

# An Empirical Evaluation of the Viability of the Serverless Paradigm for Scientific Workflows

by

Abdallah Elshamy

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Mathematics  
in  
Computer Science

Waterloo, Ontario, Canada, 2023

© Abdallah Elshamy 2023

## **Author's Declaration**

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Statement of Contributions

This work was done in collaboration with Ahmed Alquraan under the guidance of our advisor Prof. Samer Al-Kiswany. The results of this work are included in two publications. The preliminary results have been published [32] at the Workshop on Serverless Systems, Applications and Methodologies (SESAME'23). The complete analysis is under submission to the ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'24). I am the main contributor to this thesis and the first author of the aforementioned papers. Ahmed Alquraan helped with analyzing the results (Chapter 5).

## Abstract

Scientific workflows are typically data-intensive. They consist of many stages, each of which may contain hundreds to even thousands of tasks. Traditionally, scientific workflows have been executed using the serverful computing model. Serverless computing presents an attractive alternative to the serverful computing model as it frees developers from managing and provisioning resources and offers a fine-grained billing model. In this work, we study the viability and evaluate the trade-offs of using the serverless paradigm to run scientific workflows. We follow an empirical approach to evaluate the performance and cost benefits of this paradigm and to study the suitability of the current serverless software stack to support complex data-intensive scientific workflows. Specifically, we discuss, implement, and evaluate three orchestration approaches for executing scientific workflows: serverful-centralized, serverless-centralized, and serverless-decentralized. This work is the first to implement and evaluate a purely serverless orchestration approach that does not require deploying a dedicated workflow manager for scientific workflows.

Our evaluation shows that serverless orchestration approaches cause a noticeable performance overhead for some workflow patterns (e.g., reduce stages) due to accessing a large amount of remote data. We propose two optimizations (i.e., prefetching file privileges and container placement) that exploit data locality to mitigate that impact. Our evaluation with three scientific workflows—Montage, 1000Genomes, and SRA Search—shows that serverless-centralized and serverless-decentralized achieve a comparable performance to a serverful approach. Also, our results show that prefetching file privileges and container placement optimizations improve the performance by 32.6% and 44% respectively when compared to an unoptimized version for the Montage application. We also introduce a cost model to estimate which approach is cheaper for a given application and a cloud provider. Our cost analysis shows that answering this question depends on the characteristics of the workflow and the pricing of the cloud provider.

## **Acknowledgements**

I would like to thank all the little people who made this thesis possible.

## **Dedication**

This work is dedicated to my beloved parents, wonderful brothers, amazing friends, and awesome teachers. From the bottom of my heart, thank you. I am grateful to have all of you in my life.

# Table of Contents

<b>Author's Declaration</b>	<b>ii</b>
<b>Statement of Contributions</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>Dedication</b>	<b>vi</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>List of Abbreviations</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Scientific Workflows . . . . .	5
2.2 Workflow Management . . . . .	7
2.3 Serverless Environments . . . . .	8
2.4 Container Orchestration . . . . .	8

2.4.1	Knative Serving Overview . . . . .	9
2.5	Distributed File Systems . . . . .	10
2.5.1	Ceph File System (CephFS) Design Overview . . . . .	11
2.5.2	CephFS Metadata Path . . . . .	12
2.5.3	CephFS IO Path . . . . .	12
<b>3</b>	<b>Workflow Orchestration Approaches</b>	<b>13</b>
3.1	Serverful Model . . . . .	13
3.2	Serverless Model . . . . .	14
3.2.1	Centralized Orchestration Approach . . . . .	14
3.2.2	Decentralized Orchestration Approach . . . . .	15
3.3	Queuing Paradigms . . . . .	16
<b>4</b>	<b>Methodology</b>	<b>18</b>
4.1	Workflows Selection . . . . .	19
4.2	Experimental Setup . . . . .	22
<b>5</b>	<b>Evaluation</b>	<b>24</b>
5.1	Case Study: Montage . . . . .	24
5.1.1	Comparing Different Orchestration Approaches . . . . .	24
5.1.2	Locality Optimizations . . . . .	28
5.1.3	Effect of the Queuing Approach . . . . .	30
5.1.4	Effect of Cold Starts . . . . .	31
5.2	Case Study: 1000Genomes . . . . .	34
5.2.1	Comparing Different Orchestration Approaches . . . . .	35
5.2.2	Effect of the Number of Parallel Tasks . . . . .	35
5.2.3	Effect of Intermediate Data Compression . . . . .	37
5.2.4	Effect of Cold Starts . . . . .	38

5.2.5	Locality Optimizations . . . . .	40
5.3	Case Study: SRA Search . . . . .	41
5.3.1	Comparing Different Orchestration Approaches . . . . .	41
5.3.2	Locality Optimizations . . . . .	44
5.3.3	Effect of the Order of Execution of the Parallel Tasks . . . . .	45
5.3.4	Effect of Cold Starts . . . . .	47
<b>6</b>	<b>Cost Analysis</b>	<b>48</b>
6.1	Cost Model . . . . .	48
6.2	Serverful-Centralized Vs Serverless-Centralized . . . . .	50
6.2.1	Execution Cost . . . . .	50
6.2.2	An Analysis of the Equivalence Point . . . . .	51
6.2.3	Orchestration Cost . . . . .	53
<b>7</b>	<b>Related Work</b>	<b>54</b>
7.1	Utilizing Serverless Environments . . . . .	54
7.2	Scientific Workflows Execution in Serverless Environments . . . . .	55
7.3	Accelerating Workflows in a Serverless Environment . . . . .	55
7.4	Serverless Workflows Orchestrators . . . . .	57
7.5	Decentralized Orchestration of Serverless Workflows . . . . .	57
<b>8</b>	<b>Discussion and Concluding Remarks</b>	<b>59</b>
8.1	Viability of Using Serverless Computing to Execute Scientific Workflows . . . . .	59
8.2	Characteristics of Scientific Workflows . . . . .	59
8.3	Impact of the Optimizations . . . . .	60
8.4	Implementing the Decentralized Orchestration Approach . . . . .	61
	<b>References</b>	<b>62</b>
	<b>APPENDICES</b>	<b>69</b>

<b>A Dataset of the SRA Search Workflow</b>	<b>70</b>
A.1 Sequence Read Archives (SRA) IDs in the Dataset . . . . .	70
A.2 Selected subset of the dataset . . . . .	74

# List of Figures

2.1	Invocation flows in Knative Serving. . . . .	10
2.2	Architecture of CephFS. . . . .	11
3.1	Different orchestration approaches for scientific workflows . . . . .	15
5.1	The structure of the Montage workflow. . . . .	25
5.2	The effect of the used orchestration approach on the Montage workflow execution time. . . . .	26
5.3	The effect of locality optimizations on the Montage workflow execution time. . . . .	29
5.4	The effect of the used queuing approach on the execution time of parallel stages in the Montage workflow. . . . .	31
5.5	The effect of cold starts on the Montage workflow execution time. . . . .	32
5.6	The effect of using general executors on the Montage workflow execution time. . . . .	33
5.7	The structure of the 1000Genomes workflow. . . . .	34
5.8	The effect of the used orchestration approach on the 1000Genomes workflow execution time. . . . .	36
5.9	The effect of the number of parallel tasks on the 1000Genomes workflow execution time. . . . .	37
5.10	The effect of intermediate data compression on the 1000Genomes workflow execution time. . . . .	39
5.11	The effect of cold starts on the 1000Genomes workflow execution time. . . . .	40
5.12	The structure of the SRA Search workflow. . . . .	42

5.13	CDF of the execution time of the tasks of the parallel stage in the SRA Search workflow. . . . .	42
5.14	CDF of the execution time of the selected tasks of the parallel stage in the SRA Search workflow. . . . .	43
5.15	The effect of the used orchestration approach on the SRA Search workflow execution time. . . . .	44
5.16	The effect of the prefetching file privileges optimization on the SRA Search workflow execution time. . . . .	45
5.17	The effect of the order of execution on the SRA Search workflow execution time. . . . .	46
5.18	The effect of cold starts on the SRA Search workflow execution time. . . . .	47
6.1	The relation between $\alpha$ and the costs of the serverful and the CaaS version of the serverless approaches. The horizontal lines represent the cost of the serverful approach. . . . .	51
6.2	The relation between $\beta$ and the costs of the serverful and the FaaS version of the serverless approaches. The horizontal lines represent the cost of the serverful approach. . . . .	52

# List of Tables

2.1	Common workflow patterns. . . . .	5
4.1	Considered workflows sorted by the number of the stages they have. The highlighted workflows are the ones we use in our study. . . . .	20
6.1	Execution cost for the studied workflows. . . . .	50
6.2	Orchestration cost for the studied workflows. . . . .	53

# List of Abbreviations

**CaaS** Container as a Service [7](#), [8](#), [14](#), [49–51](#), [53](#)

**CephFS** Ceph File System [10–12](#), [22](#), [27](#)

**DAG** Directed Acyclic Graph [1](#), [2](#), [5–7](#), [13–15](#), [18](#), [19](#), [56](#)

**FaaS** Function as a Service [7](#), [8](#), [14](#), [49–53](#)

**IaaS** Infrastructure as a Service [7](#), [8](#), [13](#)

**MDS** Ceph File System Metadata Servers [11](#), [12](#)

**SRA** Sequence Read Archives [2](#), [18](#), [19](#), [41–47](#), [50](#), [61](#), [63](#), [67](#)

**WMS** Workflow Management System [7](#), [55](#)

# Chapter 1

## Introduction

Expressing a science application as a workflow of tasks is a popular development and deployment paradigm [39, 57, 60, 56, 45, 36]. Scientific workflows are composed of multiple execution stages, each stage consist of hundreds to thousands of computational tasks. Those tasks communicate through intermediate files stored on a shared storage service [29, 63, 19, 38]. Scientific workflows are expressed as a [Directed Acyclic Graph \(DAG\)](#), in which nodes represent tasks and edges represent data dependencies.

Scientific workflows are currently executed using a serverful environment such as large on-premises or cloud clusters [36, 61, 37, 41, 38]. In a serverful environment a science application allocates dedicated physical or virtual resources with fixed capacity for the duration of the application. The user deploys the software stack needed for the science workflow, configures the platform, and runs the application. This approach has the following drawbacks. First, high administration effort as the user has to provision, deploy, and maintain the needed resources. Second, resource usage is inefficient as the workflow does not utilize the allocated resources all the time during its execution [43]. For instance, workflow applications often have long-running reduce stage at which one node is used to aggregate intermediate results from other nodes. During this reduce stage the majority of nodes in the cluster are idle. These drawbacks have cost implication as maintaining workflows increases operation cost, and inefficiencies increase running cost.

Recently, the serverless computing model was introduced to address the drawbacks of the serverful model. In serverless computing, developers express their application as functions that are triggered by events. The event can be the arrival of new data or a request from a client. In this model, the cloud provider deploys, maintains, and scales the application while the developer focuses on building application functions. Serverless

offers a fine-grained billing model in which an application is billed only for the time of the function execution. No cost is incurred if the service is not running.

The serverless model may offer a viable alternative to the serverful model for running scientific workflows. Serverless computing can eliminate the administration efforts needed to provision, deploy, and maintain an application. The serverless model allows cloud providers to consolidate applications at a fine granularity, leading to higher platform utilization. Furthermore, the fine-grained billing approach of serverless computing may lower the application cost.

