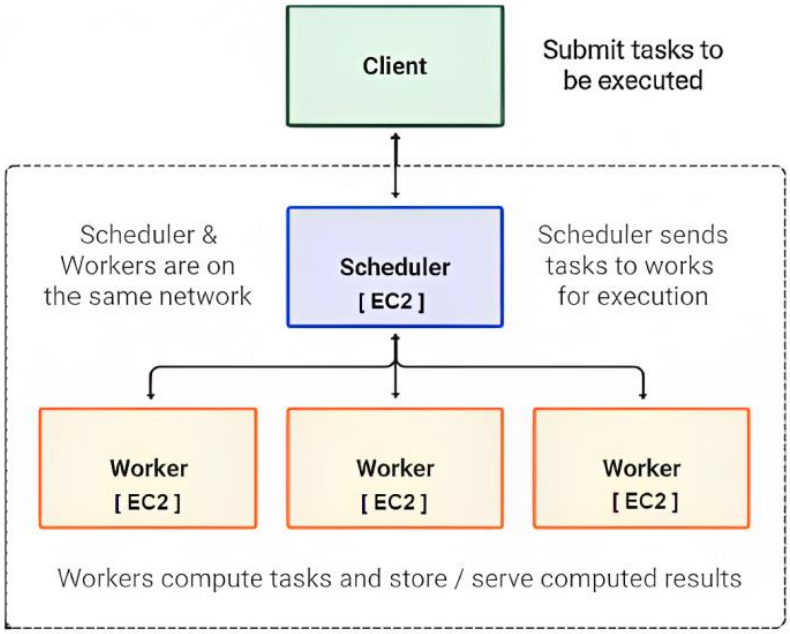


Figure 8. Dask EC2Cluster cluster

EC2Cluster() effectively shields users from the intricacies involved in configuring and overseeing the underlying infrastructure, offering a seamless avenue for users to fully exploit Dask's parallel computing capabilities on AWS EC2 instances.

Dask KubeCluster

The *KubeCluster()* functionality also simplifies the process of deploying and managing Dask clusters on Kubernetes infrastructure. Leveraging the power of Kubernetes, each worker and scheduler within the Dask cluster is represented as a pod, offering fine-grained control over

resource allocation. This approach allows users to establish distributed computing clusters that can be dynamically scaled up or down based on computational demands.

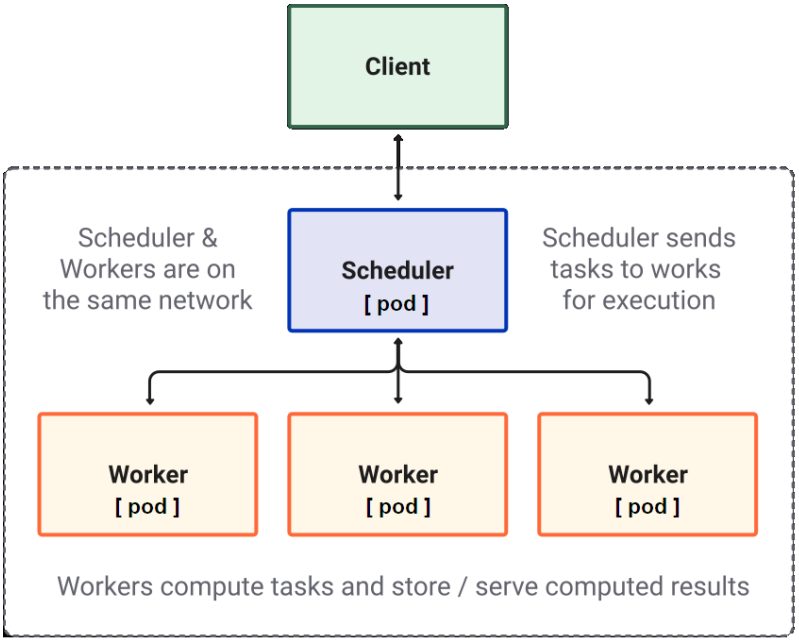


Figure 9. Dask KubeCluster cluster

It uses an Operator to enable the user to create and manage the Dask Kuberentes cluster in Python. This Operator is enabled via a Helm repository. The goal of this cluster manager is to abstract away the complexity of the Kubernetes resources and provide a clean and simple Python API to manager clusters while still getting all the benefits of the operator.

2.3. Ray

Ray is an open-source unified framework for scaling AI and Python applications like machine learning. It was developed by the company *Anyscale* and provides the compute layer for parallel processing so that users don't need complex distributed systems knowledge.

Figure 10. Ray logo

Ray aims to simplify the process of building and scaling applications that require parallel and distributed computation, making it easier for developers to utilize the resources of a cluster or a multicore machine. This is achieved by following several design principles:

1. **Simplicity and abstraction**: Regardless of whether we want to utilize all the cores of a computer or all nodes in a cluster, the Ray code remains nearly identical. Also as with any good distributed system, Ray manages task distribution and coordination under the hood.
2. **Flexibility and heterogeneity**: Ray's API is designed to make it easy to write flexible and composable code, and this is very useful for AI workloads, in particular when dealing with paradigms like reinforcement learning. It's also flexible in resource usage since Ray supports heterogeneous hardware (CPU and GPU).
3. **Speed and scalability**: Ray can handle millions of tasks per second with just milliseconds of latency. It's also efficient at distributing and scheduling tasks across the cluster, supporting autoscaling to support highly elastic workloads.

2.3.1. Three layers: Core, AI libraries and Clusters

Ray's unified compute framework consists of three layers:

1. **Ray Core**: An open-source, Python-based, versatile distributed computing library that enables machine learning engineers and Python developers to scale up Python applications and enhance the efficiency of ML workloads.
2. **Ray AI runtime**: A set of high-level libraries built and maintained by the creators of Ray. This includes the so-called Ray AIR to use these libraries with a unified API in common machine learning workloads. Each of Ray AIR's native libraries distributes a specific ML task:
    - *Data*: Built on the powerful Arrow framework, this library contains a data structure aptly called Dataset, an API for transforming such datasets, a way to build data processing pipelines with them, and many integrations with other data processing frameworks.
    - *Tune*: Scalable hyperparameter tuning to optimize model performance.
    - *RLib*: Scalable distributed reinforcement learning workloads that integrate with the other Ray AIR libraries.
    - *Train*: Distributed multi-node and multi-core model training with fault tolerance that integrates with popular training libraries.
    - *Serve*: Scalable and programmable serving to deploy models for online inference, with optional microbatching to improve performance.

3. **Ray Cluster**: A set of nodes connected to a Ray head node. Workloads utilizing the aforementioned libraries can be distributed across Ray clusters on a massive scale.

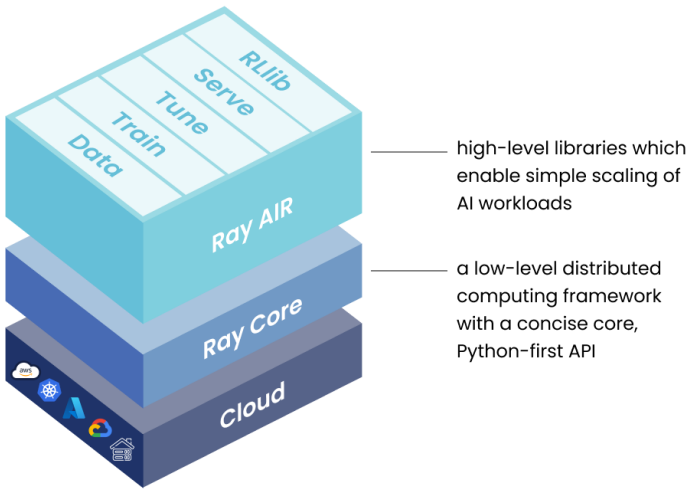


Figure 11. Stack of Ray libraries

In this project, none of the AI libraries have been used, except for Ray Data, which has been employed for carrying out data ingestion tests. For this reason, the following sections will solely discuss the functioning and architecture of Ray Core, Ray Data, and Ray Clusters.

2.3.2. ***Scaling Python applications***

Although it's a topic that generates some controversy, Python by itself proves to be rather inefficient for distributed computing tasks. The interpreter it employs operates primarily in a single-threaded way. This aspect creates challenges when attempting to leverage numerous CPUs within a single device, not to mention orchestrating an entire cluster of machines. Consequently, additional tools become a necessity, and fortunately the Python ecosystem offers several solutions to this problem. Libraries such as *multiprocessing* enable the distribution of tasks across multiple cores within a single machine, but not beyond.

API call	**Description**
ray.init()	Initializes a Ray cluster.
@ray.remote	Turns functions and classes into tasks and actors.
.remote()	Runs tasks remotely on the Ray Cluster.
ray.put()	Stores values into Ray's object store.
ray.get()	Gets values from object store.
ray.wait()	Returns a list with object references of the finished tasks and another list with the unfinished ones.

Table 1. Ray Core API

Ray allows its users to build and scale distributed applications on the cloud with just six API methods. In Table 1 there is a brief description of them. Throughout this section, it will be described how Ray converts Python functions and classes into remote entities and how it manages in-memory objects to ensure they are always accessible to all nodes.

Functions and remote Ray Tasks

Ray enables arbitrary functions to be executed asynchronously on separate Python workers. Such functions are called Ray remote functions and their asynchronous invocations are called Ray Tasks.

To illustrate the functioning of Ray Tasks, a practical case is presented in Table 2. This example is very useful to observe the few required changes to a Python function in order to execute it remotely using Ray.

Simple Python function	Remote Ray Task

`import time`
 	
`def increment (value):`
`  time.sleep(1)`
`  return value + 1`
 	
`st = time.time()`
 	
`results = [increment(value) for value in range(5)]`
 	
`print(f'Runtime: {time.time() - st:.2f} seconds')`
 	
`#Runtime: 5.05 seconds`
	`import time`
	`import ray`
	
	`@ray.remote`
	`def increment (value):`
	`  time.sleep(1)`
	`  return value + 1`
	
	`ray.init()`
	
	`st = time.time()`
	
	`futures = [increment.remote(value) for value in range(5)]`
	
	`results = ray.get(futures)`
	
	`print(f'Runtime: {time.time() - st:.2f} seconds')`
	
	`#Runtime: 1.09 seconds`

Table 2. Remote Ray Task

In this example, there is a very basic Python function that applies a one-second delay and returns the received parameter value incremented by one. This delay is used to simulate what could be a more complex computation in a real-case scenario. The function is executed a total of five times sequentially, which becomes highly inefficient specially if it is being executed on a machine with more than one core.

To transform the increment function into a Ray Task, you only need to add the *'@ray.remote'* decorator, which means the code within the function remains unchanged. Now, to invoke

this remote function, the *remote()* method must be used. This will immediately return an object reference (a future) and then create a task that will be executed on a worker process. The last step is to use the *get()* method to retrieve the results from the Ray Cluster.

As a test, this code has been executed on a 12-core machine, resulting in a time of 5.05 seconds for the sequential version, whereas the parallel version has executed the five calls in just 1.09 seconds. The overhead of distributing the tasks across the cluster can be observed in this last value, yet it becomes negligible when compared to the efficiency being gained.

Classes and remote Ray Actors

So far, it has been described how Ray enables the remote execution of stateless code, which is code that returns a result and doesn't require saving any state. But Ray also allows the execution of stateful computations within a cluster.

Ray Actors extend the Ray API from functions to classes. Actors are essentially stateful workers (or services) that can communicate between them. When a user instantiates a class that is a Ray Actor, Ray will start a remote instance of that class in the cluster. These actors can then execute remote method calls and maintain its own internal state.

In this project, this functionality of Ray Core hasn't been utilized, hence there will be no further emphasis on its operation.

Ray’s remote objects

Ray tasks and actors work with object refs, which essentially consist of a pointer or unique ID referencing a remote object without knowing its value. These objects are stored in Ray’s distributed shared-memory object store, enabling workers within a cluster to share them.

Every time a remote function is invoked, it creates a remote object and returns its reference, which can then be used to retrieve the function's result using the *get()* method. However, this isn't the only way to create an object. They can also be created with put() method, which creates a new object in Ray's memory with the variable received as a parameter, and it returns the corresponding object reference.

2.3.3. *Scaling data ingest and preprocessing*

Ray Data is a scalable data processing library specifically designed for machine learning workloads. This library provides a flexible API for performing tasks such as maps, grouped aggregations, or shuffling operations in a distributed manner.

Under the hood, this library implements distributed Apache Arrow [22], which is a unified columnar data format for data processing libraries and applications. This integration allows Ray Data to get interoperability with many of the most popular processing libraries, such as NumPy and Pandas.

Ray Data's main abstraction is a Dataset, which consists of a list of Ray object references pointing at a block of data, as it can be observed in Figure 12. These blocks are either Arrow tables or Python lists stored in Ray's shared memory, and each one contains a subset of rows. The compute operations over the data such as map or filter are essentially Ray tasks, so Ray Data inherits the key benefits of Ray core: scalability to hundreds of nodes, efficient memory usage, as well as object spilling and fail recovery.

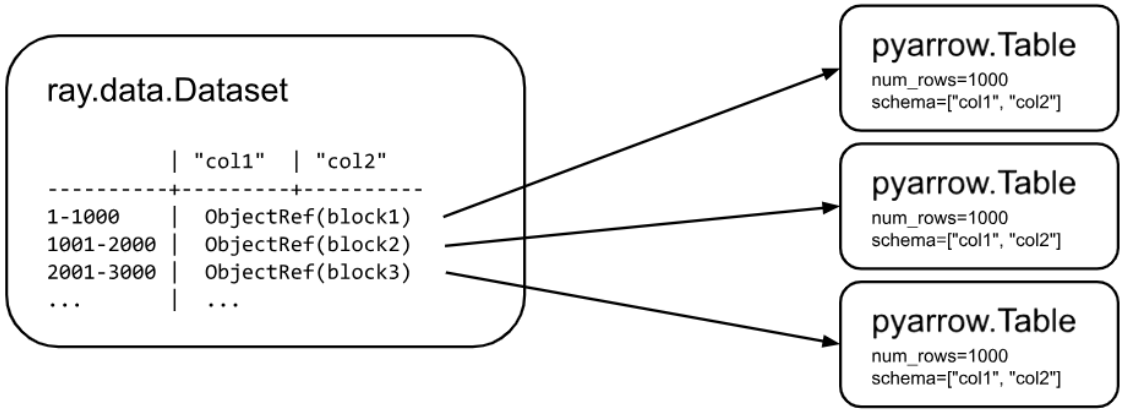


Figure 12. Ray Dataset structure

2.3.4. *Ray Clusters*

In the preceding sections, the tools provided by Ray for scaling Python functions and classes in a distributed manner have been examined. However, efficient libraries like Ray Core and Ray Data provide limited value if executed on a single machine, where parallelism is constrained by the number of cores and memory it possesses. Besides, Python already offers libraries like *multiprocessing* that enable parallel computation locally.

The last and arguably most significant layer of this framework comprises the Ray Clusters. They consist of a set of worker nodes connected to a common Ray head node, which also acts as a worker executing distributed tasks.

Ray Clusters can be fixed size or dynamically scale up and down, adjusting their resources to the workload they receive at any given moment. This is achieved through the Ray Autoscaler, a process that runs on the head node (or as a container in the head pod if using K8s). It adds worker nodes to the cluster when the demands of the workload exceed the current capacity and remove them when they are idle for a certain period of time.

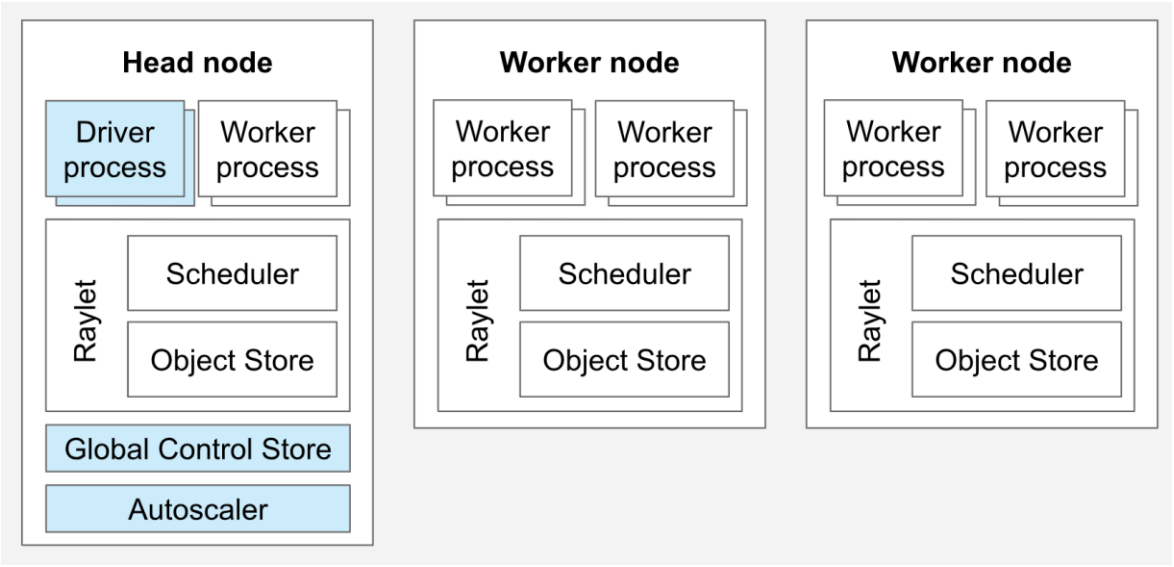


Figure 13. Ray Cluster architecture

In figure 13, the basic components of a Ray Cluster consisting of the Head node and two worker nodes can be observed. The blue-colored items of the diagram correspond to the process that are exclusively hosted by the Head node, while the rest of the processes are present on all nodes. Next, a brief description is provided for the purpose of each of these processes:

- **Worker process**: Responsible for task submission and execution. As it can be observed in the example, each worker node can contain multiple worker processes. In this project when we refer to a Ray Cluster with 128 workers, it means that the cluster have the necessary resources for running 128 worker processes simultaneously, regardless of the number of nodes.