It is not clear if using the serverless paradigm to run scientific workflows will bear the aforementioned benefits. The design and implementation of serverless platforms are tuned to support streaming and web applications. Scientific applications have significantly different characteristics than streaming and web applications. First, scientific workflows are data-intensive. An input data set for a single scientific task can range from few kilobytes to gigabytes [36]. Second, the amount of data transferred between tasks is significantly larger than those transferred in web applications. Some science task may transfer gigabytes of data between stages [36]. Third, task execution time is considerably longer in science applications. While serverless function take 10s of milliseconds [48] to complete a web request, a science task can take tens of minutes to complete [36]. Fourth, scientific workflows are bursty: In a parallel stage, a workflow can launch thousands of parallel tasks [36]. These characteristics can limit the benefits or render it unviable to run scientific applications on serverless platforms.

In this work, we study the viability and evaluate the trade offs of using the serverless paradigm to run scientific workflows. We follow an empirical approach to evaluate the performance and cost benefits of this paradigm and to study the suitability of the current serverless software stack to support complex data-intensive scientific workflows.

We use three scientific workflows: Montage [39], an astronomy application; 1000Genomes [1], a bioinformatics application; and SRA Search [17], a bioinformatics application. We chose these workflows as they have diverse communication patterns and have different workload characteristics in terms of task execution time and input/output data size (Chapter 4). In our evaluation we compare the serverful paradigm to two alternatives of the serverless approach. The two approaches differ in their orchestration approach: centralized and decentralized approaches. A centralized orchestration approach employs a central workflow scheduler that uses the workflow DAG to submit a task when its inputs are available. The decentralized approach does not have a central scheduler, instead it leverages the platform mechanism for issuing triggers that launch the next task in a DAG (Chapter 3).

In our evaluation we aim to answer the following questions: Is the state-of-the-art

serverless software stack able to support complex workflow applications? What are the challenges and tradeoffs of different workflow orchestration approaches? What is the performance and cost impact of different orchestration approaches? Are there easy-to-implement optimization opportunities for running workflows on serverless platforms?

Through our evaluation (Chapter 5) we identify two optimizations to improve data access locality: prefetching file privileges and container placement. Prefetching file privileges is a novel optimization that overlaps the execution of a reduce stage with the parallel stage preceding it in order to acquire the privileges required to access the files needed by the reduce stage as soon as they are written. Container placement optimization targets placing tasks that share a large amount of data on the same node. Our results show that prefetching file privileges and container placement optimizations improve the performance of the unoptimized serverless version by 32.6% and 44%, respectively.

We also introduce a cost model (Chapter 6) to estimate which approach is cheaper for a given application and a cloud provider. Our cost analysis shows that answering this question depends on the characteristics of the workflow and the pricing of the cloud provider.

Our evaluation shows that serverless computing is a viable approach to execute scientific workflows, achieving comparable performance and potential cost savings. It also brings operational benefits as developers focus only on the development of their applications, while the cloud provider is responsible for managing, provisioning, and dynamically scaling resources. However, it comes with challenges due to the characteristics of scientific workflows. First, Scientific workflows show a high variability in its resource demand due to its burstiness. While this burstiness makes autoscaling an attractive feature for scientific workflows on serverless platforms, it can stress the infrastructure. Second, scientific workflow tasks tend to be long, potentially exceeding the current time limits imposed by serverless platforms. Third, scientific workflows share a large amount of intermediate data, which has a notable impact on performance. Fourth, scientific workflows can have a large variance in the execution time of tasks, which complicates taking scaling decisions. Finally, scientific workflows usually use a POSIX-compliant file system interface, which may not be optimized for the patterns and demands of scientific workflows. We discuss these challenges in Chapter 8. We also discuss the impact of the introduced optimizations and the challenges faced when implementing the decentralized orchestration approach.

The rest of the thesis is organized as follows. We present an overview of scientific workflows, serverless environments, and the technologies used in the experiments in Chapter 2. Chapter 3 provides a detailed discussion of different orchestration approaches for executing scientific workflows. Chapter 4 discusses the methodology used in the experiments. Chap-

ter 5 presents our empirical evaluation. Chapter 6 analyzes the execution and orchestration costs associated with the experiments. Chapter 7 discusses the related work. Finally, we discuss our main insights and concluding remarks in Section 8.

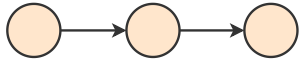
# Chapter 2

## Background

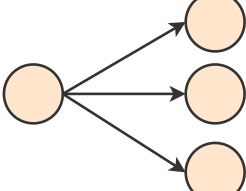
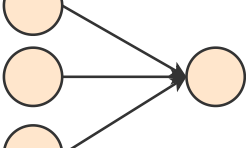
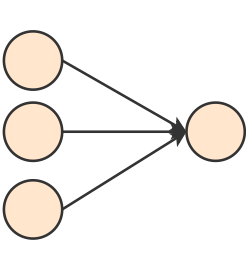
### 2.1 Scientific Workflows

Scientific workflows allow scientists to express complex scientific applications by expressing the data processing steps and the dependencies between these steps. Scientific workflows usually consist of multiple steps, each of which may contain a large number of independent tasks. Typically, a scientific workflow is expressed as a [DAG](#), in which nodes correspond to tasks and edges represent data dependencies. Tasks of different stages communicate through files stored on a shared file system (e.g., CephFS [\[62\]](#)). A task is only executed when all its dependencies are available. Data access patterns define how the output of a task is being used by other tasks. A scientific workflow can consist of different access patterns including pipeline, broadcast, scatter, reduce, and gather patterns. [Table 2.1](#) defines the common patterns. Al-Kiswany et al. [\[19\]](#) discuss these patterns in detail.

Table 2.1: Common workflow patterns.

Pattern	Representation in a <a href="#">DAG</a>
<b>Pipeline:</b> A series of tasks connected sequentially, forming a linear chain. Each task in the pipeline processes data and passes the output to the next.	

Continuation of Table 2.1

Pattern	Representation in a DAG
<p><b>Fanout:</b> A single task triggers multiple parallel tasks. The single task processes data and passes the output, following either a broadcast pattern (entire output) or a scatter pattern (portion of the output), to the parallel tasks.</p>	
<p><b>Reduce:</b> A single task performs a reduction operation on results from several parallel tasks. The single task can reduce data into a single output (reduce pattern) or multiple outputs (allreduce pattern).</p>	
<p><b>Gather:</b> A single task collects results from several parallel tasks. In contrast to the reduce pattern, the single task does not perform reduction operations on the results, focusing solely on collecting and arranging the data. The single task can collect data into a single output (gather pattern) or multiple outputs (allgather pattern).</p>	

Scientific workflows are typically more data- and compute-intensive than other serverless workflows. To demonstrate this, we contrast the characteristics of serverless workflows in a subset of Azure Durable Functions [51] production workloads [48] with the characteristics of popular scientific workflows from different fields [36]. Firstly, the number of tasks in parallel stages is much larger in scientific workflows. For instance, while the maximum number of tasks in a parallel stage in 90% of production serverless workflows is less than 5, this number is typically in the hundreds to hundreds of thousands for scientific workflows. Secondly, the execution time of scientific workflows is much longer than that of production serverless workflows. For instance, while the median execution time of a serverless func-

tion is 670 ms and the median execution time of a complete serverless workflow is 5.6 sec, the execution time of a scientific task ranges from milliseconds to hours, and the entire scientific workflow typically takes hours to complete. Finally, the total size of transferred intermediate data is much larger in scientific workflows. While 85% of the transferred intermediate data in production serverless workflows are of sizes  $\geq 1$  KB and the median is 8 KB, the total size of transferred intermediate data in scientific workflows is typically in the hundreds of gigabytes and can even reach hundreds of terabytes. Those differences suggest that scientific workflows should not be in the same category with other serverless workflows. Specialized studies that target scientific workflows are needed to assess the viability of using serverless infrastructure to execute them.

## 2.2 Workflow Management

A [Workflow Management System \(WMS\)](#) [29, 63, 19] is a tool that is responsible for executing a workflow in different execution environments (e.g., a cluster, a grid, or cloud computing). [WMSs](#) are typically designed to exploit a large amount of computing and storage resources to execute the target workflow in parallel in order to reduce the total execution time. A [WMS](#) receives a workflow [DAG](#) as an input and then it generates an execution plan based on the workflow [DAG](#). The generated execution plan of the tasks must adhere to the data dependencies specified by the workflow [DAG](#). Moreover, a [WMS](#) acts as a scheduler that maps tasks to computing resources. Typically, a [WMS](#) maintains all tasks in a queue. The [WMS](#) keeps track of the dependencies of each task. Once the dependencies of a task are met, the [WMS](#) schedules the task on one of the available nodes. Also, a [WMS](#) is typically responsible for keeping track of the status of all running tasks and handling failures during task execution. Several [WMSs](#) [29, 35, 49, 27] have been introduced to enable executing scientific workflows in various computing environments, including [Infrastructure as a Service \(IaaS\)](#), [Function as a Service \(FaaS\)](#), and [Container as a Service \(CaaS\)](#).

Scientific workflows use intermediate files to pass data between tasks of different stages. Workflows use a shared file system to store these intermediate files. The distributed file system is ideally deployed on the same nodes that run the workflow to increase access locality.

## 2.3 Serverless Environments

Serverless computing is a new paradigm that differs from the prevalent [IaaS](#) paradigm, in which applications rent virtual machines or containers for a duration of time or for online services. In serverless computing, developers focus on the development of their applications, while the cloud provider is responsible for managing and provisioning resources. Serverless architecture offers automatic scalability, which means that service instances are created and removed dynamically. Another important feature of serverless architecture is the fine-grained pay-as-you-go pricing model. In this model, the cloud provider charges customers based on the actual usage of resources during the execution of a function. That is, customers are not charged for the time in which their applications are idle.

Serverless computing is offered in two models: [FaaS](#) and [CaaS](#). [FaaS](#), such as AWS Lambda [24], Google Cloud Functions [33], and Azure Functions [53], is the more popular serverless service. The main execution unit in [FaaS](#) is a function. Developers write their applications as a set of stateless functions. The developers register these functions with the [FaaS](#) platform. These functions can be invoked either by an event or through an API (e.g., an HTTP request). [FaaS](#) has restrictions on the supported programming languages, the operating systems, and versions of the runtime libraries.

[CaaS](#) tackles the aforementioned limitation of [FaaS](#). In [CaaS](#), developers provide a container to run a given function. When a function is triggered, the cloud provider deploys the provided container and runs the function. [CaaS](#) allows developers to control and configure the execution environment, including the operating system and runtime libraries. [CaaS](#) providers use a container orchestration system, such as Kubernetes [42], to handle container deployment, cluster management, and scaling. Examples of [CaaS](#) are AWS Fargate [23], Google Cloud Run [34], and Azure Container Instances [52].

## 2.4 Container Orchestration

Container orchestration systems are at the core of current offerings of serverless computing. Serverless functions are executed in a virtualized environment, such as using containers. Kubernetes [42] is the leading container orchestration framework in the industry. Knative [40] is an open-source platform for building serverless applications on top of Kubernetes. Knative has two key components: Serving and Eventing. The Serving component allows developers to deploy serverless applications and manage their lifecycle. It handles resource management, scheduling, and automatic scaling. The Eventing component allows

developers to define, manage, and consume events, providing a foundation for event-driven applications. Since we only use the Serving component in our evaluation, we present an overview of its operations.