- **Raylet**: A process present on all nodes that manages shared resources on each node. It consists of two components:
    - *Scheduler*: Responsible for resource management and task placement.
    - *Shared-memory Object Store*: Responsible for storing, transferring, and spilling large objects.
- **Driver**: It is a worker process that executes the top-level application but does not execute any tasks.
- **Global Control Service (GCS)**: It manages the Ray Cluster and serves as a centralized place to coordinate raylets and discover other cluster processes. It also serves as an entry point for external services like the autoscaler and the dashboard.

Ray Clusters can be deployed on virtual machines using one of the supported cloud providers (AWS, GCP, Azure or Aliyun), or on Kubernetes via the KubeRay [23] project. In Figure 14 we can observe a diagram representing the components of a KubeRay cluster. In contrast to using VMs where each EC2 instance was a Ray node, in this case, each node runs as a K8s pod.

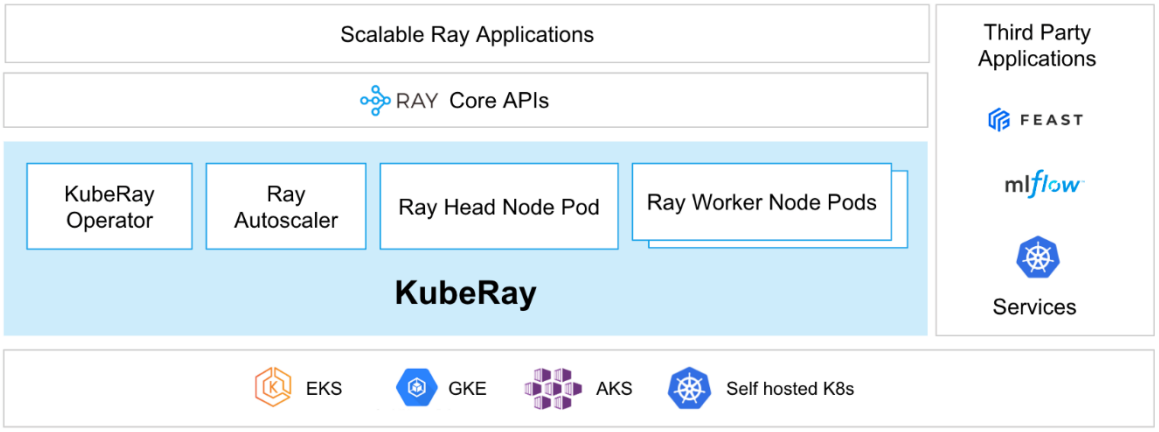


Figure 14. KubeRay architecture

In this project VMs clusters have been deployed on AWS whereas K8s clusters have been launched over IBM Cloud.

2.4. Lithops

Lithops is a serverless platform specifically designed for distributed data processing in cloud computing environments. It offers an abstraction layer over cloud services, enabling developers to execute code in parallel and distribute it without the need to manage the underlying infrastructure.

Figure 15. Lithops logo

Some of its key features include:

1. **Easy to use and learn**: It has a simple and user-friendly interface, as well as comprehensive documentation, which enables users with limited experience to effectively utilize Lithops.
2. **Cross-platform**: The same code can be executed through different cloud providers (AWS Lambda, Google Cloud Functions, IBM Cloud Functions…).
3. **Dynamic scaling**: The platform can easily adapt to compute demands, scaling up or down as needed. This ensures efficient resource utilization, eliminating issues such as over-provisioning or under-provisioning.
4. **Error friendly**: It is designed to manage failures in an elegant manner providing the user with detailed information about the encountered problems.

2.4.1. Execution configuration

Lithops is shipped with 3 different modes of execution, which allow the user to decide where and how the functions are executed:

- **Localhost Mode:** This mode enables execution of functions on the local machine through processes that use the available local CPUs. It offers a convenient and effective way to utilize Lithops' distributed computing capabilities without depending on cloud resources. This mode proves especially valuable for development, testing, and debugging objectives.
- **Serverless Mode:** This mode allows to run functions by using one or multiple FaaS serverless compute backends. Each function call represents a concurrent task operating in an isolated environment on the cloud, making it simple to parallelize task execution and take advantage of serverless environments' elasticity.
- **Standalone Mode:** This mode allows to run functions by using a remote host or a cluster of virtual machines (VM). This mode, which is similar to localhost mode, can be employed in a private cluster or in the cloud. Functions within each VM are executed making use of parallel processes.

```
import lithops

def square_function(x):
    return x * 2

iterdata = [1, 2, 3, 4]

fexec = lithops.ServerlessExecutor()

fexec.map(square_function, iterdata)

print(fexec.get_result())
```

Code 1. Lithops example

In addition, Lithops offers flexible configuration options to accommodate the specific requirements of each task or workflow. Users can define the desired number of workers or functions to be invoked, along with the allocated memory for each worker. Furthermore, Lithops provides mechanisms to manage task timeouts and set retry policies in case of failures.

In Code 1 there is a simple example of a map invocation using a Serverless executor. This call will launch as many functions as items in the input list that it receives, which in this case is a list of four integers. Each of the four functions will execute the code defined in the function called square_function and return its result.

It is important to note that the call to map is asynchronous, which means that it does not block the execution. Instead, it returns a list of futures that will eventually contain the results computed by the functions.

2.4.2. ***Runtime environment***

To ensure that functions have access to all the required libraries and modules during their execution, Lithops offers the ability to package Python packages and necessary dependencies. The way to define the runtime environment depends on the cloud provider being used as a backend. In the case of AWS, which has been the compute and storage backend used in this project, there are two ways to package the code and runtime dependencies:

- Predefined runtime: If the user does not require specific packages, they can choose to use a predefined runtime, which comes with preinstalled modules. To run a function with the default runtime nothing needs to be specified in the code, since everything is managed by Lithops.
- Container runtime: The most recommended option is to define a custom runtime using a Docker image, which will be used to initialize the containers where the functions are executed. In this case, the new runtime needs to be specified as a parameter when creating a Lithops Function Executor.

2.4.3. ***Storage and communication***

In addition to the Compute Backend on which functions are executed, Lithops also requires a Storage Backend to facilitate communication between the client and the workers.

When a client invokes a job, which is then passed to multiple workers for execution, the job data is stored in this medium. Furthermore, function results are also stored in this medium, thereby enabling communication between them. This is done by specifying a cloud storage bucket (such as an Amazon S3 bucket) through Lithops configuration. This bucket will contain all the intermediate data generated during task execution, enabling efficient and secure communication among the distributed tasks and preventing data loss.

3. Comparative Framework

In this section a comparative framework will be conducted to examine Cluster and Serverless architectures based on a set of defined criteria. The comparison is made with the information and knowledge we have before conducting any experiment. Once the Benchmarking experiments are completed, the results will be analyzed and compared to this initial comparative framework to validate or refine the initial observations and conclusions.

We have described Dask, Ray and Lithops frameworks individually. However, it is important to take it a step further and define the two main architectures they represent.

On the one hand, Cluster architecture refers to a computing environment in which multiple interconnected computers or servers, often referred to as nodes, work together to perform tasks or provide services. The goal of cluster architecture is to enhance performance, scalability, and reliability by distributing the workload across multiple machines.

On the other hand, Serverless architecture is a cloud computing approach where developers focus on writing code to implement specific functions or services without needing to manage the underlying infrastructure. In a serverless architecture, the cloud provider automatically manages the allocation and scaling of resources required to run the code.

The criteria used as the basis for comparison are defined in the following list:

- ***Deployment:*** The time required to deploy and configure the technology.
- ***Cold start issues:*** The delay the technologies can have when executing the application for the first time.
- ***Resource utilization:*** How each technology allocates and distributes resources.
- ***Scalability:*** How well can the technology handle varying workloads and scale resources as needed.
- ***Control over the infrastructure:*** The level of configuration capacity the developer possesses over each technology.
- ***Maintenance:*** The level of maintenance required for the technology.
- ***Cost-effectiveness:*** The economic costs associated with each architecture.

	Cluster	Serverless
Deployment	Nodes startup required before running applications	Negligible initialization time
Cold start issues	Less of an issue since nodes are typically always running and ready to handle requests	Occur when a new instance of the function needs to be created
Resource utilization	Under-provisioning and over-provisioning may appear	Allocates resources as needed
Scalability	Scales applications based on cluster resources	Scales functions automatically based on demand
Control over the infrastructure	Allows control for customizing servers, software, and configurations	Abstracts much of the infrastructure management, giving less control over low-level infrastructure details
Maintenance	Requires ongoing maintenance, updates, and troubleshooting	Minimal maintenance
Cost-effectiveness	Pay for the whole time the nodes are running, including startup	Pay only for the actual execution time of functions

Table 3. Framework comparison Cluster vs Serverless

Based on the information gathered so far regarding cluster and serverless technologies, Table 3 provides an overview of the behaviour of each according to the established criteria. Below, we dig into each of these points for both architectures.

Cluster

Whenever working with a cluster technology, it is essential to consider the cluster's startup time. During this period, the nodes are initialized by installing the needed libraries, and all the necessary connections are established among them to form the cluster.

In this technology there should not be any cold start issue since nodes are already running when the user submits work to them. However, some issues regarding resource utilization may appear. Firstly, if the cluster resources are insufficient to meet some peak demands, there is a situation of under-provisioning which can lead to performance degradation and increased response times. On the contrary, if the cluster has more allocated resources than required there is an over-provisioning situation, leading to inefficient resource utilization and higher costs.

Regarding scalability, most cluster technologies can horizontally autoscale adding or removing nodes adapting to the workload they are running. However, again it is important to consider the startup times of the new nodes that are added to the cluster.

Another strength of cluster architecture is that it allows a high customization degree over the resources, since the user knows exactly in which machines will be the application executed. Nevertheless, clusters require some maintenance if they are running for long periods of time, and the user must pay for every minute the machines are up.

Serverless

As for serverless technologies, deployment involves packaging and uploading individual functions or services to the cloud provider. Developers do not have to manage deployment at the server level. Moreover, the resources are provisioned based on the requirements in order to have an effective resource utilization.

Serverless functions can experience a delay in execution known as "cold start" when a new instance of the function must be generated and executed for the first time. This can impact

response time for certain types of applications. Relating to scalability, serverless platforms automatically scale functions in response to demand. Functions are often executed in parallel and can handle varying workloads effectively. In addition, serverless architectures abstract much of the infrastructure management, giving developers less control over low-level infrastructure details.

In reference to the maintenance and costs, it is only required the slightest maintenance, as infrastructure is managed by the provider. Furthermore, developers only pay for the actual computing resources consumed during the execution of their code, making it cost-effective and efficient.

Serverless architectures are well-suited for event-driven applications, microservices, and tasks that require rapid scaling or have variable workloads.

4. Benchmarking

In this section are described the different experiments that were conducted in order to compare and analyze the Serverless and the Cluster architectures. Most of the tests are focused on the cluster architecture and its constraints. Therefore, there is also a subsection containing the analysis and conclusions from the study.

For each test, a preliminary study has been carried out to choose the most suitable instances in each scenario, trying to minimize resource wasting and economic costs. The selected instances for each experiment are detailed in the specification subsection, along with how many of them have been used to build the corresponding clusters.

Here are the specifications of every different VM instance type used in the upcoming experiments (the costs are on-demand):

	CPUs	Memory (GB)	Bandwidth (Gbps)	Platform	Cost (hourly)
m6i.large	2	8	Up to 12.5	AWS	$0.096
m6i.4xlarge	16	64	Up to 12.5	AWS	$0.768
m6i.8xlarge	32	128	12.5	AWS	$1.536
m6i.16xlarge	64	256	25	AWS	$3.072
e2-standard-8	8	32	16	GCP	$0.268
cx2.8x16	8	16	16	IBM	$0.410

Table 4. VMs costs

As for the cost of the AWS Lambda functions used with Lithops for the study, it is illustrated in the table of Figure 16. This table outlines the AWS Lambda pricing based on the specific memory requirements.

Memory (MB)	Price per 1ms
128	$0.0000000021
512	$0.0000000083
1024	$0.0000000167
1536	$0.0000000250
2048	$0.0000000333
3072	$0.0000000500
4096	$0.0000000667
5120	$0.0000000833
6144	$0.0000001000
7168	$0.0000001167
8192	$0.0000001333
9216	$0.0000001500
10240	$0.0000001667

Figure 16. AWS Lambda costs according to its memory [23]

In all the experiments we set the memory of the Lambda functions to 4GB in order to make it equivalent to the memory that has a worker with a VM. So, the concrete cost is $0.0000000667 per every millisecond each function is used.

4.1. Cluster Loading

4.1.1. ***Description***

One of the main differences mentioned above between Cluster architecture and Serverless architecture is that in the first one, VMs must initialize and that implies a considerable amount of time.

The goal of this experiment is to observe the startup time of a Ray and Dask cluster, both on VMs and Kubernetes, as it is a time that is important to consider when making any comparisons, since it is by no means negligible. Additionally, the test also aims to contrast whether the creation and initialization time of the cluster increases as we increase the size in terms of nodes.

Firstly, we will measure the startup time of 3 different clusters by increasing the number of CPUs starting with 8 and reaching 128, with an intermediary value of 32. Our hypothesis is

that as we horizontally scale the cluster, the time will increase, since the more nodes, the more connections between them will need to be established.

Secondly, to ensure that the total startup time is directly proportional to the number of connections that need to be established within the cluster, we have conducted a second test in which the total number of CPUs is kept constant and what varies is the number and size of nodes. In other words, the cluster scales vertically.

We believe that as the instances configuring the cluster grow larger (while keeping the same number of CPUs), it will require fewer connections since there will be fewer nodes, thus leading to shorter processing times. To validate this part of the theory, we employed VMs.

4.1.2. ***Setup and Specifications***

Test 1

	VMs	**K8s**
Cloud provider	AWS	IBM, GCP
Cloud services	EC2	IKS, GKE
Instance family	m6i	cx2, e2-standard
Region	us-east-1	us-south-2, us-central1
Ray version	2.5.0	2.5.0
Dask version	2023.4.0	2023.4.0

Table 5. General specifications for Cluster Loading - Test 1

VMs Cluster configurations:

	Nodes	**Instance type**	**CPUs x node**	**Total CPUs / Workers**
Cluster 1	4	m6i.large	2	8
Cluster 2	16	m6i.large	2	32
Cluster 3	25	m6i.large	2	50

Table 6. Cluster configurations with VMs for Test 1

K8s Cluster configurations:

	Nodes	Instance type	CPUs x node	Total CPUs	Total pods / Workers
Cluster 1	7	e2-standard-8 cx2.8x16	8	56	8
Cluster 2	7	e2-standard-8 cx2.8x16	8	56	32
Cluster 3	7	e2-standard-8 cx2.8x16	8	56	50

Table 7. Cluster configurations with K8s for Test 1

Test 2

	VMs
Cloud provider	AWS
Cloud services	EC2
Instance family	m6i
Region	us-east-1
Ray version	2.5.0
Dask version	2023.4.0

Table 8. General specifications for Cluster Loading - Test 2

VMs Cluster configurations:

	Nodes	Instance type	CPUs x node	Total CPUs / Workers
Cluster 1	64	m6i.large	2	128
Cluster 2	8	m6i.4xlarge	16	128
Cluster 3	4	m6i.8xlarge	32	128
Cluster 4	2	m6i.16xlarge	64	128

Table 9. Cluster configurations for Test 2 of Cluster Loading

4.1.3. ***Procedure***

The procedure of this experiment involves deploying the various clusters described in the previous section and measuring the time it takes for all nodes to become available. In other words, evaluate the time span from issuing the command to create the cluster until all workers are operational.

In the case of Ray on VMs, all the configuration is defined in a YAML file, along with other options such as the size of EBS volumes associated with each instance. It's worth noting that the instances do not use any Docker image, but rather are manually configured through a set of commands specified in the configuration file. This is because the Docker images provided by the Ray repository are quite heavy and include packages that are not required for conducting this study. To deploy the cluster, the command *ray up -y config.yaml* must be executed.