### 2.4.1 Knative Serving Overview

To deploy a serverless application a developer provides the application logic and a configuration. The application logic can be expressed as a function in the FaaS model or a container that has the application software in the CaaS model. The configuration specifies how the platform should manage the application including when and how to automatically scale the application.

Figure 2.1 shows the main components of Knative and the main steps to process an invocation. A client sends an invocation for a serverless application ((1) in Figure 2.1) to the ingress component. If an application has running containers and has capacity to serve a new request, the ingress component forwards the request to the Knative Service (2). The Knative Service forwards the request to one of the deployed application containers (3). If an application does not have any deployed containers or it has deployed containers but does not have capacity to serve a new request, then the application needs to deploy new containers. This scenario is known as the cold start scenario. In this case, the Ingress component forwards the request to the Activator (4). The Activator queues the invocation and submits a request to deploy additional containers to the Autoscaler (5). The Autoscaler takes a scaling decision based on the system and application configurations (6). Once a new container is deployed, the Activator forwards all queued requests to the Knative Service (7).

The Queue-Proxy (Figure 2.1) serves three main purposes. It queues invocations for a container. It controls the concurrency for a container by controlling how many concurrent invocations a container handles. It collects metrics related to the system load and sends these metrics to the Autoscaler. These measurements help the Autoscaler make scaling decisions.

The Autoscaler component (Figure 2.1) automatically scales an application deployment to match the demand. The Autoscaler can scale down an application to zero or a minimum configurable number of containers when there are no client invocations. The Autoscaler can add more containers when the demand increases for an application. The Autoscaler uses the current deployment status, system configuration, and application configuration to make scaling decisions. The current deployment status is collected from the Queue-proxies of an application deployment.

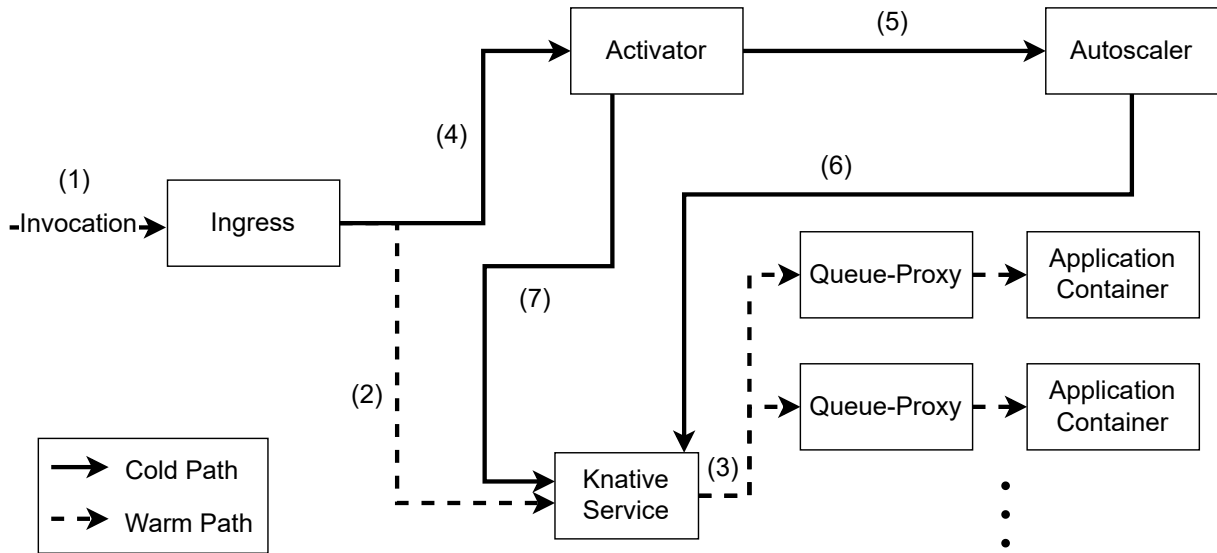


Figure 2.1: Invocation flows in Knative Serving.

## 2.5 Distributed File Systems

Tasks at different stages of a workflow communicate through intermediate files stored on a shared storage system. Scientific workflows usually access those files via a POSIX-compliant file system interface. One task writes its output to a file and this file is the input for one or more tasks at a later stage in a workflow. Juve et al. [38] compared using popular cloud storage services such as object storage and database systems to store intermediate results between workflow tasks. To avoid modifying legacy workflow applications to use a new storage API, this approach writes intermediate files to the node-local file system then stores the file as an object at the cloud object store or database. The node that needs to process a file will read the object from the cloud storage and write it to a local file, then run the task. This additional data transfer to and from cloud storage adds a significant overhead. Juve et al. [38] report that using a distributed file system for sharing intermediate data achieves the highest performance compared to other storage options. In our experiments, we use a distributed file system.

A number of file systems have been used to support workflow applications such as CephFS, LustreFS [11], GlusterFS [8], and NFS. In our experiments, we use CephFS. CephFS is among the most popular file systems used at scientific platforms [19, 58, 59, 54] and is readily supported by Kubernetes.

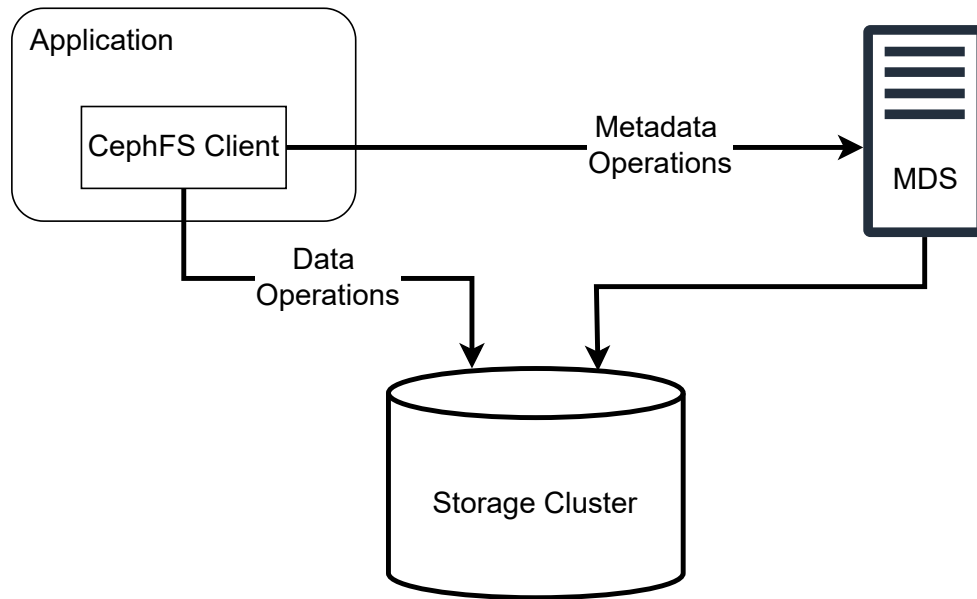


Figure 2.2: Architecture of [CephFS](#).

[CephFS](#) is a distributed storage system that offers a POSIX-compliant file system interface. [CephFS](#) is designed to provide a scalable and high-performance storage solution. [CephFS](#)'s focus on high availability and performance made it popular at High-Performance Computing (HPC) platforms including systems deployed as a scratch space and shared storage for workflows [5].

### 2.5.1 [CephFS](#) Design Overview

Figure 2.2 shows the system architecture of [CephFS](#). [CephFS](#) has three main components: [CephFS](#) storage cluster, [Ceph File System Metadata Servers \(MDS\)](#), and the client-side file system. The [CephFS](#) storage cluster is an object storage system. The storage cluster is used to store files' data and metadata as objects. The [MDS](#) processes file system metadata operations. [CephFS](#) can have multiple [MDS](#) servers each serving a subtree of the file system hierarchy. The file system is a POSIX-compliant file system mounted on the client node. It translates file system calls to [CephFS](#) metadata and data operations. Clients submit metadata operations to the [MDS](#) and access file data objects directly at the storage cluster. This improves performance as the [MDS](#) is not on the data path between the client and the storage cluster.

## 2.5.2 CephFS Metadata Path

The **MDS** maintains the file system metadata and serves metadata operations. The metadata is stored as objects in the storage cluster. Metadata operations are served in two ways. Clients can send their metadata operations to an **MDS** server. The **MDS** server processes the request, updates the corresponding metadata object if needed, and responds to the client. Alternatively, a client can request privileged access to a file metadata. If the **MDS** grants a client privileged access to a file metadata, a client updates a file metadata locally without consulting the **MDS** server. When a privilege is revoked, the client flushes metadata updates to the **MDS** server. A privilege, a.k.a. capability in **CephFS** terminology, allows a client to perform operations on a file without consulting the **MDS** server. Capabilities are granted per file for a specific operation such as read/write, data caching, and metadata caching. If another client requests access to a file that its capabilities are granted to another client, the **MDS** will revoke the capability from the first client. The first client returns the capability, along with a modified version of the inode's metadata if changes were made during its possession of the capability. Then, the **MDS** grants the capabilities to the new client. The **MDS** assures that only one client has the capabilities for a given file at a time to guarantee file and metadata consistency.

## 2.5.3 CephFS IO Path

File data is stored as objects at the storage cluster. Large files are split into multiple objects. To perform a read or a write operation, a client acquires **file read/write** capabilities from the **MDS** for the inodes associated with the file. Once a client acquires the **file read/write** capabilities the client can access the file object directly in the storage cluster. If a single client accesses a file, the **MDS** grants two additional capabilities: **file cache** and **file buffer**. **file cache** capability guarantees that the cached data does not change on the storage cluster and can be used to fulfill local read requests by the client. **file buffer** capability allows a client to buffer writes in a local buffer. Both capabilities improve performance by serving requests locally instead of remotely from the storage cluster.

# Chapter 3

## Workflow Orchestration Approaches

In this chapter, we discuss the computing models for executing scientific workflows and the orchestration approaches used in each model. First, we discuss the serverful model which is the traditional model used to execute scientific workflows. Then, we discuss the serverless model and detail two approaches for orchestrating workflows: centralized and decentralized. Figure 3.1 shows the computing models. A distributed file system is deployed on the same nodes that run the workflow to store the intermediate files.

### 3.1 Serverful Model

Serverful Model (e.g., [IaaS](#)), represents the traditional execution model in which a set of computing nodes (i.e., workers) are responsible for executing the tasks of a workflow. In this model, the orchestration approach is centralized, in which a workflow manager is deployed on one node and responsible for orchestrating the execution of workflow tasks as shown in Figure 3.1. This approach uses a fixed allocation approach in which the allocated resources are reserved for the entire duration of a [DAG](#) execution even if, at times, a subset of resources set idle. The workflow manager receives a workflow [DAG](#) as an input and is responsible for executing the tasks of the [DAG](#) on worker nodes while respecting the data dependencies between tasks.

The main advantage of this approach is that the workflow manager has full knowledge of the status of each worker node (e.g., its hardware resources and the number of assigned tasks) and tasks dependency in a [DAG](#). Hence, the workflow manager can make informed scheduling decisions when assigning tasks to nodes, reducing the end-to-end execution

time. The main drawbacks of this approach are that the developers are responsible for the management and provisioning of the computing and storage resources. This results in high administration efforts. Also, allocated resources may not be fully utilized during the entire execution of a workflow. This results in inefficient use of allocated resources. These drawbacks have cost implications as maintaining workflows increases operation costs, and inefficiencies increase running costs.