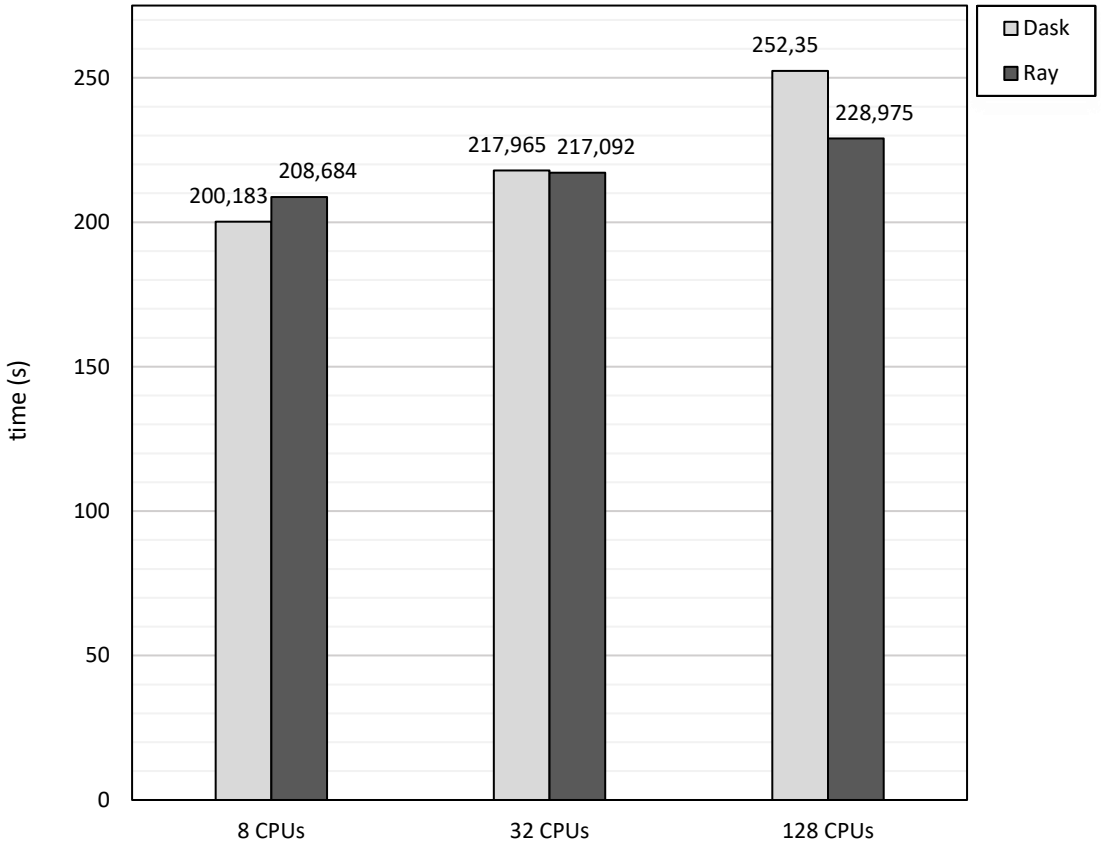
Regarding K8s, Ray clusters have been created on a Kubernetes cluster previously established on the IBM IKS service. The initial step involves deploying the KubeRay Operator, which streamlines the deployment and management of Ray applications on Kubernetes. This operator is deployed on the cluster using a Helm chart repository. Once the KubeRay Operator is up and running, the cluster is launched through a deployment specified in a YAML file, in which the resources assigned to the pods that will become cluster workers are configured.

To create the VM cluster in Dask, the *EC2Cluster()* functionality of Dask has been used. AWS credentials are passed to it as an environment variable to ensure all nodes have access. In this method, we essentially provide the configuration and specify that we want the nodes to use the Docker image *"daskdev/dask:2023.4.0-py3.10"*. Dask offers this image in a repository, which essentially includes all Dask packages with version 2023.4.0 and Python 3.10 to match the client version.

Finally, to create the Kubernetes cluster in Dask, the *KubeCluster()* functionality has been employed on a cluster previously established on Google Kubernetes Engine (GKE) service of GCP. The latest version of KubeCluster utilizes an Operator to facilitate the deployment of applications in a manner similar to Ray. This Operator is deployed on the cluster using a Helm chart repository. Then, we provide the configuration as parameters to the

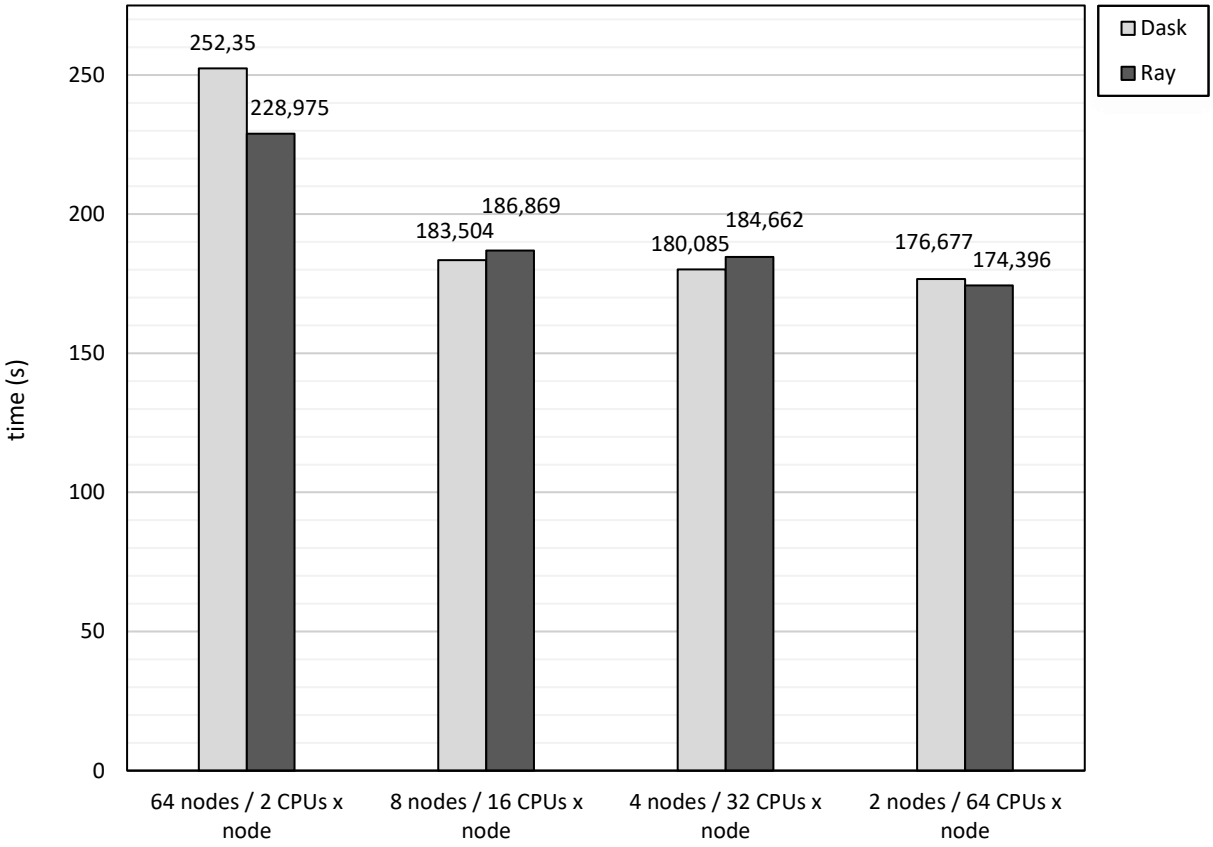
KubeCluster() method, specifying the same image as with VMs and indicating that each worker should have 1 CPU.

4.1.4. ***Results obtained and analysis***



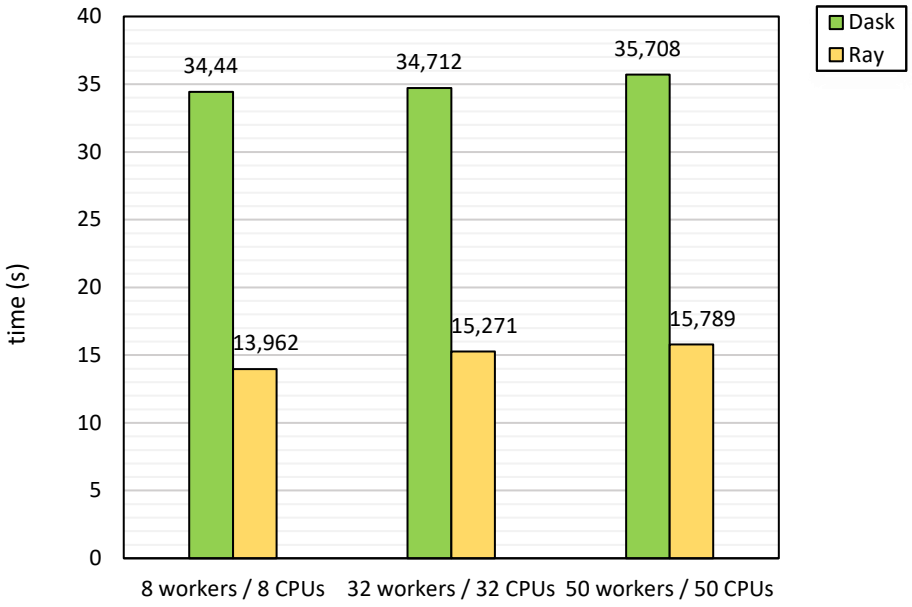
Plot 1. Dask vs Ray cluster startup times on VMs varying cluster size

As we can observe from Plot 1, startup times increase as the cluster size grows. Ray delivers better results for large clusters, whereas Dask performs better in smaller ones.



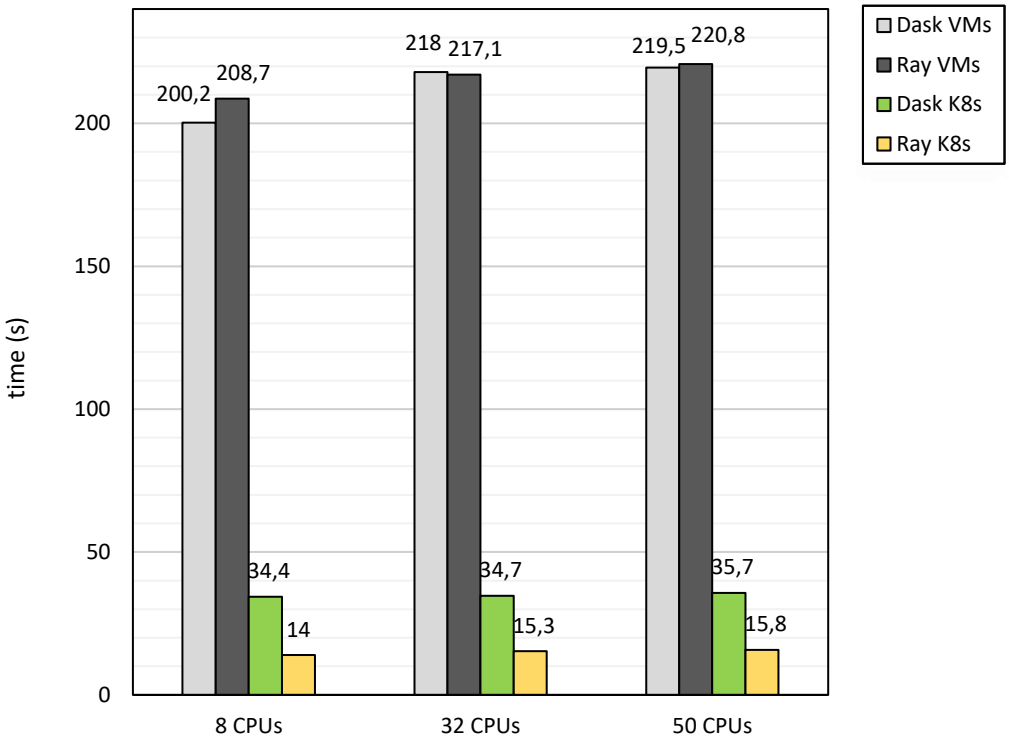
Plot 2. Dask vs Ray cluster startup times on VMs varying node size

As expected, the time required for achieving operational status in a cluster increases proportionally with the number of VMs or nodes present. This correlation arises due to the increasing number of connections that need to be established within the cluster as the nodes or VMs grow in quantity. Both Ray and Dask achieve very similar startup times.



Plot 3. Dask vs Ray cluster deploying times on K8s

Regarding the comparison between the two frameworks, we observe that while the times are quite similar in VMs, in K8s, Ray achieves significantly better results than Dask.



Plot 4. Dask vs Ray cluster deploying comparison on VMs & K8s

If we take a look at the obtained results in Plot 4, it is evident that both in Ray and Dask, are much faster to deploy a cluster on K8s than on VMs. This significant difference is attributed to the fact that in the case of Kubernetes, we are measuring the time it takes to create pods on a cluster of pre-initialized and configured nodes. The creation time of the Kubernetes cluster, on which we create our Ray and Dask clusters, may vary between 5 and 20 minutes. Taking this into consideration, we can conclude that if we disregard this initialization time of the K8s cluster, it is much more efficient to use this technology due to the very low pod initialization times compared to the time required to initialize the VMs.

The main conclusion that can be drawn from this experiment is that no matter how much we optimize the startup times of the clusters, they will never be negligible, thus remaining a significant limitation towards the serverless model. This limitation becomes more crucial for shorter tasks, as for tasks with a duration less than 4-5 minutes in the case of VMs, or 30 seconds on K8s (assuming the cluster deployment has been done beforehand), the time taken to create the cluster will be comparable to the time taken to execute the actual task.

4.2. Data Ingestion

4.2.1. ***Description***

The first phase in most machine learning pipelines consists in data extraction and preprocessing, which involves extracting a dataset from a source such as object storage and transforming it so that it can be processed by the ML model. This data can come in various formats, whether structured (CSV, JSON, Parquet) or unstructured

In this experiment, we aim to address the question of which technology is best suited for performing the ingestion of a dataset stored in cloud object storage. To accomplish this, we will conduct a comparison of the reading times provided by Ray, Dask, and Lithops for different data sizes.

As can be observed in the specifications section, in the case of the cluster technologies we have worked with a cluster comprising 32 CPUs and another with 128 CPUs. Regarding Lithops, 32 and 128 single-core functions have been deployed, aiming to achieve the most equitable resource comparison possible.

The experiment has been conducted using EC2 m6i.large instances, which come equipped with 2 CPUs, 8 GB of RAM, and a bandwidth of 1.6 GBps. As there will be 2 workers operating concurrently within each virtual machine, it can be noted that each worker will have 1 CPU, 4GB of memory, and a bandwidth of 0.8 GBps. For the Lithops functions, a runtime memory of 4 GB has been allocated to ensure they operate under the same conditions.

The data to be ingested is stored in CSV format across multiple files within an S3 bucket located in the same zone as the VMs. After conducting several tests, the decision was made to partition the data into as many files as there are CPUs in the cluster designated for ingestion. This approach enables each worker to process a file in parallel, ensuring that no worker remains idle. It is evident that in a real-world scenario, the total number of files might differ, but our goal is to measure the reading speed of the three technologies under the best-case scenario.

Finally, it's important to note that ingestion times decrease with each execution. That means that the initial reading of data is much slower than the subsequent ones. For this reason, this study has separated the times corresponding to the first execution (cold) from the times obtained after several executions (warm).

4.2.2. ***Setup and Specifications***

	Ray & Dask	**Lithops**
Cloud provider	AWS	AWS
Cloud services	EC2, S3	AWS Lambda, S3
Instance family	m6i	/
Region	us-east-1	us-east-1
Version	2.5.0 2023.4.0	2.9.0

Table 10. General specifications for Data Ingestion Test

VMs Cluster configurations:

	Nodes	Instance type	CPUs x node	Total CPUs / Workers
Cluster 1	16	m6i.large	2	32
Cluster 2	64	m6i.large	2	128

Table 11. Cluster configurations for data ingestion

Data sizes for 32-CPU clusters:

Size (GB)	Files	Chunk size (MB)
5	32	160
10	32	320
20	32	640
50	32	1600

Table 12. Data sizes for 32-CPU clusters in Data Ingestion

Data sizes for 128-CPU clusters:

Size (GB)	Files	Chunk size (MB)
5	128	40
10	128	80
20	128	160
50	128	400

Table 13. Data sizes for 128-CPU clusters in Data Ingestion

4.2.3. ***Procedure***

As said in the description section, we will approach this experiment with 3 different technologies: Ray, Dask and Lithops.

Ray approach

To carry out data ingestion in Ray, the Ray Data library is utilized, which lets us convert data from CSV files stored in an S3 bucket into a Ray Dataset.

To achieve this, the *read_csv()* function is employed, which generates an Arrow dataset from a set of CSV files residing in a bucket named *data_to_ingest*. This function will initiate as many Ray tasks as files within the directory provided. Each task is responsible for reading

one file and producing an output block. As a result, the resulting dataset will comprise as many blocks as the number of files read, which in our case will be 32 and 128.

It is important to mention that unlike technologies such as Dask and Lithops, Ray does not provide the option to define a chunk size when reading the data. This presents a significant limitation, as if all the data were in a single file, parallel reading would not be feasible. Instead, a single worker would have to handle the entire ingestion. The only way of processing the dataset with all workers reading a portion of it, is having as many files as there are workers in the clusters, as we have done in this experiment.

```
ray.init(address='auto')
st = time.time()´
df = ray.data.read_csv("s3://data_to_ingest/5GB_128/")
df.materialize()
et = time.time()
```

Code 2. Ray Python code for Data Ingestion with 128 workers

When measuring the execution time of the *ray.data.read_csv()* function, it might seem confusing that it executes almost instantly, regardless of the data volume. This is due to the inherent lazy execution nature of Ray Data functions. In this case, the complete data reading process does not occur until an operation that transforms or consumes the dataset is executed. To measure the cluster's ingestion time for all the data, the *materialize()* function is invoked, which effectively reads all data blocks into memory.

Dask approach

To conduct this experiment in Dask we used the dask.dataframe library. The *read_csv* method allows Dask to read the CSV files from the S3 bucket and transform them into a Dask DataFrame.

Dask.dataframe's methods are lazy by necessitating the use of the *compute()* function to obtain definitive results. However, it is not recommended to apply the *compute()* directly upon the *read_csv* method. To force the cluster to initiate the DataFrame reading process, we applied a *head()* function. This operation effectively retrieves data from each partition, ensuring comprehensive data ingestion as exemplified in the provided code snippet below.

```
st = time.time()

df = dd.read_csv("s3://data_to_ingest/5GB_128/*.csv", assume_missing = True,
blocksize="160MB")

df.head(n=1000, npartitions=128).compute()
et = time.time()
elapsed = et-st
```

Code 3. Dask Python code for Data Ingestion with 128 workers

Nevertheless, in the first version we measured the times as shown in the code, but it was incongruous to measure the time that it takes to compute the head part. So, in order to make it in a more accurate manner we accessed the task stream and got the exact amount of time that required the read part only as shown in Figure 17.

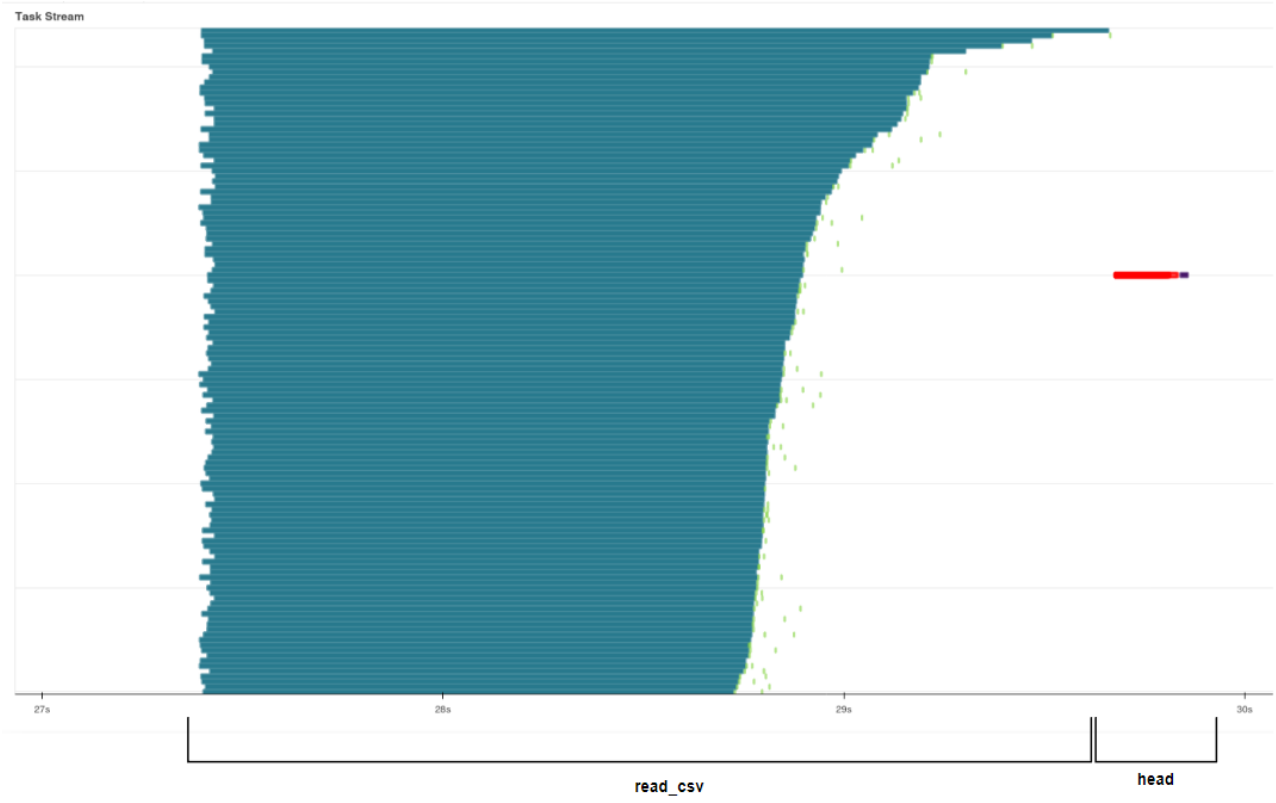


Figure 17. Task Stream of Dask in Data Ingestion

Lithops approach

Looking at the official Lithops documentation, we can see that in the Data Processing section, there's a demonstration of processing files stored in object storage using the *'obj'* parameter. This is incredibly useful, as you can invoke the *map()* function and provide it with, for instance, the URL of an S3 bucket along with an appropriate chunk size. This approach ensures that each lambda function receives a portion of the data from the bucket, sized as defined in the *'obj_chunk_size'* parameter.

However, in this experiment, since the data is already partitioned into the desired number of files, there's no need to define a chunk size. Instead, the URL of the file that each function needs to process can be directly passed to them.

It's important to consider that both Ray and Dask not only read the data within CSV files but also create a dataframe from this data. This allows the execution of filter or transformation operations to the data once it's been read, allowing for fast and efficient processing. For this reason, we've chosen to integrate the Pandas library into the Lithops implementation. Each function will establish a connection to S3 and, using the *read_csv* function provided by the library, create a dataframe.

As it can be observed in the code, the functions don't return the dataframe they create; rather, they return the time they've consumed to build it. This approach is adopted because Lithops functions aren't designed to return such large objects, and our focus lies in capturing the time spent on the reading process. Consequently, each function returns its specific time, and once all functions have completed, the maximum value among these times is extracted. This maximum time is the value showcased in the plots.

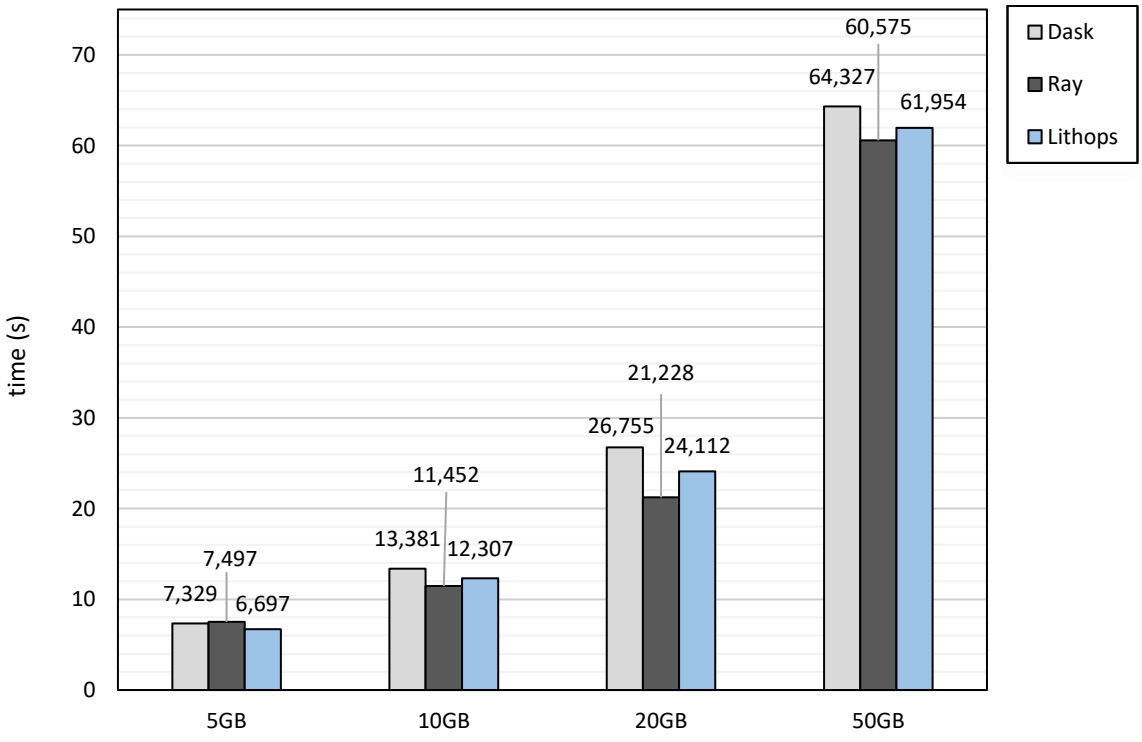
```
bucket_name = 'data_to_ingest’

def read_csv(s3_uri):
    st = time.time()
    df = pd.read_csv(s3_uri)
    elapsed = time.time() - st
    return elapsed

#Initialize Lithops ServerlessExecutor
fexec = lithops.ServerlessExecutor(runtime='lithops-runtime',
runtime_memory="4096")

#Get the list of CSV files in the bucket
storage = lithops.Storage()
csv_files = storage.list_objects(bucket_name, "5GB_128/")
```

```
#Prepare the inputs for the map operation
inputs = [(bucket_name, file['Key']) for file in csv_files]
uris = []
for key in inputs:
    uris.append(f's3://{bucket_name}/{key[1]}')

#Invoke the map operation to read CSV files
result = fexec.map(read_csv, uris)

#Wait for the results
times = fexec.get_result(result)
read_time = max(times)
```

Code 4. Lithops code for data ingestion

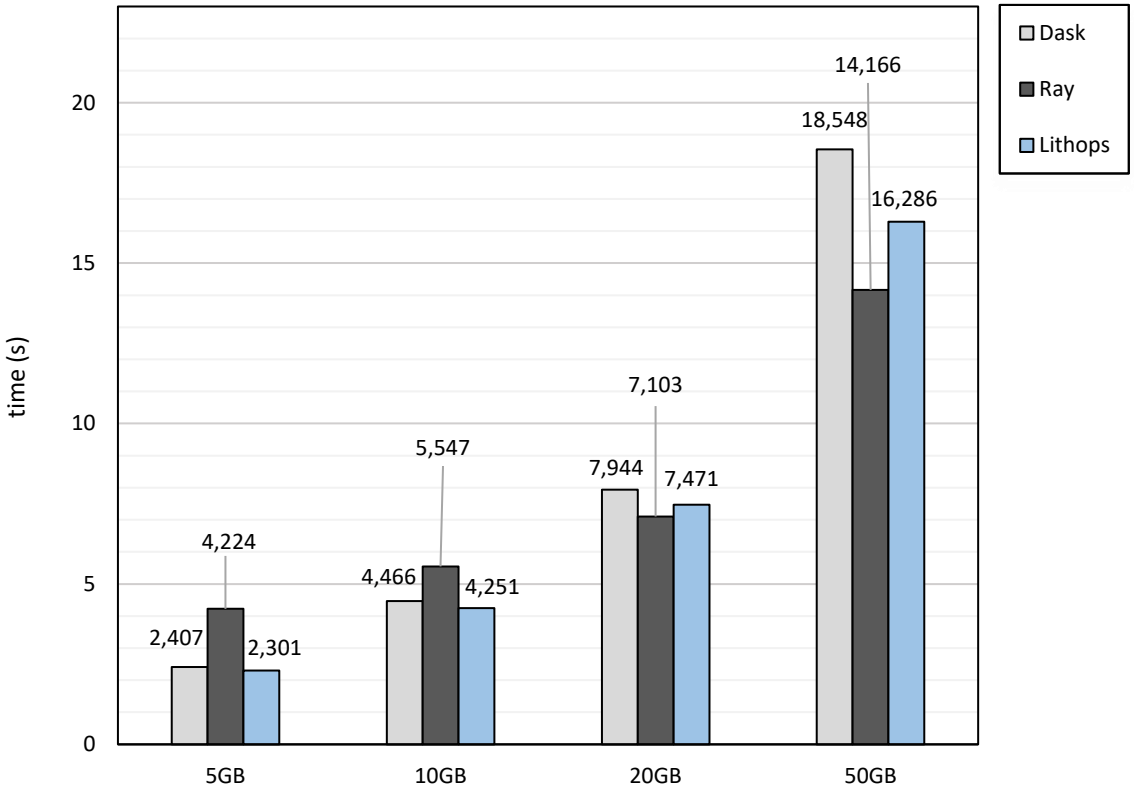
4.2.4. ***Results obtained and analysis***

The times obtained by the three frameworks in the warm state are quite similar for all the ingested data sizes as described in Plot 5. The first plot illustrates the results from the 32-worker clusters and the 32-function execution for Lithops. The differences are minimal and do not demand excessive concern, as times can slightly vary depending on the network traffic at any given moment.



Plot 5. Dask vs Ray vs Lithops data ingestion 32 CPUs

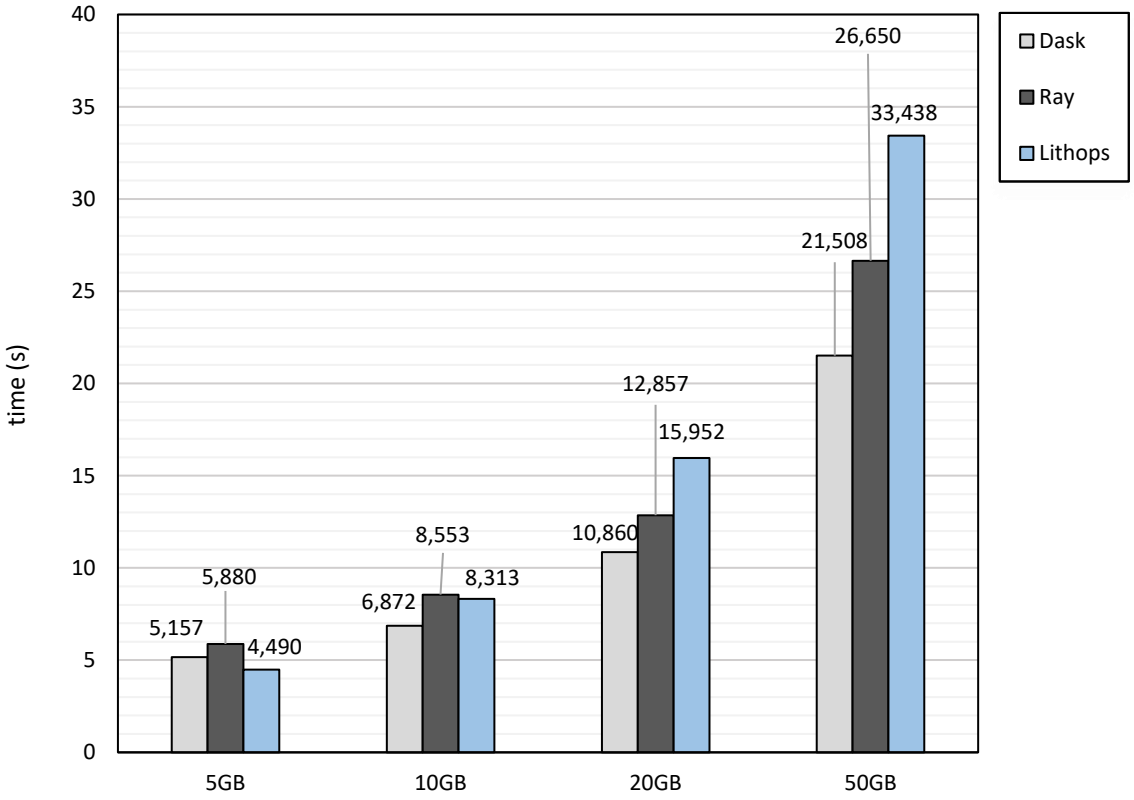
Plot 6 displays the ingestion times of the 128-CPU clusters, and here some interesting differences can be noticed. For smaller sizes, Dask and Lithops are faster than Ray, but as the size increases this behavior is reversed.



Plot 6. Dask vs Ray vs Lithops data ingestion 128 CPUs

When comparing the times from the two previous plots, it becomes clear that higher parallelism leads to less processing time. While the 32-worker clusters require over 60 seconds to ingest 50 GB of data, the 128-worker clusters achieve the same task in less than 20 seconds.

Finally, cold times corresponding to the first execution of data ingestion with 128 workers have also been measured in Plot 7. In this scenario, Lithops is the most negatively affected technology, yielding times that are 2.02x slower than in warm executions. However, Dask and Ray are also significantly impacted in cold executions, being 1.55x and 1.66x slower, respectively.



Plot 7. Dask vs Ray vs Lithops data ingestion 128 CPUs cold start

The main conclusion we can draw from this experiment is that both the studied cluster technologies and Lithops perform data ingestion in similar times, thus preventing us from definitively favoring one over the other.

However, as mentioned earlier, if the data had not been arranged in the optimal number of files, the results would have differed. The primary disadvantage would have been for Ray, given its inability to partition data with a customized chunk size. Therefore, it can be asserted that when the data cannot be divided into numerous files, Dask and Lithops become more suitable options.

4.3. Autoscaling

4.3.1. ***Description***

So far, we have worked with clusters that were limited in terms of scaling, as we prevented them from scaling up or down. In the case of Ray, we restricted this by assigning the same value to *min_workers* and to *max_workers* in the configuration file. While for Dask, we simply didn't configure the cluster's adapt option. The goal of this experiment is to study the ability of both cluster technologies to adapt to workload changes by adding and removing nodes or workers.

In this demonstration, we aim to illustrate the efficiency gains in compute speed achieved by leveraging Ray and Dask clusters. The experiment involves a compute-intensive image processing task, specifically palette extraction, using a sizable dataset comprising randomly selected images, all of approximately the same size.

To facilitate this test, we have developed a streamlined application solely dedicated to extracting the color palette from this dataset of random images. The images are selected to ensure uniformity in size, providing a consistent and controlled environment for analysis. These images are stored in an S3 bucket.

Figure 18 showcases the rendered output resulting from the processing of a single sample from this dataset. For simplicity in this demonstration, we have chosen not to persist or visualize the output.

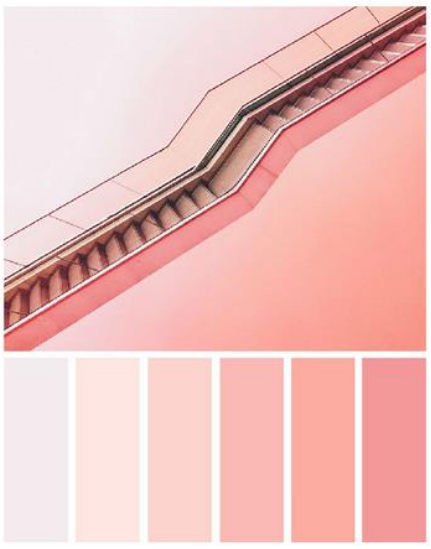


Figure 18. Rendered output of processing a single sample

Therefore, we created clusters with 4 initial workers and the ability to scale up to a maximum of 50 workers to observe the time required for both upward and downward scaling, as well as how they performed. Furthermore, we have tested with different quantities of images: 1000, 2000, 5000, 10000, 15000, 20000.

Additionally, we have executed it on a serverless technology such as Lithops to establish a baseline for performance.

This experiment is based on a post from the Anyscale [25] webpage that discusses Ray clusters. From there, we have adapted it to our needs, not only by implementing it in Dask but also by incorporating features such as fetching images from S3, among others.