## 3.2 Serverless Model

Workflow orchestration in the serverless model has three main differences from the orchestration in the serverful model. First, the serverless model does not use a fixed allocation of resources. The platform dynamically scales up and down computing resources for an application based on demand. Second, the workflow manager in the serverful model tracks both the status of servers in the cluster and the task dependency. In serverless, the workflow manager is not aware of the cluster resources and is not aware of the location of running tasks. The workflow manager only tracks task dependency and invokes tasks when their input data is available. Third, using a serverless approach, the developer specifies how many concurrent requests a container can handle and the maximum number of containers to spin. These parameters are not intuitive to set and impact application performance and cost. We discuss the tradeoffs of these parameters in Chapter 5.

A developer provides the application logic as a [DAG](#) of tasks. The tasks are implemented in functions in the case of [FaaS](#) and containers in the case of [CaaS](#). We follow a [CaaS](#) model because it is more flexible in deploying custom application stacks for science applications.

In the following, we discuss the centralized and decentralized approaches for orchestrating workflow applications.

### 3.2.1 Centralized Orchestration Approach

As shown in Figure 3.1, workflow tasks are orchestrated by a centralized workflow manager. The workflow manager is responsible for sending requests to execute the workflow tasks, checking the responses of these requests, and triggering subsequent tasks while respecting data dependencies between tasks. The main issue with this approach is that the workflow manager has to be deployed on dedicated resources, such as a dedicated VM, as it is a

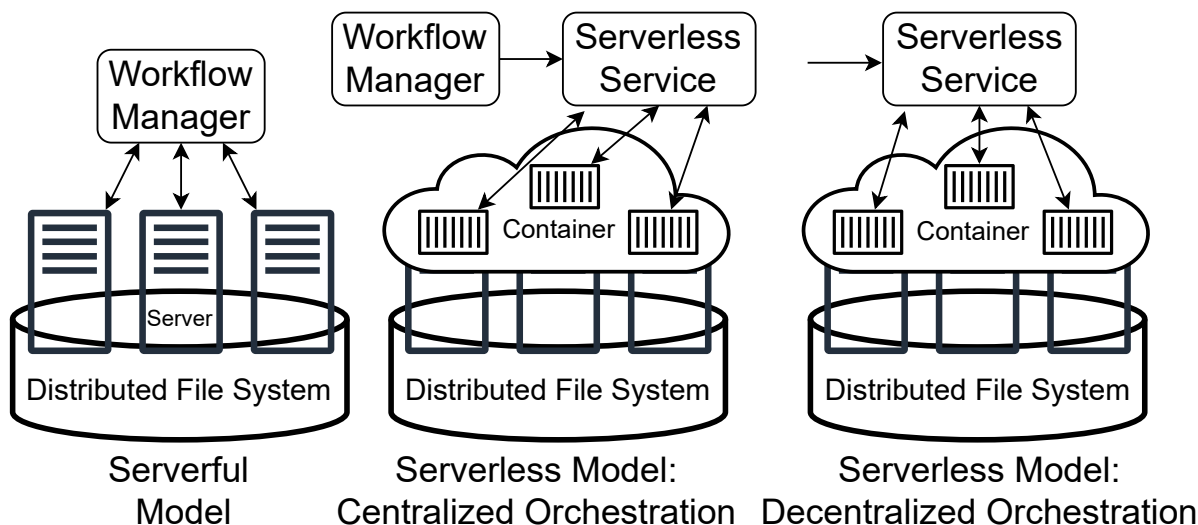


Figure 3.1: Different orchestration approaches for scientific workflows

stateful, long-running process that has to run until the workflow execution is over. This increases the cost of running a workflow application.

While state-of-the-art serverless providers offer a rudimentary support for stitching serverless functions in an application such as AWS Step Functions [25] and Azure Durable Functions [51]. These offerings cannot support complex workflow patterns present in scientific workflows. We discuss the shortcomings of current approaches for stitching serverless functions in Chapter 8. To adequately support complex scientific workflows, we opt to port the serverful workflow manager to use the serverless paradigm. This approach offers the needed flexibility and enables optimizing the execution of scientific workflows.

### 3.2.2 Decentralized Orchestration Approach

In this approach, the logic of the workflow manager is embedded into the workflow stages. A task of a stage triggers one or multiple tasks in the following stage by sending an invocation to the serverless service (Figure 3.1). The main advantage of this approach is that it eliminates the use of a central workflow manager. This reduces the cost of the deployment.

The main challenge of this approach is the need to translate the data dependency expressed in a DAG to triggers between tasks. This translation requires a considerable

development effort. Workflows include communication patterns such as reduce, gather, allreduce, and allgather that are not triggered by a single task, but they are triggered when a group of tasks from a previous stage completes execution. Expressing such data dependencies in serverless triggers is complicated. For example, implementing reduce stages is challenging. A reduce stage should start after all parallel tasks in the previous stage finish. However, it is hard to know when the last parallel task finishes without using a dedicated workflow manager.

To implement a reduce stage we modify the parallel tasks in the stage before the reduce stage such that every parallel task checks if all output files exist. If all output files exist, the task will trigger the reduce stage. Multiple parallel tasks may attempt to trigger the reduce task. To guarantee that the reduce task is triggered once, we use the file system to synchronize multiple parallel tasks, for instance through using an atomic operation (e.g., renaming a file) before triggering the reduce stage. The task that successfully changes the name of a dummy file will trigger the reduce stage. The main advantages of this approach are that the number of trials to trigger the reduce stage is limited by the number of parallel tasks, and the reduce stage is not affected by failures of parallel tasks. However, this approach requires the use of a strongly consistent distributed file system that supports atomic operations. We use this approach in our evaluation.

A different approach we considered for implementing the reduce stage is to use a stage manager task. The stage manager task is triggered when the first parallel task completes. The stage manager task checks if the output files of all tasks of the parallel stage exist. If all files exist, it triggers the reduce stage. Else, it sends a request to itself to keep spinning. We avoided this approach due to its significant drawbacks. The stage manager task runs for the duration of the parallel stage. Due to execution time variance among the parallel tasks this can be a long time. The stage manager task continuously performs file system operations to check for the output files, this significantly increases the file system load and impacts the performance of the parallel tasks. Furthermore, if a parallel task fails, the stage manager task will not be aware of this failure and it will keep spinning forever.

### 3.3 Queuing Paradigms

Scientific workflows have a large number of tasks. In most cases, there will not be enough resources to concurrently run all tasks, and some tasks will be queued. Tasks can be queued at a central queue (centralized queuing) or distributed across multiple queues (distributed queuing) with a queue on each worker node. A study [30] that compared both approaches found that centralized queuing performs better than distributed queuing. It is also easier

to configure and handles well workloads with skewed execution time of tasks. However, it does not scale well. If the number of requests is too large, the central queue might be overwhelmed. The distributed queuing approach does not suffer from such effects. However, it is difficult to fine tune its configuration. Moreover, if there is a large skew in the execution time of tasks, long tasks can be queued at a single container, increasing the end-to-end workflow execution time. We evaluate the effect of both approaches on the performance of the parallel stages in Chapter 5.

In the serverful model, the workflow manager has full knowledge of the status of each worker node. Hence, it can centrally queue the requests and dispatch them efficiently. Queuing could also be easily distributed across worker nodes as resources are fixed.

In the serverless model, Centralized queuing can efficiently happen at the serverless platform as it is aware of the cluster resources. Distributed queuing is harder in the serverless model as the resources are not fixed. The serverless platform will spin containers to handle the load based on the number of concurrent tasks per container configured by the user. Determining this value is not trivial. A simple approach is setting this value to one or a small number. This is not efficient for scientific workflows as it may result in spinning a large number of containers that contend over limited resources. This may result in overwhelming the cluster increasing the end-to-end execution time. Setting this value to a very high number is inefficient as well. This may result in spinning a few containers that do not exploit the available resources in the cluster increasing the end-to-end execution time.

# Chapter 4

## Methodology

We selected three diverse applications to conduct our study. These applications are Montage [39], an astronomy application; 1000Genomes [1], a bioinformatics application; and SRA Search [17], a bioinformatics application. We chose these diverse workflows as they have diverse communication patterns and have different workload characteristics in terms of task execution time and input/output data size.

Then, we port the selected workflows to the three orchestration approaches we study. The porting process involved two components. The first component is the workflow code. We use the workflow code without any change to its structure or storage API. For the serverless approaches, we package the code and its dependencies in containers. For the serverful approach, we configure each node in the cluster with the needed dependencies. The second component is the workflow manager code. For the serverful- and serverless-centralized approaches, we implement the workflow DAG using the API of the respective workflow manager. For the serverless-decentralized approach, we extend the used container with the logic of the workflow manager.

We use the ported workflows to evaluate the studied orchestration approaches (Chapter 5). We run a series of experiments for each workflow. Each experiment aims to evaluate the effect of a certain factor on performance. Some of these factors are the orchestration approach, data locality optimization, and cold starts. We also analyze the costs associated with the orchestration approaches (Chapter 6). We suggest a cost model to facilitate and generalize the analysis. We use the cost model to do a cost comparison between the serverful and serverless approaches.

In the rest of this chapter, we discuss our methodology for selecting the three applications (Section 4.1) and we describe our experimental setup in Section 4.2.

## 4.1 Workflows Selection

Our goal is to select applications with diverse workflow complexity and characteristics. We started by collecting a list of workflows that were studied and characterized previously as this facilitates our work. Table 4.1 contains this initial list. We studied the DAG of every considered workflow. From the DAG, we identified the used patterns (Table 2.1), the number of stages, and any unique features of the workflow. The table also shows if the source code is released as open source.

From this list we selected workflows with the following criteria:

- The code of the workflow is open-source
- Are diverse in their complexity and access patterns to represent a spectrum of applications.
- Can be understood by a non-specialist so it can be modified easily.
- Are diverse in their workload characteristics in terms of data size and execution time.

These criteria led to the selection of Montage [39], an astronomy application; 1000Genomes [1], a bioinformatics application; and SRA Search [17], a bioinformatics application. These workflows are open-source and can be understood by a non-specialist. Hence, it is feasible to port them to the studied orchestration approaches. They also have diverse communication patterns and unique features. Montage [39] has a large number of different types of stages. 1000Genomes [1] can be configured to control the length and number of parallel tasks. SRA Search [17] exhibits a large variation in the execution time of tasks of the parallel stage.

Table 4.1: Considered workflows sorted by the number of the stages they have. The highlighted workflows are the ones we use in our study.