*4.3.2. **Setup and Specifications***

	VMs	**K8s**
Cloud provider	AWS	IBM, GCP
Cloud services	EC2	IKS, GKE
Instance family	m6i	cx2, e2-standard
Region	us-east-1	us-south-2, us-central1
Ray version	2.5.0	2.5.0
Dask version	2023.4.0	2023.4.0

Table 14. General specifications for Autoscaling test

Lithops configuration:

	Lithops
Cloud provider	AWS
Cloud services	Lambda
Lithops version	2.9.0
Region	us-east-1
# Lambda functions	as many as images

Table 15. Lithops configuration for Autoscaling test

Cluster configurations:

	Nodes	Instance type	CPUs x node	CPUs / Workers
Cluster VMs	2 - 25	m6i.large	2	4 - 50
Cluster K8s	7	e2-standard-8 cx2.8x16	8	4 - 50

Table 16. Cluster configurations for VMs cluster and K8s cluster in Autoscaling test

With respect to the capacity for auto-scaling downwards, it is of significance to underscore that Ray provides the facility to configure the idle node's time-to-live. This entails that after the designated time interval, the node will be terminated. Conversely, such a concept lacks applicability in Dask, as it is not conceived as a technology intended to sustain an active cluster in an idle state.

4.3.3. ***Procedure***

There is an S3 bucket named *my-image-set* that works as a working directory where there are 20 images of similar size (all less than 1MB) stored. The first thing to do is load up the ids from the S3 bucket, which is done by a function called *arrange_data* that gets the objects of the bucket, and then we apply the *create_ids* method to get as many images as it is needed:

```
def arrange_data(bucket_name):
    # Create a Boto3 S3 client
    s3_client = boto3.client('s3')
    # List objects in the specified bucket
    objects = s3_client.list_objects_v2(Bucket=bucket_name)
    # Extract file names from the objects
    file_names = [obj['Key'] for obj in objects['Contents']]
    uris = []
    for key in file_names:
        uris.append(f's3://{bucket_name}/{key}')
    return uris


# Make as many ids as it is needed
def create_ids(desired_num, uris):
    i = 0
    desired_uris = []
```

```
    while i != desired_num:
        desired_uris = desired_uris + uris
        i = len(desired_uris)
    return desired_uris
```

Code 5. Source code to load images from S3 bucket for Autoscaling

Then we defined another method called *get_palette* to extract the palette using the ColortThief class. This method relies on a separate function called *load_image* that loads up the image so it can be processed by the ColorThief *get_palette* class method:

```
def load_image(image_url):
    session = boto3.session.Session()
    # Next, we create a resource client using our thread's session
object
    s3 = session.resource('s3')
    #s3 = boto3.resource('s3')
    bucket_name, key = image_url.split('/', 3)[2:]
    obj = s3.Object(bucket_name, key)
    return io.BytesIO(obj.get()['Body'].read())

def get_palette(image_id):
   return ColorThief(load_image(image_id)).get_palette(color_count=6)

uris = arrange_data('my-image-set')
ids = create_ids(20000, uris)
```

Code 6. Source code to get the palette rendered from one image in Autoscaling test

Finally, there is a loop through the image ids sequentially. The code also uses the 'tqdm' package to show a progress bar and outputs the total processing time at the end. Here is the fundamental code without executing Ray or Dask atop:

```
start = time.time()
palettes = []

for img_id in tqdm(ids):
    palettes.append(get_palette(img_id))

print("Finished {} images in {}s.".format(len(ids), time.time() - start))
```

Code 7. Base code of Autoscaling test without Dask or Ray

Before explaining the Dask or Ray approaches we have created, it is important to highlight that prior to the following code, the deployment of clusters has been carried out, whether with EC2 or K8s, on which the code will be executed. Additionally, all aspects related to S3 access and image creation, such as the functions *load_image*, *arrange_data*, and *create_ids*,

remain the same for both approaches. Therefore, they will not be reflected in the following code snippets.

Ray approach

To begin, the ray module should be imported, and *ray.init()* should be employed for initialization purposes. Moreover, the *@ray.remote* decorator must be added to the function (in this case, *get_palette*) to convert it into a Ray task suitable for parallelization.

```
ray.init(address='auto')

@ray.remote
def get_palette(image_id):
    return ColorThief(load_image(image_id)).get_palette(color_count=6)

uris = arrange_data('my-image-set')
ids = create_ids(20000, uris)

start = time.time()
palettes = []

for img_id in ids:
    palettes.append(get_palette.remote(img_id))

def progress_bar(obj_refs):
    ready = []
    with tqdm(total=len(obj_refs)) as pbar:
        while len(obj_refs) > 0:
            new_ready, obj_refs = ray.wait(obj_refs, num_returns=10)
            pbar.update(len(new_ready))
            ready.extend(new_ready)
    return ready

palettes = ray.get(progress_bar(palettes))
print("Finished {} images in {}s.".format(len(ids), time.time() - start))
```

Code 8. Source code for Ray's approach of Autoscaling test

Secondly, the loop necessitates alteration to achieve asynchronous execution of the *get_palette* task via the utilization of *get_palette.remote()*, as opposed to directly invoking the function.

Dask approach

```
# Decorate the function with `delayed` to create a lazy computation
get_palette_dask = delayed(get_palette)

uris = arrange_data('my-image-set')
ids = create_ids(20000, uris)

start = time.time()
# Create a list to store the delayed computations
palettes = []
# Loop over the image IDs and create delayed computations
for img_id in ids:
    palettes.append(get_palette_dask(img_id))

# Execute the delayed computations using Dask
results = client.compute(palettes)
progress(results)
print("Finished {} images in {}s.".format(len(ids), time.time() - start))
```

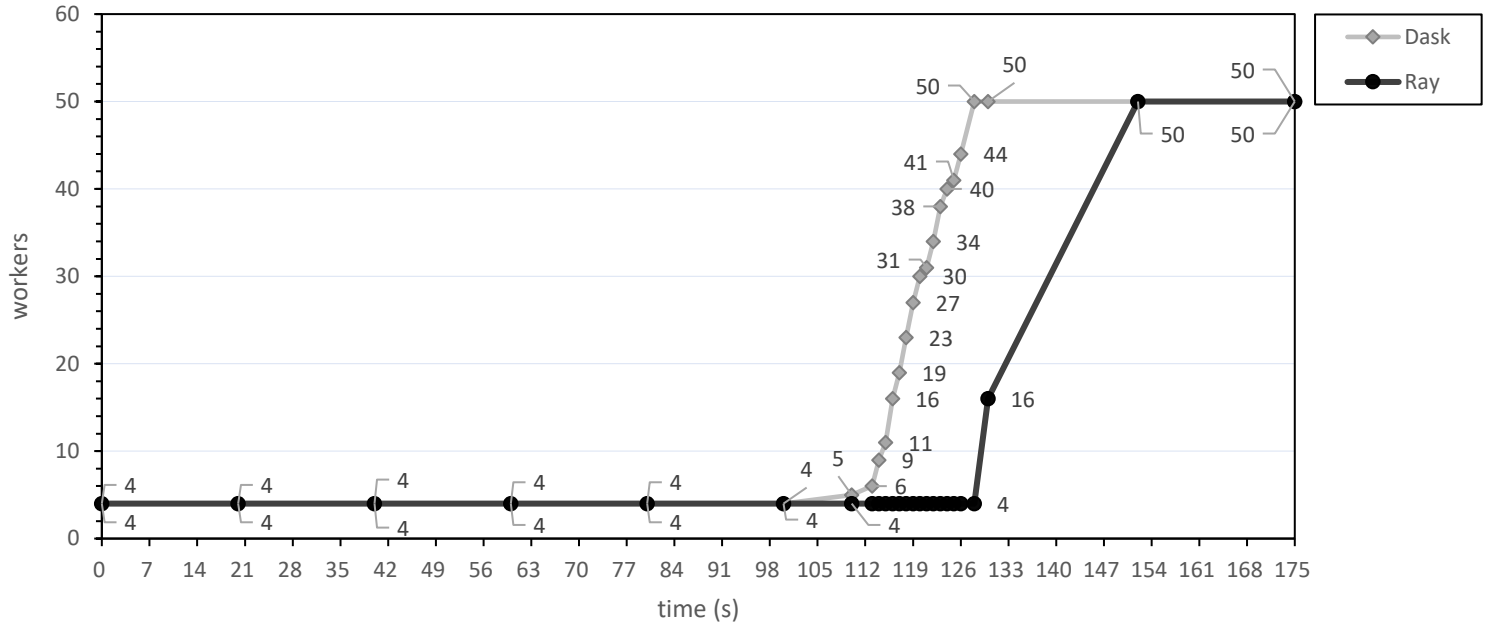
Code 9. Source code for Dask's approach of Autoscaling test

In Dask, the *get_palette* function is enveloped with the Dask delayed decorator, deferring computations until a later time. Following this, the image URIs are organized, and the corresponding image IDs are generated. Within a loop, delayed computations for each image ID are established and collected. Ultimately, the Dask *client.compute* method coordinates the parallel execution of these computations, culminating in an optimized efficiency for managing extensive datasets and intricate computational tasks.

4.3.4. ***Results obtained and analysis***

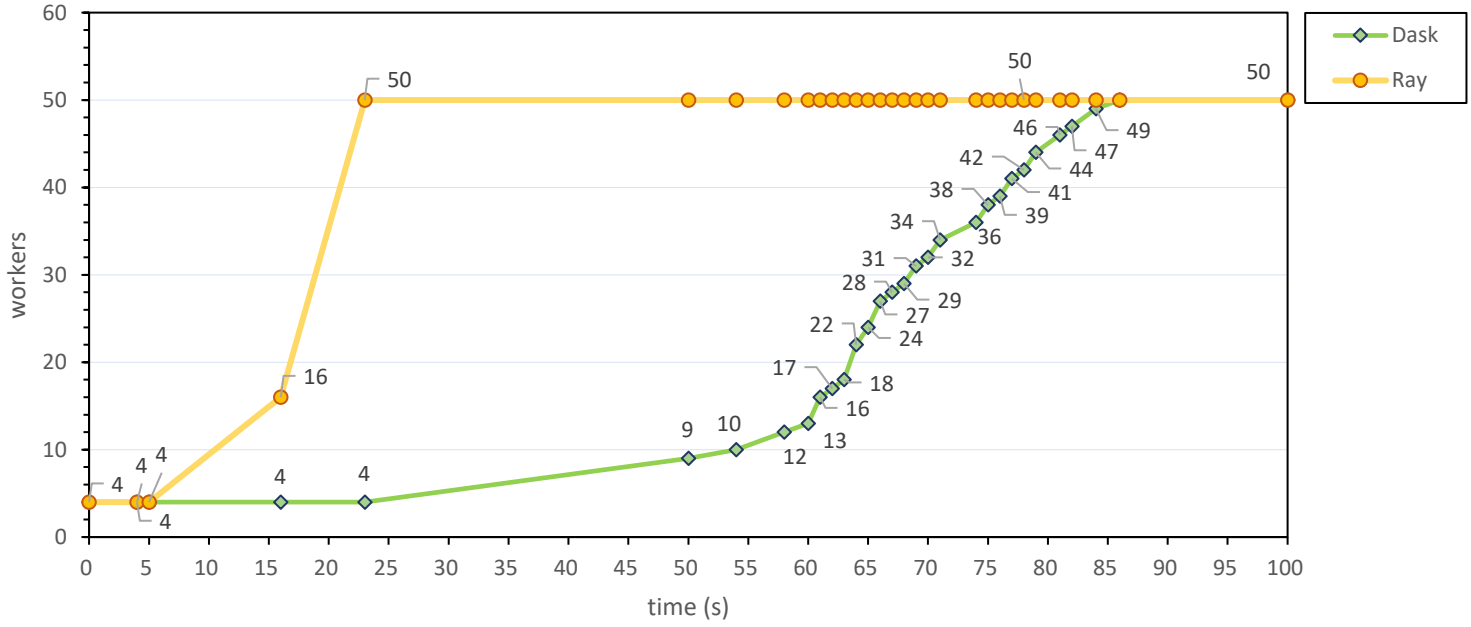
Scaling up

First of all, it is important to emphasize that the time required for a cluster to auto-scale depends directly on the initialization time of the nodes. Consequently, the velocity at which the cluster scales is intricately linked to the quantity of packages installed on these nodes, resulting in either faster or slower auto-scaling accordingly.



Plot 8. Dask vs Ray scaling up in VMs cluster

So as we can see when scaling up using VMs in Plot 8, Dask scales faster probably due to requiring less time to create the specific VMs for this experiment. But Ray scales in a more linear way.

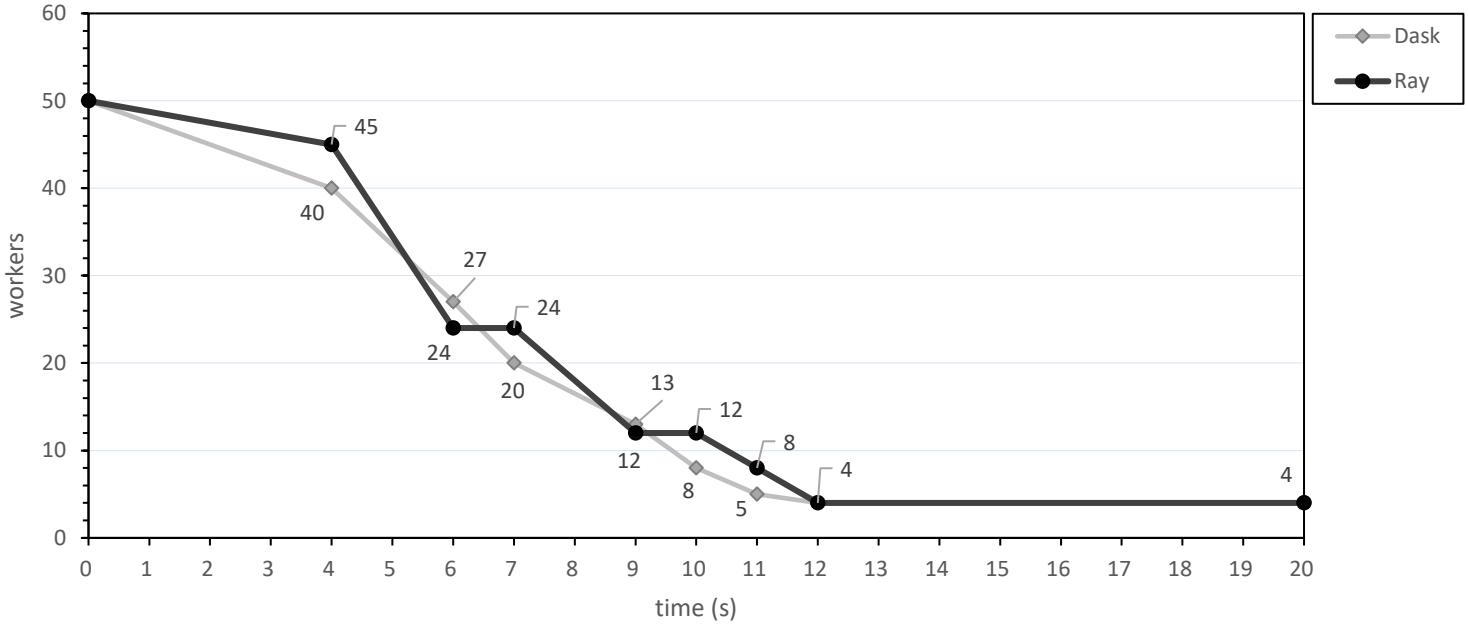


Plot 9. Dask vs Ray scaling up in K8s cluster

On the contrary, when employing K8s as seen in Plot 9, the process of creating workers (pods) is indeed expeditious; however, Dask requires a significantly longer period (approximately 20 seconds) to commence its scaling up. Conversely, Ray incorporates all the workers simultaneously.

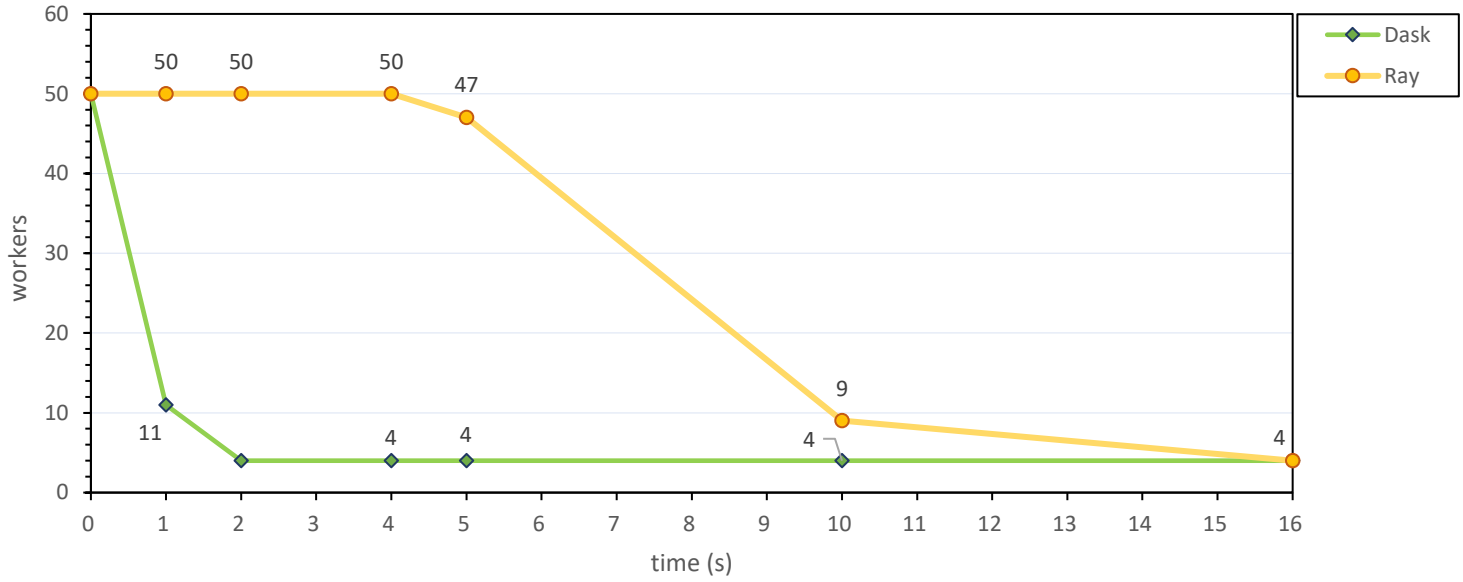
Scaling down

To induce rapid downscaling of clusters, once all tasks are completed, we request the execution of an additional set of 20 tasks that do not require all cluster's resources.



Plot 10. Dask vs Ray scaling down in VMs cluster

In the realm of virtual machines, we observe in Plot 10 a similar behavior in both technologies, whereby they transition from 50 to 4 workers within a brief 12-second interval. However, it is noteworthy that in the context of Kubernetes (k8s) presented in Plot 11, Dask exhibits a swifter elimination of workers compared to Ray.



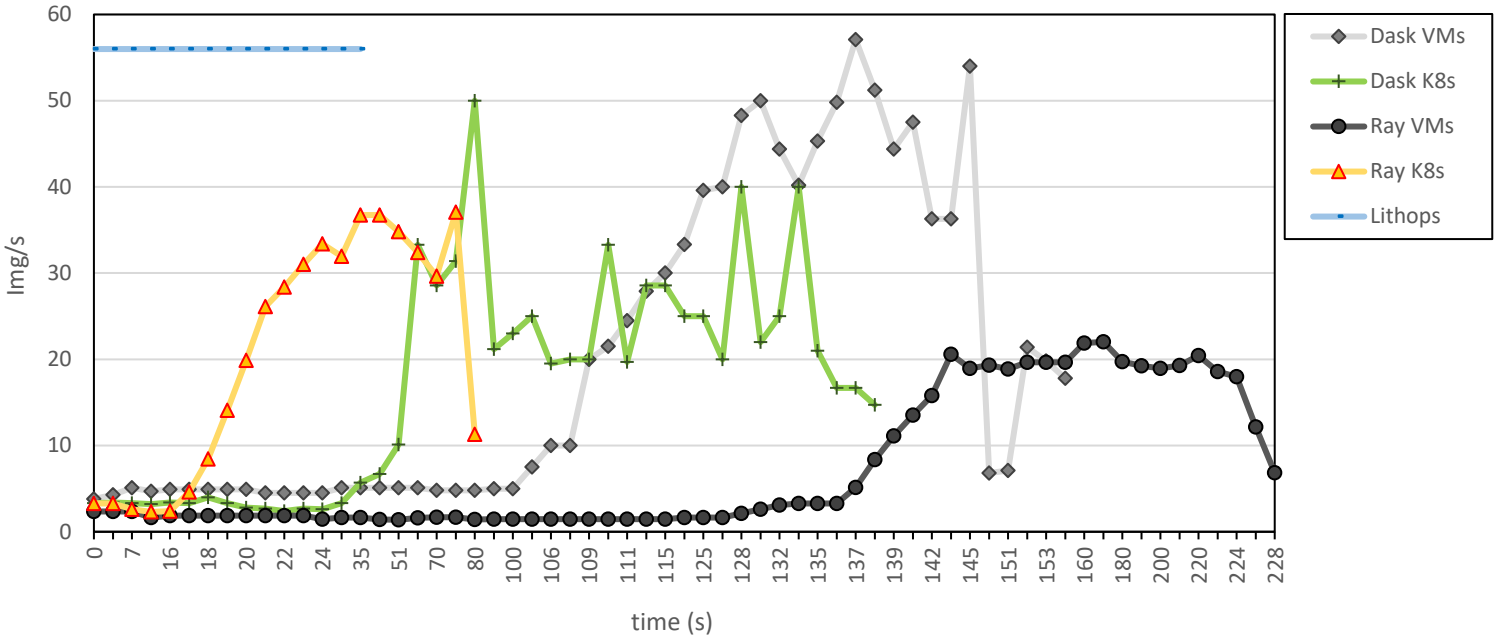
Plot 11. Dask vs Ray scaling down in K8s cluster

So as a conclusion, in the domain of scaling capabilities, both Dask and Ray showcase impressive performance within Kubernetes infrastructure. However, when shifting to VMs, the initiation of the autoscaling process becomes notably challenging. The adoption of a serverless technology like Lithops offers a remedy for this predicament.

Performance

To assess the performance, we have utilized the results obtained from analyzing 2000 images as it is displayed in Plot 12. We have evaluated the rate at which images are processed per second. From the observations, it becomes evident that Lithops demonstrates a significantly higher speed due to its immediate launch of the requisite functions for executing the designated workload.

On the other hand, we also note that Dask exhibits occasional higher peaks of processing, but its performance is comparatively more irregular. Additionally, K8s achieves greater swiftness as it manages to provision workers and scale promptly, thus justifying its faster performance.



Plot 12. Autoscaling experiment performance plot

Finally, Ray, using Kubernetes, also exhibits enhanced speed for the same reason as Dask. However, Ray maintains a more consistent throughput.

5. Waterfall model for Hybrid Data Ingestion

The main goal of this thesis is to separately compare cluster and serverless technologies in order to determine the strengths and limitations of each. However, one of the initial questions posed was whether it made sense to combine the use of these two technologies, and in which situations would that implementation be useful.

In this section, the implementation of a simple hybrid model that combines the use of cluster and serverless technology is demonstrated to perform data ingestion and processing. The development of this implementation has been done following a waterfall model.

5.1. Requirements

In this hybrid setup, we will be reading data and applying a filter, followed by a sorting operation on the filtered data. In this case, our underlying goal is to optimize the cluster model, which is why we aim to improve not only computational time but also its cost efficiency.

Therefore, in this hybrid approach, we perform data reading and filtering using Lithops, and the sorting operation on the filtered data is carried out using cluster technology. This approach allows us to reduce substantially the number of Dask or Ray workers required to perform the computation as we only need them to perform the sort. Consequently, it will likely reduce the costs too.

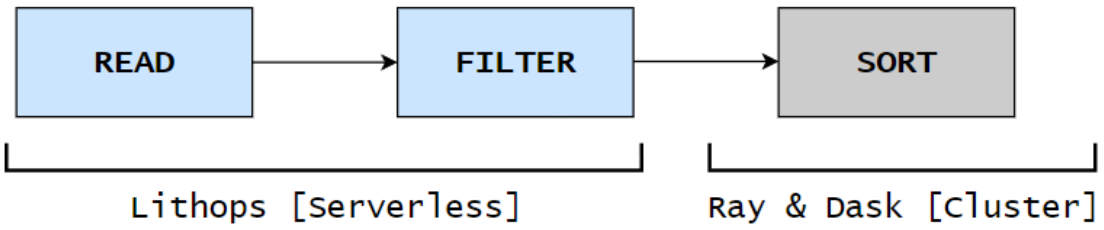


Figure 19. Hybrid sequence

In order to integrate the code in a seamless manner that remains unnoticeable at first glance, the serverless segment of the code is written into a separate file. So, the user perceives it as if they are only running the cluster technology (Ray or Dask).

5.2. Design and implementation Dask+Lithops

The implementation of the Dask+Lithops hybrid is based on first reading and filtering the data using Lithops, and then applying operations like sorting with Dask. Unlike Ray, Dask does not allow for an integration where Lambda functions can be launched from within the cluster and be worth it. That is why in this design, we first read and filter the data using Lithops. To streamline communication with Dask, we upload the data to an S3 bucket and then load it with Dask for sorting.

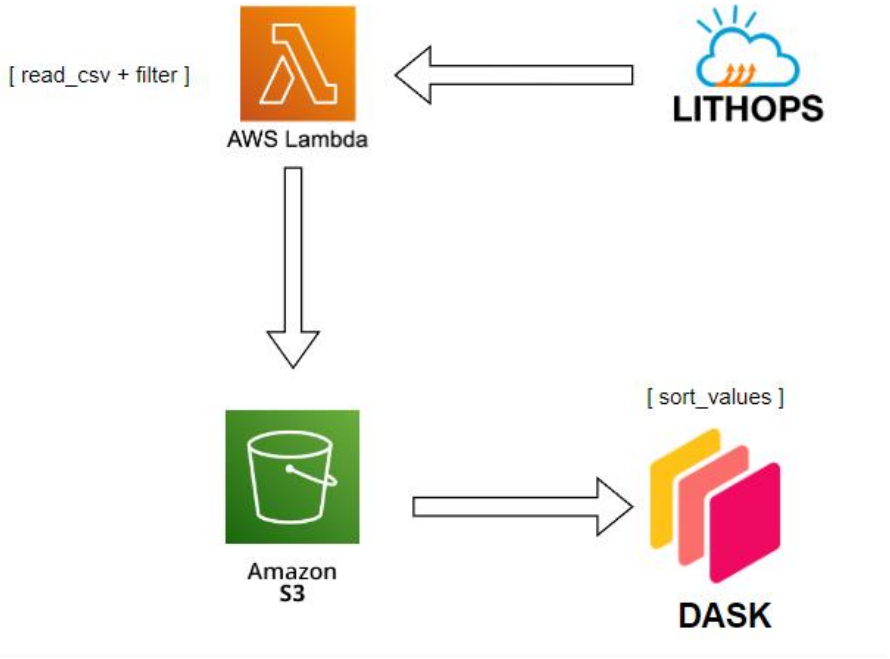


Figure 20. Dask + Lithops model structure

We use S3 to load the data from Lithops to Dask because it is faster than transforming all the data from the Pandas Dataframe to a Dask Dataframe and joining them into a single dataframe. The architechture implemented is shown in Figure 20 and its sequence diagram in Figure 21.

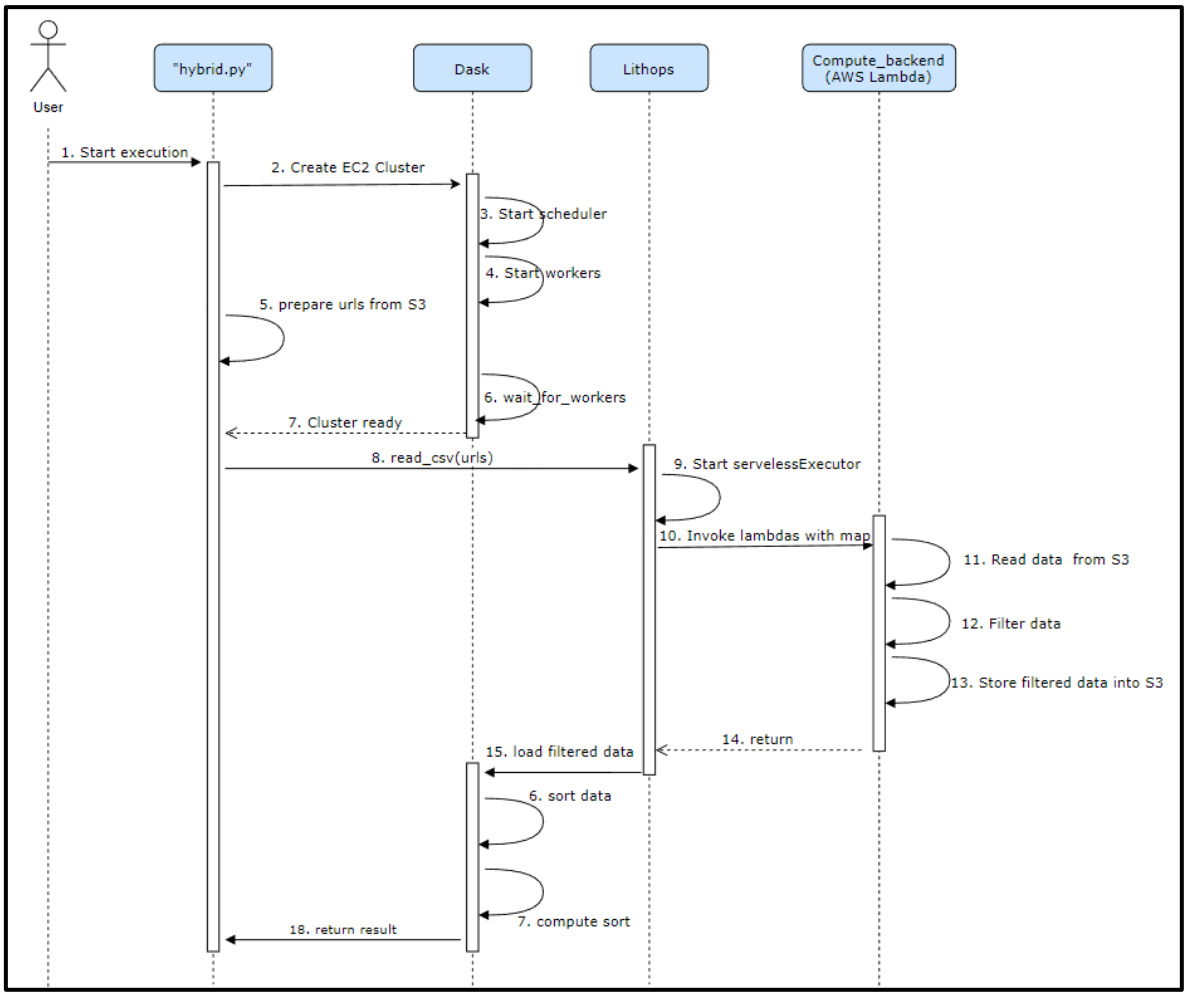


Figure 21. Sequence diagram for Dask+Lithops

As mentioned in the requirements, the goal is for the user to not realize that Lithops is being executed to read the data initially. That is why we have created a decorator as the one in Code 10 (for both the Dask and Ray versions) that enables executing any function in a serverless manner by simply applying the decorator to the function.

```
def taskless(func):
    def wrapper_taskless(*args, **kwargs):
        # Initialize Lithops ServerlessExecutor
        fexec = lithops.ServerlessExecutor(runtime='lithops-runtime-2', runtime_memory="4096")
        result = fexec.map(func, args[0])
        dfs = fexec.get_result(result)
    return wrapper_taskless
```

Code 10. Dask+Lithops Hybrid model decorator function

The decorator is named "taskless", and essentially, it initializes the Lithops ServerlessExecutor using a runtime preconfigured and adapted to the code's requirements. Afterwards, it applies a map operation with the received function and arguments as parameters. This decorator in the case of Dask is located in a file in the Dask source code and imported with "dask.decorator".

The code of the source file for the Dask+Lithops hybrid version, visible in Code 11, starts with the deployment of the cluster as done in the Benchmarking experiments. Then with the function "*arrange_data*" we get the urls of the files in the S3 bucket and prepare them for Lithops.

When the cluster is ready, the *read_csv* function is called with the urls. This function has the decorator so it is called by a map done in Lithops as shown in the "taskless" decorator above. Each read_csv function of the map basically reads one file of the S3 bucket, filters the data where *['ss_net_profit'] > 5000]* using the *loc* method of Dask Dataframe and store the results in another S3 bucket directory.

```
import configparser
import contextlib
import os
import sys
import dask
import time, boto3, s3fs, random, lithops
import botocore.config
from dask_cloudprovider.aws.ec2 import EC2Cluster
from dask.distributed import Client
import dask.decorator
import dask.array as da
import dask.dataframe as dd
import pandas as pd
import numpy as np

workers = 4

def get_aws_credentials():
    # Read in your AWS credentials file and convert to environment variables.
    parser = configparser.RawConfigParser()
    parser.read(os.path.expanduser('~/.aws/config'))
    config = parser.items('default')
    parser.read(os.path.expanduser('~/.aws/credentials'))
    credentials = parser.items('default')
    all_credentials = {key.upper(): value for key, value in [*config, *credentials]}
    with contextlib.suppress(KeyError):
        all_credentials["AWS_REGION"] = all_credentials.pop("REGION")
    return all_credentials

env_vars = get_aws_credentials()

# The workers will need `s3fs` to read from S3.
```

```
env_vars["EXTRA_PIP_PACKAGES"] = "s3fs lithops"

def arrange_data(bucket_name, directory_name):
    # Create a Boto3 S3 client
    s3_client = boto3.client('s3')
    # List objects in the specified bucket
    objects = s3_client.list_objects_v2(Bucket=bucket_name, Prefix=directory_name)
    # Extract file names from the objects
    file_names = [obj['Key'] for obj in objects['Contents']]
    file_names.pop(0)
    uris = []
    for key in file_names:
        uris.append(f's3://{bucket_name}/{key}')
    return uris

@taskless
def read_csv(s3_uri):
    df = pd.read_csv(s3_uri, dtype=np.float32)
    filtered = df.loc[df['ss_net_profit'] > 5000]
    index = random.randint(0, 1000)
    filtered.to_csv("s3://data-ingestion-s3/tmp_5/result"+str(index)+".csv")
    return 0

cluster = EC2Cluster(docker_image="daskdev/dask:2023.4.0-py3.9",
                     security=False,
                     # Set our region here to be the same as our credentials.
                     region=env_vars["AWS_REGION"],
                     scheduler_instance_type="m6i.large",
                     worker_instance_type="m6i.large",
                     bootstrap=True,
                     n_workers=workers,
                     filesystem_size=40,
                     worker_options={'nthreads': 2},
                     # Pass along our environment variables to the workers.
                     env_vars=env_vars
                     )

client = Client(cluster)
print(client.dashboard_link)

uris = arrange_data('data-ingestion-s3', '50GB_128')

# Wait for our workers to be ready for processing.
if workers > 0:
    client.wait_for_workers(workers)

st = time.time()

# Submit the computation task
filtered_df = read_csv(uris)
combined_df = dd.read_csv("s3://data-ingestion-s3/tmp_50/*.csv", assume_missing=True,
blocksize="80MB")
sorted_df = combined_df.sort_values('ss_net_profit').compute()

elapsed = time.time() - st
print(sorted_df)
print("Time spent: ", elapsed)
cluster.close()
```

Code 11. Dask+Lithops Hybrid model

Once the initiall data is read and filtered by Lithops, Dask reads the result from S3 using *read_csv* function with a *"blocksize"* adapted, in this case to 80MB, so every worker reads the same amount of data. Afterwards, the data is sorted by the *"sort_values"* method of Dask Dataframe. It is sorted based on the *"ss_net_profit"* column but it could be any other column. Finally, the sorting operation is computed and the cluster is closed.