Begin of Table					
Workflow	Patterns	Open-source	#Stages	Unique	Features
SoyKB [27]	pipeline, fanout, and reduce	✗	15		A stage triggers multiple stages
Sipht [36]	pipeline, fanout, and reduce	✗	13		A stage triggers multiple stages
Montage [39]	pipeline, fanout, and reduce	✓	9		Large number of different types of stages
LIGO Inspiral Analysis [36]	pipeline, fanout, and reduce	✓	9		A stage triggers multiple stages
AlphaFold [2]	pipeline, fanout, and reduce	✓	8		A stage triggers multiple stages, A stage reduces multiple stages
Epigenomics [36]	pipeline, fanout, and reduce	✓	8		
Geospatial Workflow [21]	pipeline, fanout, and reduce	✓	8		
Galaxy Classification [7]	pipeline, fanout, and reduce	✓	8		A stage reduces multiple stages
Lung Instance Segmentation [10]	pipeline, fanout, and reduce	✓	7		A stage reduces multiple stages
Variant Calling [57]	pipeline and reduce	✓	7		A stage reduces multiple stages
Cycles [6]	pipeline, fanout, and reduce	✗	7		A stage triggers multiple stages, A stage reduces multiple stages

Continuation of Table 4.1

Name	Patterns	Open-source	#Stages	Special Features
1000Genomes [1]	fanout and reduce	✓	5	A stage triggers multiple stages, A stage reduces multiple stages, can control the length and number of parallel tasks
CyberShake [36]	pipeline, fanout, and reduce	✓	5	
Casa Wind [4]	fanout and reduce	✓	5	A stage triggers multiple stages, A stage reduces multiple stages
Ellipsoids [27]	fanout and reduce	✓	5	
Orcasound [13]	fanout and reduce	✓	4	A stage triggers multiple stages
Page Imputation [12]	pipeline and fanout	✓	4	
SRA Search [17]	fanout and reduce	✓	3	Exhibits a large variation in the execution time of the tasks of the parallel stage
AutoDock Vina [27]	pipeline	✓	3	
modFTDock [19]	pipeline and reduce	✓	3	
Seismic Cross Correlation [15]	reduce	✓	2	
BLAST [19]		✓	1	
KINC [27]		✓	1	

End of Table

## 4.2 Experimental Setup

**Testbed.** We conducted our experiments using 11 CloudLab [31] nodes at the Wisconsin data center. Each node has two Intel Xeon Silver 10-core CPUs and 196 GB of RAM. We use Knative [40] to implement a serverless environment. We use Knative version 1.8.0 and Kubernetes version 1.24.6. We do not limit the CPU and RAM a container can use in our experiments. We use one node to run the orchestration logic and 10 nodes to run workflow tasks. For serverless alternatives, unless otherwise specified, we run a single container per node and configure it to take up to 20 concurrent requests.

**Distributed File system.** We use a distributed file system as a storage service to share files between worker nodes. The file system we use is CephFS [62], the Pacific release (version 16.2.1). We configure CephFS to use RAM disks to ensure that the results we obtain are not affected by the performance of persistent disks. Each worker node has a 50 GB RAM disk.

**Alternatives.** We compare the following orchestration approaches:

- **Serverful-Centralized:** This alternative represents the execution of the workflow in a serverful environment using a centralized workflow manager. We use PyFlow [19] to implement the workflow manager. PyFlow is a workflow management system that supports the parallel execution of workflow tasks on worker nodes while respecting the dependencies between different tasks as specified by the workflow DAG. PyFlow offers a simple representation of how workflow DAGs are currently executed in a serverful environment. PyFlow workflow manager runs on one node.
- **Serverless-Centralized:** This alternative uses a centralized workflow manager to orchestrate the execution of workflow tasks in the serverless environment. One node runs the workflow manager, while the rest of the nodes are used to run containers that execute the tasks. We implemented a simple workflow manager using Python.
- **Serverless-Decentralized:** In this approach, we modify the workflow tasks to integrate the functionality of the workflow manager within the tasks themselves. That is, a task will trigger the next tasks in a workflow. Only 10 worker nodes are used in this alternative to ensure that all alternatives use the same amount of computing resources. We implement this approach by modifying the application code of the Serverless-Centralized alternative in order for each task to invoke the next task in the DAG or implement the logic for reduce operation.

- **Sequential:** This alternative implements the workflow as sequential steps without any parallelism. In this alternative, the workflow is executed on one node. Also, it does not use a distributed file system rather it uses the local RAM disk. This alternative represents a baseline for the other alternatives.

**Queuing Approaches.** We implement the centralized queuing approach in Knative by limiting the number of containers we can spin and setting a concurrency limit that equals the number of requests a container can handle concurrently (20 concurrent requests) without queuing. This forces the requests to be queued at the serverless platform since it cannot spin a new container or send the request to the existing containers. For the distributed queuing approach, we set the concurrency limit equal to the desired number of requests that we want to be queued at a container. Then, Knative will spin enough containers to handle the requests. As centralized queuing is easier to configure, we use it in all experiments unless otherwise specified.

**Cold Starts.** For the cold start experiments, we store the containers' images on the nodes. We clear the operating system cache before each experiment to ensure that we always hit a cold start. For the warm start experiments, we set the minimum and the maximum number of containers to the same number. This prevents creating new containers while keeping a number of warm containers, avoiding any cold starts. We use warm start in all experiments unless otherwise specified.

# Chapter 5

## Evaluation

In this chapter, we evaluate the studied orchestration approaches using three scientific applications: Montage (Section 5.1), 1000Genomes (Section 5.2), and SRA Search (Section 5.3).

### 5.1 Case Study: Montage

We evaluate different orchestration approaches by comparing the end-to-end and per-stage execution time of Montage [39] (Section 5.1.1). We identify optimization opportunities and evaluate their benefits in Section 5.1.2. We evaluate the impact of different queuing approaches in Section 5.1.3 and cold starts in Section 5.1.4.

**Workflow.** Montage [39] is an open-source astronomy workflow. Figure 5.1 shows the Montage workflow. Montage is a complex workflow with nine stages and different patterns, such as pipeline, fanout, and reduce. We use the "2MASS J" dataset with the following parameters: size 4 and location "M 101". This creates a workflow with 4034 tasks, which include 713 parallel `mProjectPP` tasks, 2602 parallel `mDiffFit` tasks, and 713 parallel `mBackground` tasks. We report the average of 15 trials for each experiment. The standard deviation of the end-to-end execution time is less than 9% for serverful and warm start experiments and 15% for cold start experiments.

#### 5.1.1 Comparing Different Orchestration Approaches

We start by comparing the performance of the Montage workflow using different orchestration approaches. Figure 5.2 shows the average execution time of each stage and the

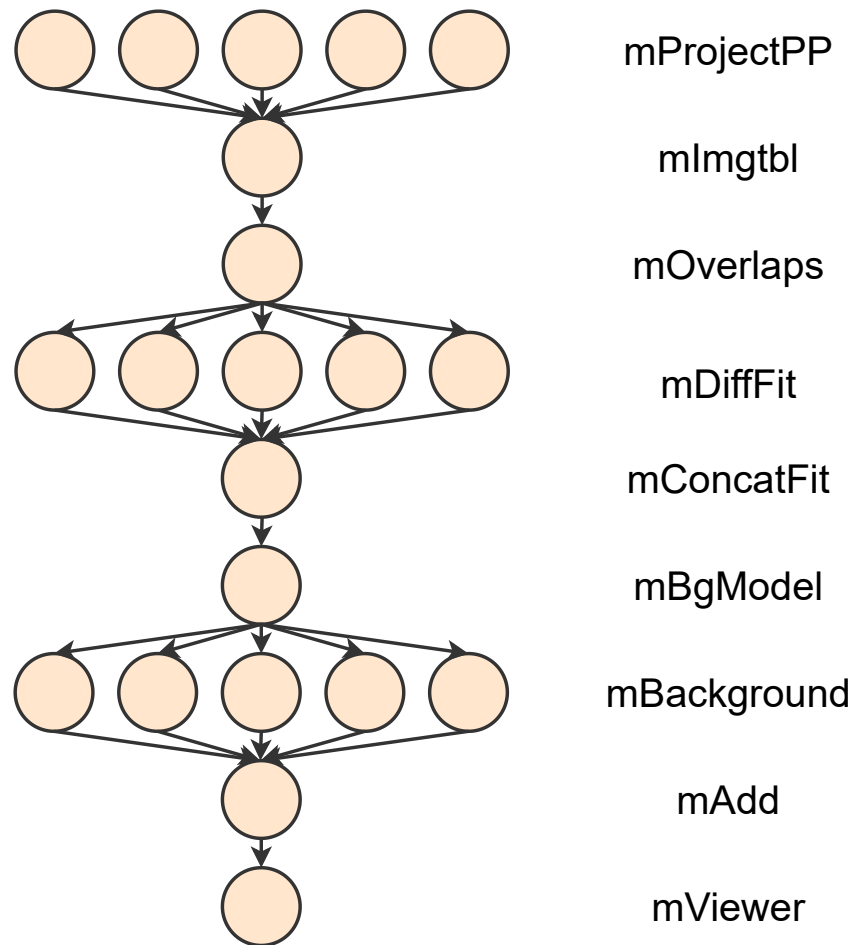


Figure 5.1: The structure of the Montage workflow.

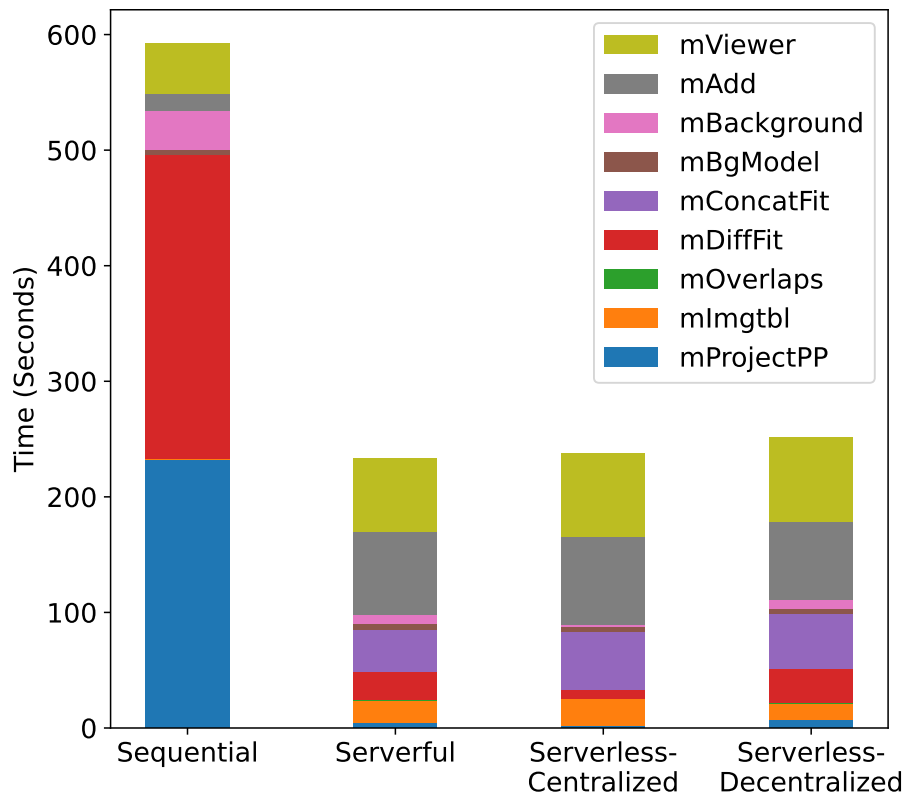


Figure 5.2: The effect of the used orchestration approach on the Montage workflow execution time.

end-to-end execution time of different alternatives. Results show that parallel alternatives (i.e., serverful-centralized, serverless-centralized, and serverless-decentralized) improve performance by at least 57% compared to the sequential alternative. We note that the parallel stages achieve faster execution time compared to the sequential alternative while reduce stages are slower. We investigate this observation in the following analysis. Results show that serverless-centralized and serverless-decentralized alternatives achieve a comparable performance to the serverful-centralized alternative.