5.3. Design and implementation Ray+Lithops

The implementation of the hybrid model that integrates Ray and Lithops exhibits some notable differences compared to the Dask model. The most significant one is that in this case, as showed in Figure 22, lambda functions are launched from within the Ray Cluster.

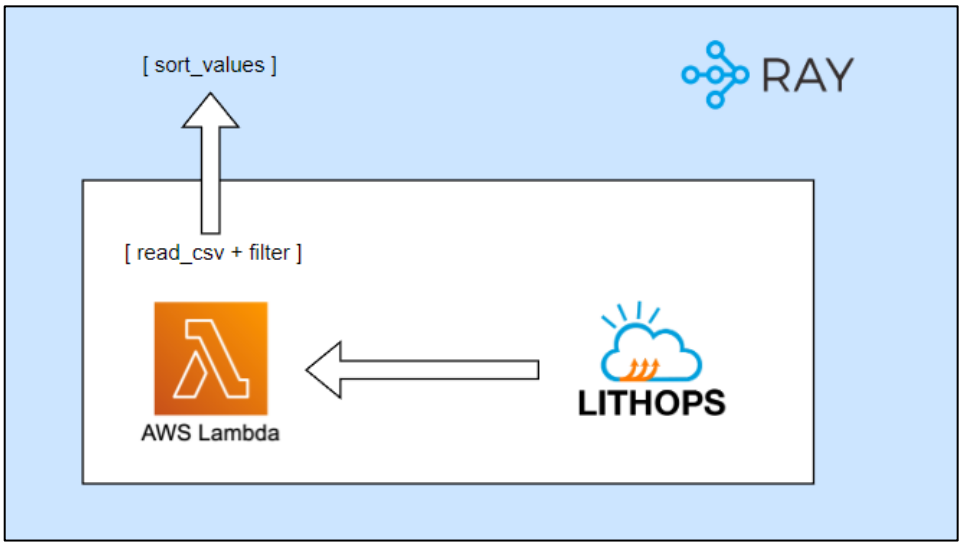


Figure 22. Ray + Lithops model structure

In the previous model lambda functions are launched from outside of the Dask cluster, and once they finish ingesting and filtering the data, the result is sent to the cluster for sorting. In this case, the Lithops Serverless executor is initialized inside a Ray task that is executed within the cluster, so the results returned by the lambda functions are already in Ray's object store memory. In Figure 23 there is a sequence diagram representing this design.

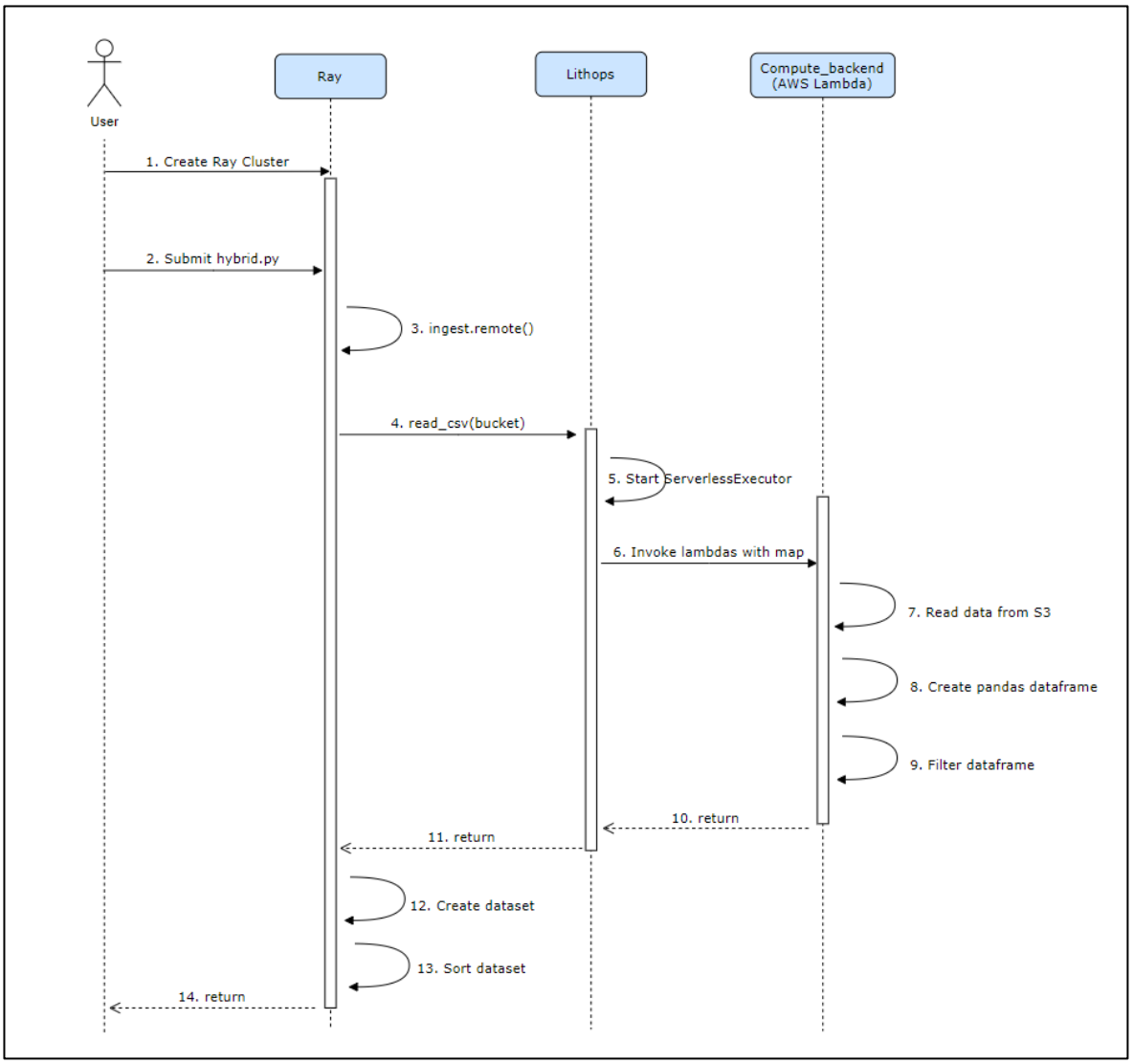


Figure 23. Sequence diagram for Ray + Lithops

Similar to the Dask+Lithops model, a function called "taskless" has been implemented in a file named "taskless.py" within the Ray source code. This function is used as a decorator over the *read_csv* function, which is invoked from the Ray task. This approach effectively hides the use of Lithops, and the user does not see the code required to ingest the data in a serverless way.

```
def taskless(func):
    def wrapper_taskless(*args, **kwargs):
        fexec = lithops.ServerlessExecutor(runtime='hybrid-runtime', runtime_memory="4096")
        result = fexec.map(func, args[0], obj_chunk_number=1)
        dfs = fexec.get_result(result)
        return dfs
    return wrapper_taskless
```

Code 12. Ray+Lithops Hybrid model decorator function

The taskless function initializes a Lithops Serverless Executor using a previously deployed runtime and assigning each lambda function a memory of 4096 MB, as done in data ingestion tests. Subsequently, the decorated *read_csv* function is invoked through a map operation, passing the bucket name where the files are located as a parameter.

The *obj_chunk_number* is used to determine the number of chunks to split each file. In this experiment data is divided in 128 files and we want one lambda function to process each file. Therefore, it can be assigned a value of 1 or None. In the scenario where all the data is in a single file, if we wanted to keep the same number of function activations, a value of 128 would need to be assigned.

Finally, the function waits for the lambda function results using the *get_result* method. These results consist of a list of dataframes created and filtered using the pandas library.

```
import ray
import pandas as pd
import time
from io import BytesIO
from ray import taskless

bucket_name = 's3://data_to_ingest/50GB_128/'

@taskless
def read_csv(obj):
    data = obj.data_stream.read()
    if (obj.part == 1):
        df = pd.read_csv(BytesIO(data), header=0, dtype=float)
    else:
        df = pd.read_csv(BytesIO(data), names = […], dtype=float)

    filtered = df.loc[df['ss_net_profit'] > 5000]
    return filtered

@ray.remote
def ingest(bucket):
    results = read_csv(bucket)
    dataset = ray.data.from_pandas(results)
    return dataset

ray.init(address='auto')
```

```
st = time.time()

futures = ingest.remote(bucket_name)
dataset = ray.get(futures)
sorted = dataset.sort('ss_net_profit', descending=True)
sorted.materialize()

elapsed = time.time() - st
print('Elapsed: ', elapsed)
```

Code 13. Ray+Lithops Hybrid model

In Code 13 there is the code that is executed within the Ray Cluster. As previously mentioned, the utilization of Lithops remains entirely concealed both in the list of imports and throughout the rest of the code. The only additional import required is the "taskless" file to be able to use the decorator.

The decorated function named *read_csv* is executed by each of the 128 lambda functions. It generates a pandas dataframe from the data received in byte format, thus requiring the io library. Once the dataframe is created, the function filters the data by retaining only the rows with a value greater than 5000 in the specified column, utilizing the "loc" method offered by pandas.

Following the "read_csv" function, the Ray Task named *ingest* can be found. This task is executed by a ray worker and is responsible for invoking the decorated *read_csv* function. It also uses the *from_pandas* method from Ray Data library to join all the dataframes returned by the lambda functions to create a Ray dataset.

Once the Ray Cluster has the data in dataset format, it can swiftly and efficiently apply any transformation to it. In the case of this experiment, the dataset is comprised of 128 blocks, meaning each operation applied will trigger a total of 128 Ray tasks. In the code, we observe the execution of the *"sort"* operation based on the specified column's value. Finally, the *"materialize"* method is invoked to activate the operation and enable the measurement of the execution time.

5.4. Tests

In this section it is described how the implementations of hybrid versions have been tested. It is important to emphasize that the tests have been done capturing the times in a warm state.

5.4.1. Set up and specifications

	Nodes	Instance type	CPUs x node	Total CPUs / Workers
Cluster model	64	m6i.large	2	128
Hybrid model	4	m6i.large	2	8

Table 17. Hybrid and cluster model specs

The data to process is stored 128 csv files located in a S3 bucket named data_to_ingest.

5.4.2. Cluster baseline

To establish a baseline for comparing the results of the hybrid model, we have implemented the same code without utilizing Lithops and only using the functions provided by the Ray and Dask APIs. Both implementations can be found in Appendix A.

In Dask, the *read_csv* method from Dask.Dataframe library carries out all the data ingestion as demonstrated in the Data Ingestion experiment. Then, the filter is done with the "loc" function of Dask Dataframe and works as the "loc" indexer from Pandas Dataframe. This method allows the user to select data based on row and column labels instead of numerical indices. Finally, the sort section is done with *sort_values* function also from Dask Dataframe. This method allows the user to sort the rows of the DataFrame based on one or more columns, arranging the data in ascending or descending order according to the specified column(s).

In the case of Ray, data ingestion is done using the *read_csv* function from Ray Data library, as it was done in the Data Ingestion experiment. As for filtering, the *map_batches* method is used, which applies a given function to many batches of data, deploying one Ray Task for each batch. In this case, it receives a function named "filter" that drops the rows of each dataframe block that do not satisfy our selected condition. Finally, dataset sorting is accomplished using the "sort" function as done in the hybrid model.

5.5. Validation

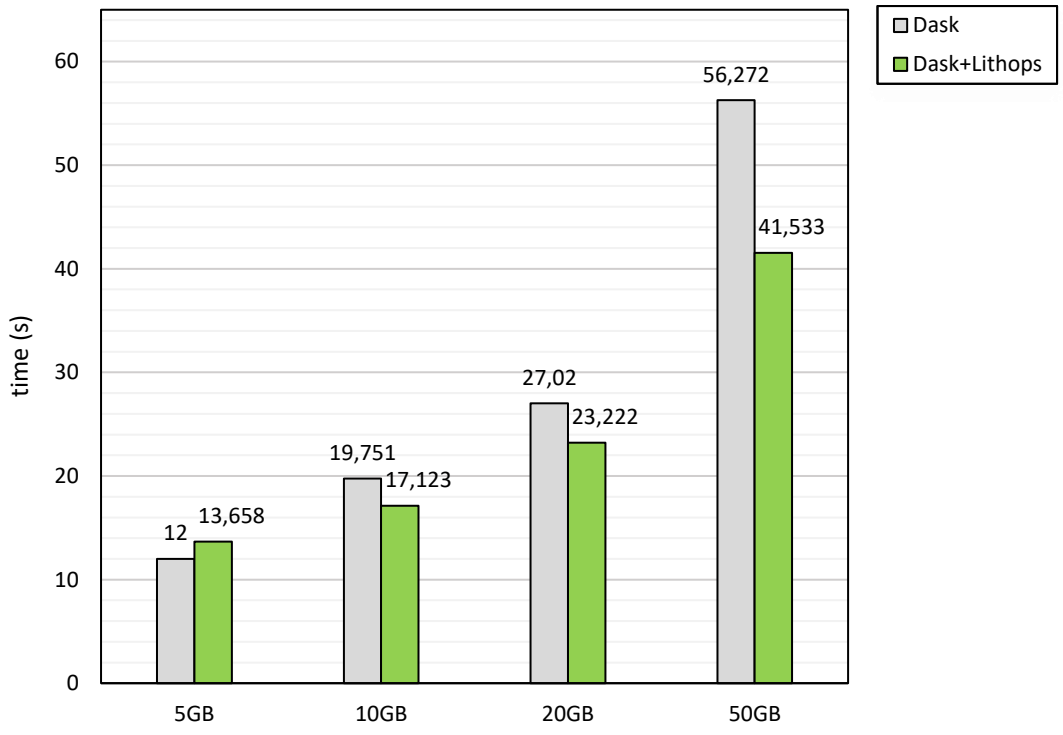
In this section the results of the conducted tests are showed and discussed in order to extract some conclusions regarding the utility of this hybrid model. In addition, there is a cost analysis section where the execution costs for each model are analyzed.

5.5.1. ***Execution times***

Dask hybrid

As seen in Plot 13, for small data sizes both models exhibit similar behaviors, but as the data size increases we observe that the hybrid model shows a progressively more significant improvement, thus establishing a positive trend compared to the times of the Dask model.

At small sizes, the behavior is similar because the time gained by applying filtering within the lambda functions is neutralized by the time it takes to load the files to S3 and read them to distribute the dataframe among the workers. However, as we increase the data, the time it takes to load and read the CSV files of the data subsets obtained from filtering becomes less significant compared to the time it takes to read the original uncompressed dataset and apply the filter within the lambda function. This is why the improvement becomes more substantial over time.

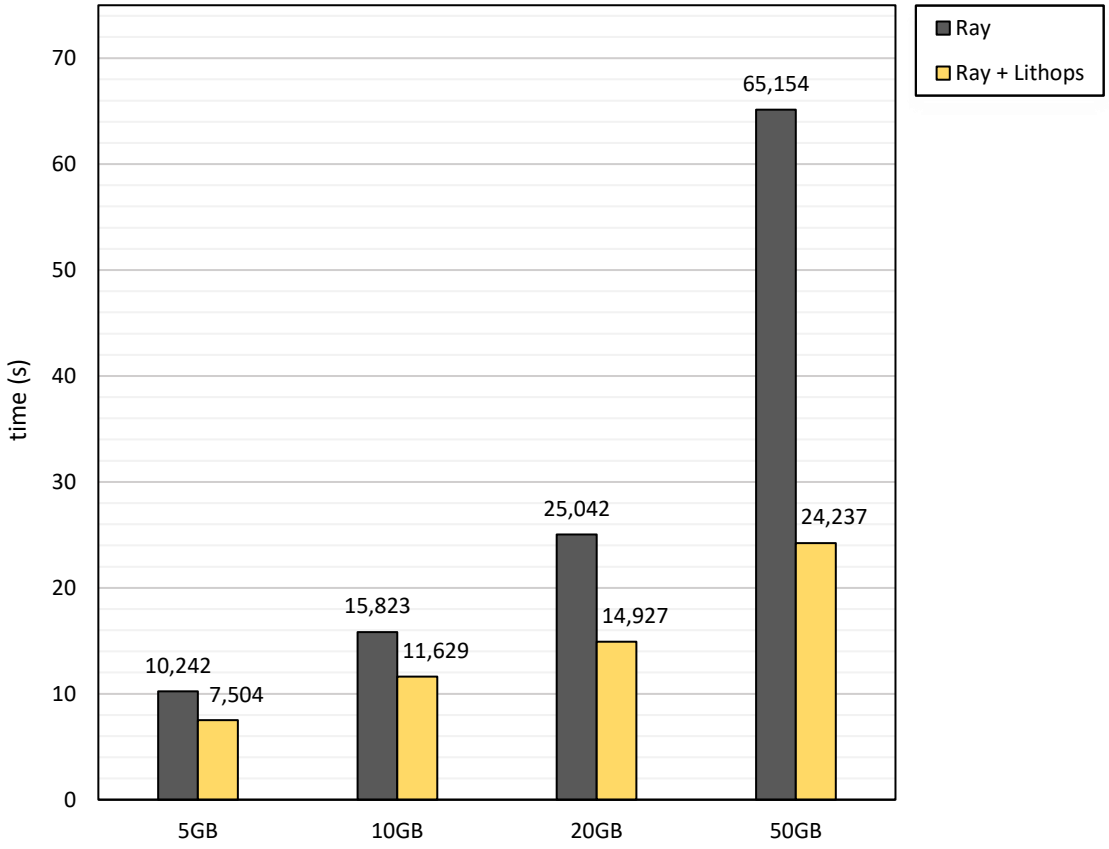


Plot 13. Dask Hybrid comparison

Ray hybrid

As shown in Plot 14, the hybrid model achieves much better execution times than the Ray model, especially as the size of the processed data grows. With 50GB the hybrid implementation runs 2.7x times faster than the cluster one.

This improvement is likely due to the fact that lambda functions filter their data chunks immediately after reading them and creating the dataframe. On the other hand, in the Ray model, there is a "map_batches" call, and each node applies the filter operation locally to its dataset block.



Plot 14. Ray Hybrid comparison

As discussed in the data ingestion experiments, Ray performs quite well for reading data from the cloud object storage as long as the data is partitioned into multiple files. However, when reading from a single file the results are much different.

In Table 18, we can observe the times for the two models reading and processing 5GB of data from a single CSV file. In this case, the time for the cluster model is significantly worse since there is a single worker reading all the data, thus the ingestion process is not parallelized. In contrast, the hybrid model is hardly affected as Lithops enables the file to be divided into 128 chunks and processed in parallel.

Data (GB)	Ray model (s)	Hybrid model (s)
5	387.320	10.265

Table 18. Ray hybrid times for 5GB and single csv file

*5.5.2. **Execution costs***

VMs and Lambda costs:

	Cost per second
m6i.large	$0.00002678
Lambda (4096 MB)	$0.0000667

Table 19. VMs & AWS Lambda costs

First, in Table 19 are the costs per second for the Lambda functions VMs we used. The calculations below are based on these costs.

In Table 20 are the times used to calculate the total costs for each model. The first column corresponds to the time that takes the cluster to deploy the master node, here the workers are not deployed yet.

The next column contains the time required by the rest of the nodes of the cluster to do their initializations, and the execution time corresponding to data processing. Finally, there is also a column showing the running time of lambda functions for the hybrid model.

The costs are calculated based on these times:

	Scheduler / master only (s)	Master & workers (s)	Lambda (s)	Total (s)
Dask+Lithops	96.638	145.466	30.546	242.104
Ray+Lithops	109.287	122.7	18.105	232.921
Dask	96.638	211.984	0	308.622
Ray	109.287	184.842	0	294.129

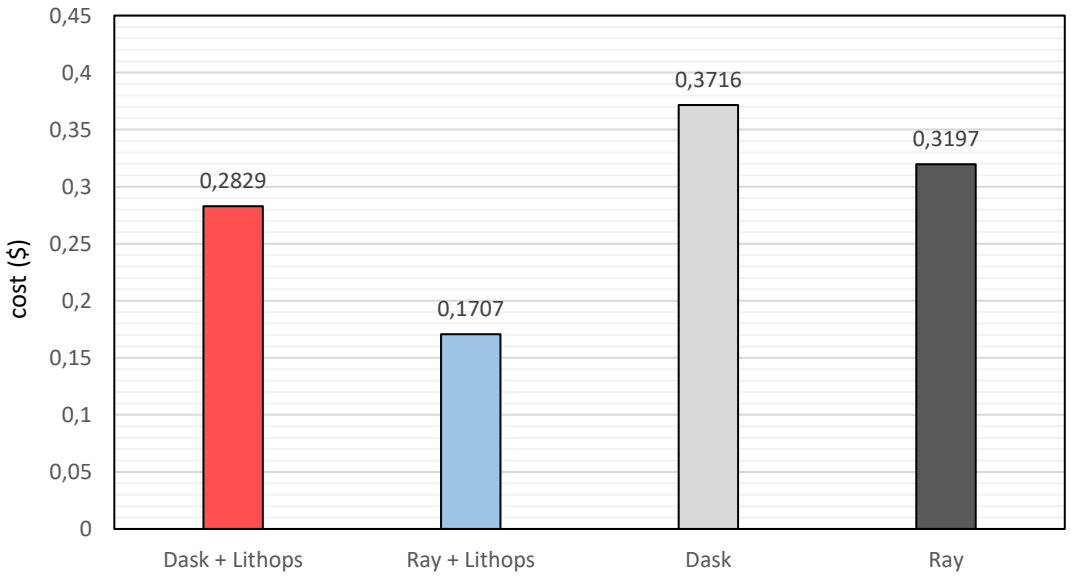
Table 20. Time used to calculate the cost for each model

The total costs generated by the cluster side are calculated considering that initially there is only the master VM running for about 90 seconds, waiting for worker nodes to be ready. When the workers are created, there are 4 machines running in the case of Ray and 5 in the case of Dask for the hybrid; and 64 and 65 machines respectively for the original version.

Here are the results obtained:

	Lambda (s)	Lambda cost ($)	Cluster total (s)	Cluster cost ($)	Total cost ($)
Dask + Lithops	30.546	0.2608	242.104	0.0221	0.2829
Ray + Lithops	18.105	0.1546	232.921	0.0161	0.1707
Dask	0	0	308.622	0.3716	0.3716
Ray	0	0	294.129	0.3047	0.3197

Table 21. Costs calculated for each model



Plot 15. Total cost of each model

6. Insights and comparisons

In this section the results obtained from the benchmarking phase are related with the comparative framework criteria that were defined before carrying out the experiments.

	Cluster	**Serverless**
Deployment	Startup of 3 - 4 min including initialization and configuration	Nearly negligible – 100ms
Cold start issues	Slightly affected (Dask is 1.55x slower & Ray is 1.66x slower on 1st execution)	Significant negative impact (Lithops is 2.02x slower on 1st execution)
Resource utilization	≈120s of under-provisioning in VMs and ≈10 -80s in K8s. Also ≈10s of over-provisioning	0 under-provisioning and 0 over-provisioning
Scalability	≈120s scaling up in VMs and ≈10-80s in K8. And ≈10s scaling down	Instantly – 0s
Control over the infrastructure	Control over all resources of each node, number of workers and how they scale – 100% of control	Control over runtime, amount of functions and memory of each function
Maintenance	Update the versions of packages, more problems of mismatched versions	Minimal maintenance required
Cost-effectiveness	$0.00002678 × #nodes × sec	$0.0000667 × #Lambda × sec

Table 22. Framework comparison Insights

1. **Cluster startup time in VMs (3-4 mins) is significantly higher than serverless one (100ms).** When using Kubernetes, the pods startup times are much lower (15-30 seconds), but it is important to keep in mind that the time it takes to deploy the K8s cluster on a cloud provider is between 5 and 20 minutes. Therefore, for tasks lasting less than 4 minutes, it is clearly better to use serverless options like Lithops. As the task duration increases, startup times become negligible and as a consequence cluster model may be a better option. In addition, to avoid the long delay generated by startups, the user might instead keep a cluster of VMs running, which increases operational complexity further and leads to potentially high under-utilization and thus overall higher cost. For sporadic usage, this complexity has the risk to be too high to be worth doing.

2. **Data ingestion on Cluster technologies can be as fast as Serverless if VMs allocate enough bandwidth and CPUs to load data.** The data ingestion experiment has shown that both cluster and serverless technologies achieve a similar performance.

3. **Serverless (time loss of 2.02x in Lithops) is more affected than Cluster (time loss of 1.55x in Dask and 1.66x in Ray) by the cold start issue.** For applications with low workloads such as data ingestion with minimal data, the impact is similar between VMs and Lithops. However, as we increase the workload, the cold start increasingly affects serverless, causing slower performance.

4. **Cluster technologies have worse resource utilization due to under-provisioning when scaling up and over-provisioning when scaling down.** In cluster technologies the user needs to choose the number and type of VM instances, which may have a significant impact on the performance and price of the application. In case the amount of resources needed is uncertain, the problem of over-provisioning and under-provisioning of resources might appear. If the cluster needs to scale up, there will be about 120s of under-provisioning using VMs, and between 10s and 80s when using K8s. In the case the cluster needs to scale down there will be ≈10s of over-provisioning. However, if the user knows the exact number of resources that will be required that should not be a problem. As for serverless technologies, this is

not a problem since they always assign the exact amount of resources needed by the application.

5. **Highly elastic workloads are ideally suited for Serverless (Lithops) due to slow Cluster auto-scaling.** As demonstrated in the autoscaling experiment, both Ray and Dask can smoothly add and remove nodes from the cluster adjusting to the workload requirements. In the experiment we showed how the clusters successfully scale up and down a total of more than 40 nodes both with VMs and with K8s pods. When scaling up, the VMs cluster needs about 120s and the K8s cluster needs about 10s in Ray and 80s in Dask to reach the desired number of workers. Referring to scaling down both VMs and K8s cluster need ≈10s. However, they are not able to match Lithops since, thanks to being serverless, it directly assigns the necessary resources from the start and does not need to invest time in creating new VMs or pods.

6. **Cluster technologies expose more system complexity, but also allow the user to have more control over the infrastructure than serverless technologies.** The user must provide container or VM images with up-to-date and correctly configured software and set up the network among the different components of the application. Instead, serverless functions essentially consist of high-level code with few configuration steps.

7. **Cluster requires more maintenance than Serverless.** Maintenance in cluster technologies includes managing hardware, software, and infrastructure components. It requires regular monitoring, updates, hardware maintenance, and scaling adjustments. However, maintenance in serverless technologies primarily focuses on application code, configuration, and monitoring. In serverless infrastructure maintenance is handled by the cloud provider, including scaling and hardware management.

8. **The most economical technology will depend on the use case.** Looking at AWS billing prices, we can state that Lambda functions are more expensive than VMs ($0.0000667 vs $0.00002678). In our case the cost of 1 second in m6i.large is equivalent to 200 milliseconds in AWS Lambda (2 functions of 1 CPU each one and 4096MB). However, Lambda functions will usually be running for lower periods of time than VMs due to the negligible deployment time. Consequently, we cannot

definitively state that one technology is more expensive than the other, as this depends on each use case. If an uninterrupted cluster is required, it will be much more expensive than simply invoking serverless functions.

9. **The developed hybrid model has proved that both architectures can be used in the same implementation**. Serverless performs better with stateless tasks whereas cluster frameworks are more appropriate for stateful ones. In the selected data processing implementation, the hybrid model outperforms the cluster ones with lower billing price, as it can be observed in the plots.

To conclude, we cannot say one technology is better than the other as each technology has its advantages and disadvantages. Each user should be aware of the strengths and limitations of each model and choose the most appropriate one according to the application characteristics.

7. Future perspectives

In this module, we will provide an elaboration on our aspirations for the future trajectory of the study presented within this document. While we have explored numerous aspects of both technologies, our appetite for understanding remains unsatiated. Our quest for knowledge and insight propels us towards a desire to dig even deeper in some aspects of the study.

Firstly, we have verified that a hybrid model combining Cluster and Serverless can operate effectively. In the future, we aim to demonstrate its applicability in other scenarios, such as in Machine Learning or Deep Learning pipelines.

One interesting approach would be to find a pipeline that utilizes Ray or Dask for distributed deployment and scalability, analyze the various stages of the workflows and their data dependencies, and determine which phases could be executed in a serverless manner. With the use of Lithops, offloading preprocessing, for instance, could be achieved.

On the other hand, we also intend to execute Lithops within a pipeline that requires communication between functions, aiming to showcase that in such scenarios Cluster technologies outperform Serverless alternatives.

8. Conclusions

In this project, an analytical study has been developed concerning two types of technologies: Cluster and Serverless; utilizing three distinct frameworks: Ray, Dask, and Lithops. At the start, we set forth two main objectives: to ascertain the pros and cons of each technology, and to demonstrate the two of them can work together. After several months of dedicated effort, we have successfully achieved these goals, gaining extensive knowledge not only about both technologies and frameworks but also about cloud computing.

This working methodology, presented as a comprehensive analysis of technologies, was new to us and posed a significant challenge. Furthermore, despite starting with very limited knowledge in the field of cloud computing, we have managed to grasp and apply the concepts required for the study, such as the utilization of VMs, K8s, and AWS Lambda.

This thesis has personally served as our introduction to working in a research group focused on the study of distributed systems. CloudLab has provided us with the necessary resources to conduct all the experiments and the opportunity to collaborate on scientific articles.

Regarding the obtained results, a detailed analysis has been conducted showcasing several pots that have been useful to determine the advantages and limitations of each technology. Moreover, it has also been demonstrated that creating a hybrid between both is viable and that it can optimize the performance that was originally present in the cluster.

In conclusion, this undergraduate thesis has been a notable achievement, showcasing the ability to tackle technical challenges and acquire knowledge in previously unfamiliar domains. However, as described in the previous section our intention is to keep working with these technologies in order to contribute with new insights in this field.

References

[1] *Apache Hadoop*. URL: https://hadoop.apache.org/

[2] *Spark*. URL: https://spark.apache.org/

[3] *Ray*. URL: https://www.ray.io/

[4] *Dask*. URL: https://www.dask.org/

[5] *Lithops*. URL: https://lithops-cloud.github.io/

[6] *CloudLab*. URL: https://cloudlab.urv.cat/

[7] *AWS*. URL: https://aws.amazon.com/

[8] *IBM Cloud*. URL: https://www.ibm.com/cloud

[9] *Google Cloud Platform*. URL: https://kubernetes.io/

[10] *AWS EC2*. URL: https://aws.amazon.com/ec2/?nc1=h_ls

[11] *IBM Cloud Servers*. URL: https://www.ibm.com/products/virtual-servers

[12] *Google Compute Engine*. URL: https://cloud.google.com/compute

[13] *Amazon S3*. URL: https://aws.amazon.com/s3/?nc1=h_ls

[14] *IBM Cloud Object Storage*. URL: https://www.ibm.com/cloud/object-storage

[15] *Google Cloud Storage*. URL: https://cloud.google.com/storage?hl=es-419

[16] *AWS Lambda*. URL: https://aws.amazon.com/lambda/?nc1=h_ls

[17] *Google Cloud Functions*. URL: https://cloud.google.com/functions

[18] *Kubernetes*. URL: https://cloud.google.com/?hl=en

[19] *Kubernetes architechture*. URL: https://magmax.org/blog/kubernetes-1-conceptos-basicos/

[20] *Dask EC2Cluster*. URL: https://cloudprovider.dask.org/en/latest/aws.html

[21] *Dask KubeCluster*. URL: https://kubernetes.dask.org/en/latest/operator_kubecluster.html

[22] *Apache Arrow*. URL: https://arrow.apache.org/docs/

[23] *Ray KubeRay*. URL: https://github.com/ray-project/kuberay

[24] *Lambda functions costs*. URL: https://aws.amazon.com/lambda/pricing/?nc1=h_ls

[25] *Anyscale autoscaling code*. URL: https://www.anyscale.com/blog/autoscaling-clusters-with-ray

Appendix

A Baseline files for hybrid

A.1. dask_baseline.py

```
import configparser, contextlib
import os, sys, time, s3fs
from dask_cloudprovider.aws.ec2 import EC2Cluster
from dask.distributed import Client, progress
import dask.array as da
import dask.dataframe as dd
import dask

workers = 64

def get_aws_credentials():
    # Read in your AWS credentials file and convert to environment
variables.
    parser = configparser.RawConfigParser()
    parser.read(os.path.expanduser('~/.aws/config'))
    config = parser.items('default')
    parser.read(os.path.expanduser('~/.aws/credentials'))
    credentials = parser.items('default')
    all_credentials = {key.upper(): value for key, value in [*config,
*credentials]}
    with contextlib.suppress(KeyError):
        all_credentials["AWS_REGION"] = all_credentials.pop("REGION")
    return all_credentials

env_vars = get_aws_credentials()
# The workers will need `s3fs` to read from S3.
env_vars["EXTRA_PIP_PACKAGES"] = "s3fs"

cluster = EC2Cluster(docker_image = "daskdev/dask:2023.4.0-py3.9",
                     security = False,
                     # Set our region here to be the same as our
credentials.
                     region = env_vars["AWS_REGION"],
                     scheduler_instance_type = "m6i.large",
                     worker_instance_type = "m6i.large",
                     bootstrap = True,
                     n_workers = workers,
                     filesystem_size = 40,
                     worker_options={'nthreads':2},
                     # Pass along our environment variables to the workers.
                     env_vars = env_vars
                 )

client = Client(cluster)
print(client.dashboard_link)

# Wait for our workers to be ready for processing.
if(workers > 0):
    client.wait_for_workers(workers)
```

```
st = time.time()

df = dd.read_csv('s3://data_to_ingest/50GB_128/*.csv',
assume_missing=True)
filtered_df = df.loc[df['ss_net_profit'] > 5000]
sorted_df = filtered_df.sort_values('ss_net_profit').compute()

et = time.time()

print(sorted_df)
print("Time spent: ", et-st)

cluster.close()
```

Code 14. Dask only baseline code for hybrid

A.2. ray_baseline.py

```
import ray
import time
import numpy as np
from typing import Dict

def filter(batch: Dict[str, np.ndarray]) -> Dict[str, np.ndarray]:
    condition = batch["ss_net_profit"]>5000
    filtered_batch = {key: value[condition] for key, value in
batch.items()}
    return filtered_batch

ray.init(address='auto')

st = time.time()

df = ray.data.read_csv("s3://data_to_ingest/50GB_128/")
filtered = df.map_batches(filter)
sorted = filtered.sort('ss_net_profit', descending=True)
sorted.materialize()

elapsed = time.time() - st

print('Elapsed: ', elapsed)
```

Code 15. Ray only baseline code for hybrid

UNIVERSITAT ROVIRA i VIRGILI