**Overhead of the orchestration approach.** We notice that in the serverless-centralized alternative, the execution time of parallel stages is lower than that of the serverful-centralized alternative. For instance, the execution time of the `mProjectPP` stage using the serverless-

centralized and the serverful-centralized alternatives is 2.26 seconds and 4.89 seconds, respectively. This performance difference is caused by the difference in the behaviour of the workflow manager in these alternatives. In the serverful-centralized alternative, the workflow manager is responsible for monitoring the worker nodes to keep track of the number of tasks that each worker node is handling in order to efficiently distribute the tasks among worker nodes, which adds overhead to parallel stages. While in the serverless-centralized alternative, the workflow manager is only responsible for communicating with the serverless platform by sending requests to execute tasks and receiving responses from these tasks.

We notice a significant slowdown in the parallel stages for the serverless-decentralized alternative compared to the serverless-centralized alternative. For instance, for `mProjectPP`, `mDiffFit`, and `mBackground` stages, the serverless-decentralized alternative execution time is higher by 3, 3.9, and 5.5 times, respectively, when compared to the serverless-centralized alternative. This slowdown is due to the overhead resulting from integrating the workflow manager within the functionality of the parallel tasks. Each parallel task, after finishing its original functionality, communicates with `CephFS` to check the existence of the output files of all parallel tasks. This mechanism results in sending 713, 2602, and 713 metadata requests to `CephFS` from the parallel stages `mProjectPP`, `mDiffFit`, and `mBackground`, respectively, causing a noticeable performance overhead. We note that this overhead increases the end-to-end execution time by only 5.29% as the workflow is dominated by long sequential stages. However, for workflows that are dominated by parallel stages, the impact on the end-to-end execution time is expected to be more noticeable.

**Effect of distributed file systems.** Figure 5.2 shows that serverful and serverless alternatives suffer from large execution time in the reduce stages and stages that process large files when compared to the same stages in the sequential alternative. For instance, for `mAdd`, and `mImgtbl` reduce stages, the serverless-decentralized alternative execution time is higher by 4.5 and 17.8 times, respectively, when compared to the sequential alternative. This performance degradation is due to optimizations that are typically implemented in distributed file systems; `CephFS` utilizes file system capabilities (i.e., privileges) to optimize write and read operations. If a file is being written by only one node, `CephFS` grants write and buffer capabilities exclusively to this writer node. These capabilities allow the writer node to write data to its local buffer, which is cached and used to serve reads coming from the same node. However, if another node tries to access (e.g., read or write) the same file, `CephFS` revokes the granted capabilities from the writer node to force it to flush its buffer to storage nodes. After that, `CephFS` grants capabilities to the other node to allow it to access the file. `CephFS` uses this approach to guarantee data consistency between different nodes. Stages that read many input files (e.g., reduce stages) have to wait till the data written by all parallel tasks of the previous stage is flushed to storage nodes. Also, stages

that read large files (e.g., `mViewer`) suffer from the same issue as large files are typically partitioned into chunks that are stored on different storage nodes. The sequential alternative does not use a distributed storage system and all files are stored in a local memory avoiding these overheads.

### 5.1.2 Locality Optimizations

In this section, we present and evaluate two optimizations that can mitigate the effects of using a distributed file system in both serverless-centralized and serverless-decentralized alternatives:

**Prefetching file privileges.** This optimization prefetches file privileges at the reduce task. To achieve this, we added a new task that only opens a given file. Opening a file forces the previous node that holds the privileges of the file to flush its buffer and for the new node to acquire file privileges. We schedule the new task on the container that will run the reduce task. This approach forces the nodes that run the parallel tasks to flush their buffers and results in acquiring the needed privileges for the output files at the reducer node. Note that this optimization overlaps the process of acquiring file privileges by the reducer with parallel tasks that are still running.

**Container placement.** In this optimization, containers of tasks that share a large amount of data are placed on the same node (e.g., a parallel stage and the following reduce stage). This optimization improves the performance significantly as it reduces the amount of data that is accessed remotely by tasks; Tasks write output data to local buffers and subsequent tasks read input data from these buffers. Although this optimization has the potential to improve performance, there are multiple challenges with using it to optimize reduce patterns. First, placing all parallel tasks of a stage on the same node can reduce resource utilization and increase the execution time of that stage as all tasks will compete over the same computing resources. Second, this optimization has to be implemented by the serverless platform as the workflow manager does not control container placement on nodes. This adds restrictions to the serverless platform provider process for placing containers in its platform and can limit its adoption.

We study the effect of each of these optimizations on performance. Figure 5.3 shows the end-to-end execution time and the execution time of each stage of the Montage workflow using the two optimizations. Results show that these optimizations reduce the end-to-end execution time. Specifically, compared to the unoptimized serverless alternative, prefetching file privileges and container placement optimizations reduce the end-to-end execution time of the unoptimized version by 32.6% and 44%, respectively. The optimizations also

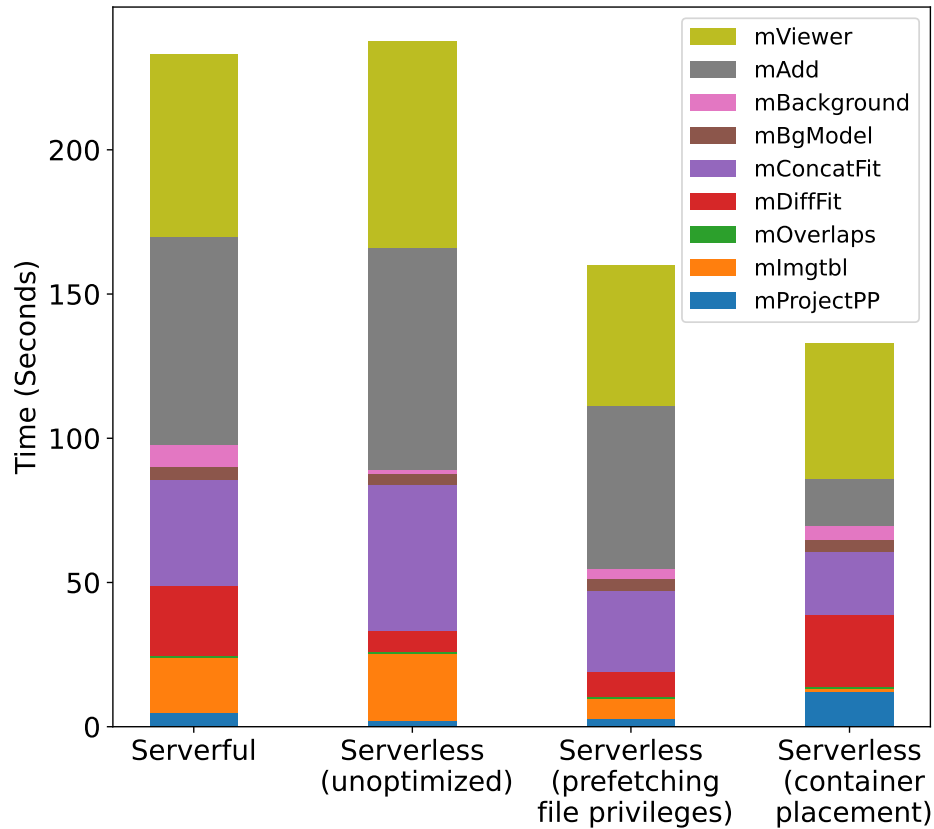


Figure 5.3: The effect of locality optimizations on the Montage workflow execution time.

reduce the variation in the end-to-end execution time with a standard deviation of less than 2.5%. Furthermore, container placement optimization reduces the execution time of reduce stages significantly when compared to the unoptimized version (e.g., `mImgtbl` by 95.8%, `mConcatFit` by 56.8%, and `mAdd` by 78.88%). However, it increases the execution time of parallel stages `mProjectPP`, `mDiffFit`, and `mBackground` by 5.33, 3.38, and 3.68 times, respectively. This performance reduction is due to resource contention as all parallel tasks are executed on one node.

### 5.1.3 Effect of the Queuing Approach

In this section, we evaluate the effects of queuing approaches on performance. Figure 5.4 shows the execution time of the three parallel stages of Montage: `mProjectPP` with 713 tasks, `mDiffFit` with 2602 tasks, and `mBackground` with 713 tasks, with central queuing and distributed queuing. We run this experiment with two configurations: cold start and warm start. Figure 5.4 shows the results with cold start configuration. The configuration with warm starts shows similar results. The following section evaluates the impact of cold start in detail. For distributed queuing, we evaluate the performance with different numbers of concurrent requests per container.

Figure 5.4 shows that centralized queuing achieves a comparable or slightly better performance to the best configuration for distributed queuing. Results show that a small number of concurrent requests per container such as 25 requests increases the execution time of the parallel stages `mProjectPP`, `mDiffFit`, and `mBackground` by 12.93%, 34.2%, and 67.6%, respectively when compared to the best configuration with 100 concurrent requests (Figure 5.4). This performance deterioration is due to spinning a large number of containers. With 25 requests, the platform spins 41 containers for `mProjectPP` and `mBackground` stages, and 141 containers for `mDiffFit`), while with 100 requests the platform spins 10 containers for `mProjectPP` and `mBackground` stages, and 36 containers for `mDiffFit`). Having 4 times higher container numbers creates contention on the nodes' limited resources and impacts performance.

Also, results show that increasing the container concurrency to a large value (150 concurrent requests) results in provisioning few containers and does not utilize the elasticity offered by the serverless platform resulting in a longer execution time for parallel stages. Since parallel tasks in the Montage workflow are short, allowing a container to handle multiple concurrent tasks takes less time than creating new containers. Hence, results suggest that there is a sweet configuration spot that results in lower execution time and better resource utilization.

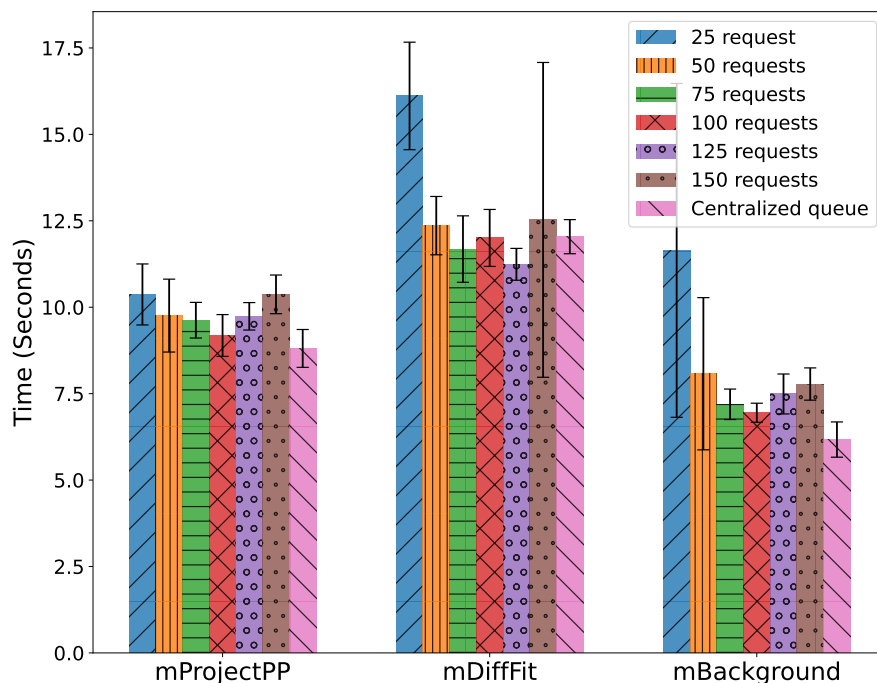


Figure 5.4: The effect of the used queuing approach on the execution time of parallel stages in the Montage workflow.

### 5.1.4 Effect of Cold Starts

We evaluate the effect of cold starts of containers on the performance. Cold starts is a known challenge in modern serverless platforms. The impact of cold starts on scientific workflows is not clear because workflows are 'bursty' and can spin hundreds of containers that may exacerbate the impact of cold starts, but also workflow tasks can take a long time to complete, which may diminish the impact of cold starts. Figure 5.5 shows the average end-to-end execution time and the execution time of each stage of the Montage workflow under cold start and warm start containers. To keep containers warm, we configure the minimum number of application containers to equal the number of containers needed to run the application. We configure the maximum number of containers to equal the minimum number of containers to prohibit the platform from creating excess containers. For the cold start version, we store the containers' images on the nodes. We clear the operating system cache before each experiment to ensure that we always hit a cold start. Figure 5.5 shows that the cold starts increases the total end-to-end execution time by 2.7%. We notice that

the effect of cold starts on some of the sequential and reduce stages is non-noticeable (e.g., `mAdd` and `mViewer`). This is because these stages run for a long time which diminishes the effect of cold starts. However, for the parallel stages, we notice a significant slowdown (e.g., 3.8 times higher for the `mProjectPP` stage). This is due to cascaded cold starts as we hit a cold start with every new container being created.

**Effect of general executors.** Knative uses a container to execute tasks. A developer can provide a single container that implements all tasks of a workflow or can provide multiple containers, with each implementing a subset of the workflow tasks. Each approach has some benefits and drawbacks. On one hand, with general executors, tasks suffer fewer cold starts since containers from previous stages can be reused for the current stage. Moreover, tasks executed within the same container can leverage shared cache, leading to faster reads for tasks passing data between each other. On the other hand, using a general executor

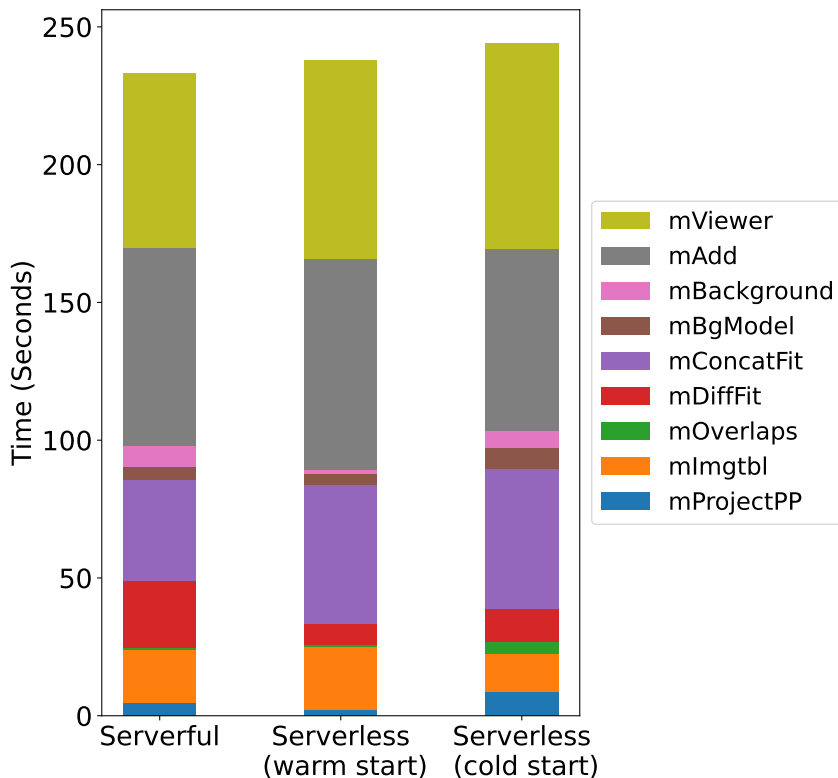


Figure 5.5: The effect of cold starts on the Montage workflow execution time.

impedes the implementation of optimizations for special stages. For instance, to optimize the data movement for a reduce task, we submit a task to pre-fetch file capabilities on a container before running a reduce task (Section 5.1.2). If a general container is used, the workflow manager does not know which container will run the reduce task, and file capabilities will not be pre-fetched to the right container. Figure 5.6 shows the end-to-end execution time and the execution time of each stage of the Montage workflow using general and special executors under warm and cold starts. The results show that there is no large performance difference between using general and special executors. We use special executors in all experiments.

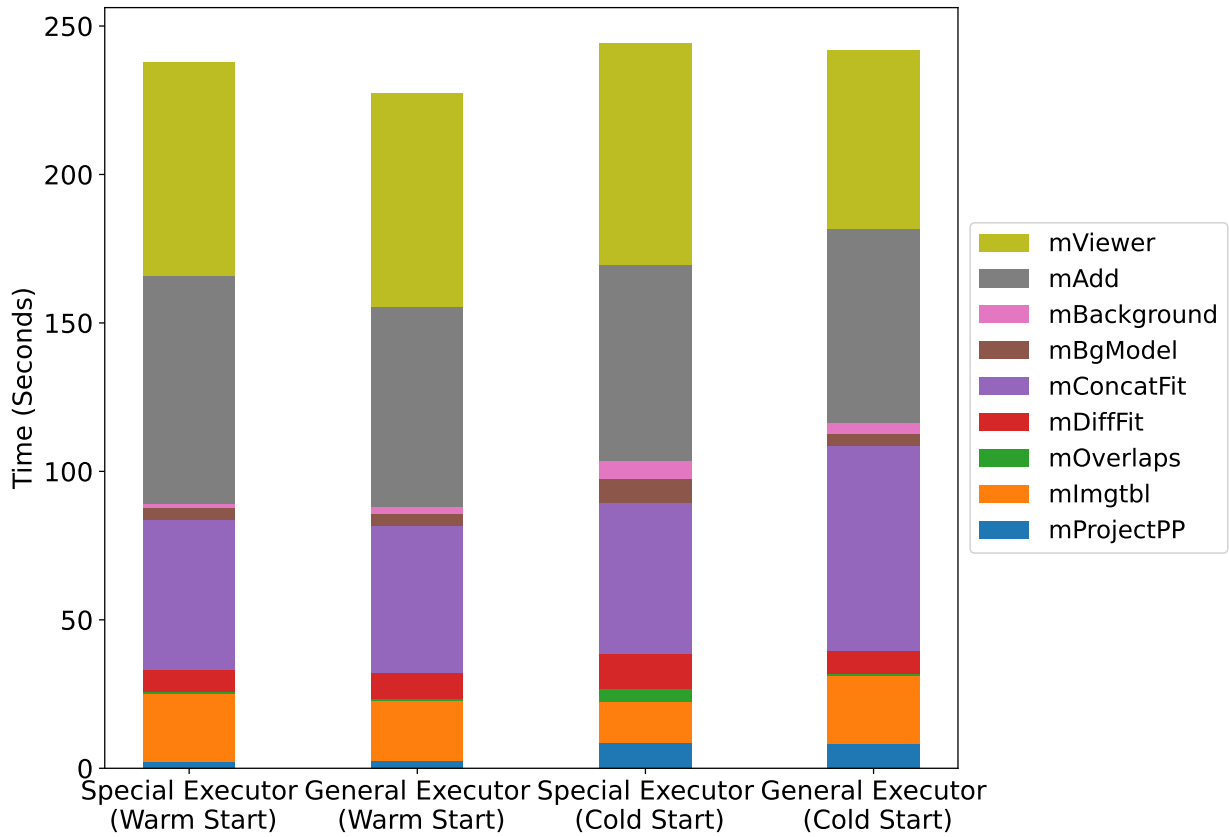


Figure 5.6: The effect of using general executors on the Montage workflow execution time.

## 5.2 Case Study: 1000Genomes

We evaluate different orchestration approaches by comparing the end-to-end and per-stage execution time of 1000Genomes [1], a real-world bioinformatics workflow (Section 5.2.1). Furthermore, We evaluate the effects of the number of parallel tasks (Section 5.2.2) and data compression (Section 5.2.3). Finally, we show the effect of cold starts (Section 5.2.4) and locality optimizations (Section 5.2.5).

**Workflow.** 1000Genomes [1] is an open-source bioinformatics workflow that analyzes gene mutations to identify diseases. Figure 5.7 shows the structure of the workflow. We selected this workflow as it has a complex pattern in which it has a parallel reduce operation. Two parallel reduce operations (`mutations_overlap` and `frequency`) reduce the output of two stages: `individuals_merge` and `sifting`. This application can be configured to control the length and number of parallel tasks. Which allows us to evaluate a spectrum of configurations.

We use the first 100,000 lines of chromosome number 1 and 2504 individuals from 7 populations (ALL, EUR, EAS, AFR, AMR, SAS, and GBR) from the input data version 20130502 [1]. Each task in the `individuals` stage processes 100 lines. Each task in the `individuals` stage produces a compressed output file, leading to a total of 1000 files. The `individuals_merge` reduces all these files. The `mutations_overlap` and `frequency` run a task for each one of the 7 populations of individuals. This creates a workflow with 1016 tasks, which include 1000 parallel `individuals` tasks, one `individuals_merge` task, one `sifting` task, 7 parallel `mutations_overlap` tasks, and 7 parallel `frequency` tasks. We

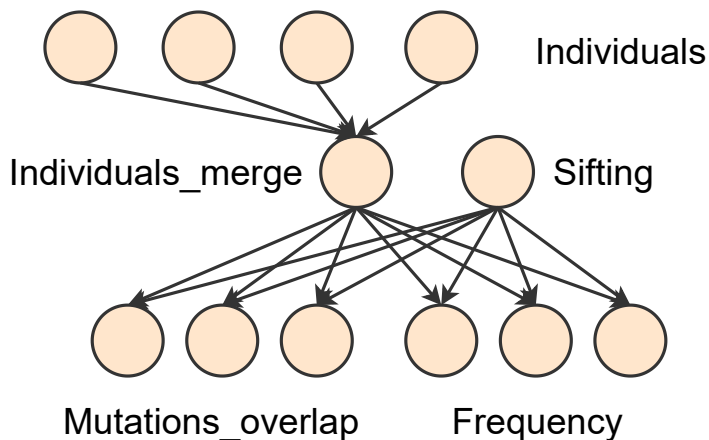


Figure 5.7: The structure of the 1000Genomes workflow.

report the average of 15 trials for each experiment. Unless otherwise specified, the standard deviation of the end-to-end execution time is less than 7% for warm start and serverful experiments and 11% for cold start experiments unless otherwise specified.

### 5.2.1 Comparing Different Orchestration Approaches

We compare the performance of the 1000Genomes workflow using different orchestration approaches. Figure 5.8 shows the average execution time of each stage and the end-to-end execution time of different alternatives. Results show that serverless-centralized and serverless-decentralized alternatives achieve a comparable performance to the serverful-centralized alternative.

We notice that the serverless decentralized alternative produces 1000 file system metadata requests. Unlike Montage, this high number of requests does not impact performance because of the long duration of the tasks of the parallel `individuals` stage which leads to spreading the metadata requests over a longer period.

### 5.2.2 Effect of the Number of Parallel Tasks

Increasing the number of parallel tasks offers more opportunities for parallelizing the execution but can also increase the scheduling overhead. In this section, we study the effect of the number of parallel tasks on performance. The workflow processes 100,000 lines of input. We vary the number of tasks and the number of lines per task in the `individuals` stage. We experiment with having 10,000 tasks (10 lines per task), 2000 tasks (50 lines per task), and 1000 tasks (100 lines per task). Each task produces a compressed output file regardless of the number of input lines.

Figure 5.11 shows the average end-to-end execution time and execution time of each stage. Increasing the number of tasks negatively affects the performance of the parallel stage `individuals` and the reduce stage `individuals_merge`. For the parallel stage, the execution time for the 50 lines and the 10 lines alternatives increases the stage execution time by 15.46% and 101.5% respectively when compared to the 100 lines alternative. This is due to the overhead from handling a larger number of parallel tasks over a fixed number of resources. Furthermore, each task produces one compressed output file, consequently, increasing the number of tasks by 10 times also increases the number of produced files proportionally. This significantly increases the load on the file system and impacts performance and cost.

For the reduce stage, the execution time for the 50 lines and the 10 lines alternatives is 1.8 and 7.9 times the execution time of the 100 lines alternative, respectively. The reduce stage processes the files produced by the parallel stage sequentially. Consequently, the increase in the number of files produced by the parallel stage increases the execution time and cost.

This experiment shows that increasing the number of parallel tasks is not always beneficial. It is unlikely that a cloud provider will have available resources to handle thousands of tasks in parallel. Even if the provider has the resources, recent studies [26] show that cloud providers limit parallelism. This indicates that developers of scientific workflows may be better off opting for a smaller number of larger tasks.

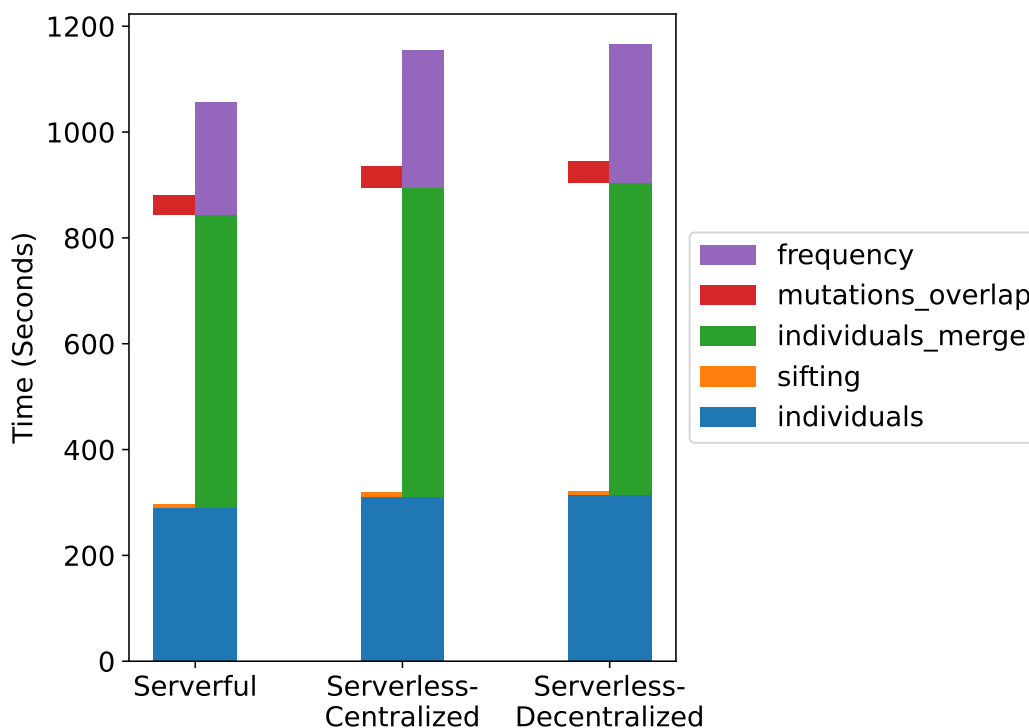


Figure 5.8: The effect of the used orchestration approach on the 1000Genomes workflow execution time.

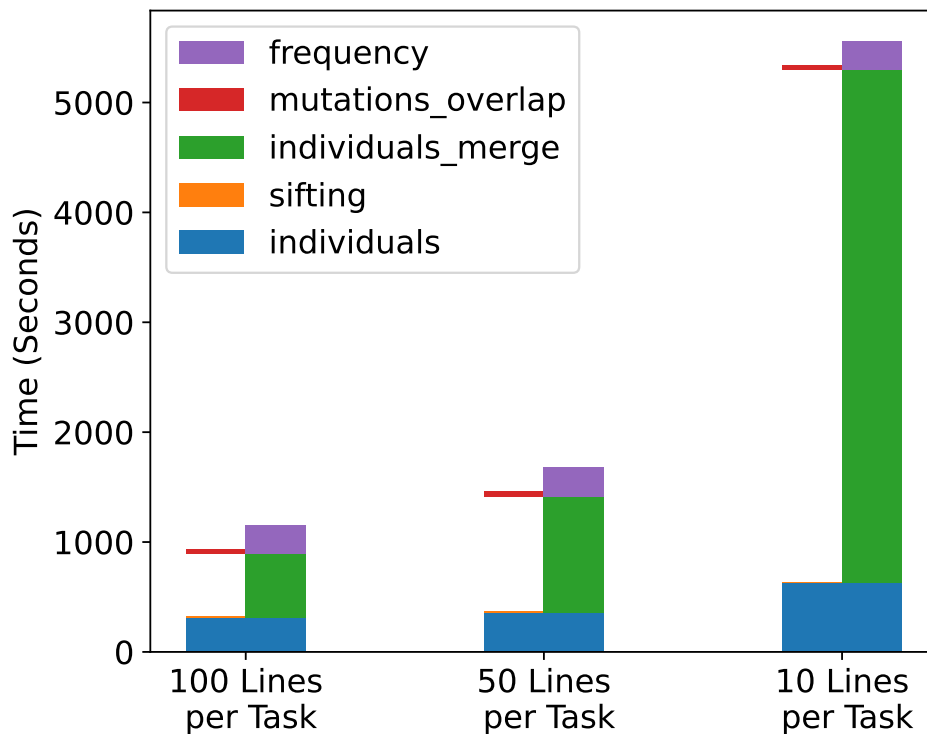


Figure 5.9: The effect of the number of parallel tasks on the 1000Genomes workflow execution time.

### 5.2.3 Effect of Intermediate Data Compression

The 1000Genomes implementation uses compression between stages. Each task in the parallel `individuals` stage produces 2504 output files. Each `individuals` task compresses its output files into a single compressed output before storing it at the shared storage. The reduce stage reads the intermediate compressed files, decompresses them, then sequentially processes the files. Compression introduces a tradeoff between adding the compression overhead and reducing file system operations by reducing the number of files and the data size transferred. To evaluate this trade-off, we implement a version of the 1000Genomes workflow without compression. This changes the size and number of files written to the shared storage which can affect performance.

Figure 5.11 shows the average execution time of each stage and the end-to-end execution time for 1000Genomes with and without compression. We use 100 lines per task leading to

1000 `individuals` tasks. We notice that not compressing the output negatively affects the performance of the parallel stage `individuals` and the reduce stage `individuals_merge`. For the parallel stage, the execution time without compression is 2.23 times the execution time with compression. This is because each parallel task writes 2504 files instead of just one file when using compression. This increases both the execution time and cost. For the reduce stage, the execution time without compression is 3.7 times the execution time with compression. Without compression, the reduce stage has to read 2504 files from the shared storage per task compared to just one read when using compression.

For the `mutations_overlap` the impact of compression is different. Not compressing the data has a positive impact on the `mutations_overlap` stage. No compression alternative reduces the execution time of this stage by 15.85% when compared to the compressed alternative. This stage only reads one file when using compression and reads 2504 files without using compression. However, the overhead from compressing and decompressing is larger than the overhead for writing and reading 2504 files which leads to reducing the execution time. For the `frequency` stage, those overheads are similar which results in a non-noticeable change in the execution time.

This experiment shows that compression presents a trade-off between the processing overhead from compressing and decompressing intermediate data and the IO overhead from writing possibly larger number and bigger files. Our results show that the benefit of using compression is application and stage-specific.

## 5.2.4 Effect of Cold Starts

We evaluate the effect of cold starts of containers on performance. Figure 5.8 shows the average execution time of each stage and the end-to-end execution time of the 1000Genomes workflow under cold starts. Our findings for this workflow match the findings from the Montage workflow. Cold starts does not have a large effect on scientific workflows given their long execution time. For instance, cold starts increased the end-to-end execution time by 1.8% for centralized-serverless and 0.7% for decentralized-serverless. Results also show that cold starts have a larger effect on shorter stages. For instance, for decentralized-serverless, cold starts increase the execution time of `sifting` and `mutations_overlap` by 69% and 15.49% when compared to the execution time with warm starts, respectively. While it increases the execution time of `individuals`, `individuals_merge`, and `frequency` by less than 2% when compared to a warm start version.

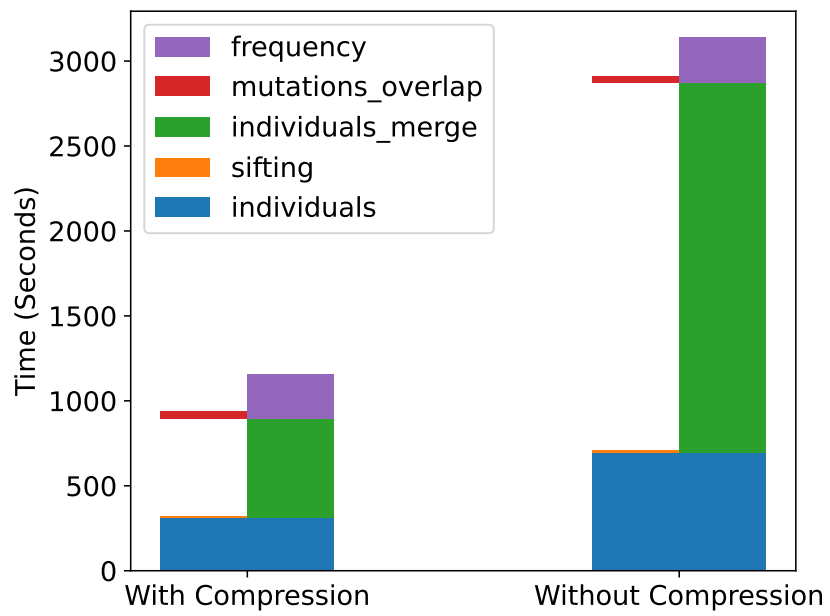


Figure 5.10: The effect of intermediate data compression on the 1000Genomes workflow execution time.

## 5.2.5 Locality Optimizations

In Montage, we identified two locality optimizations (Section 5.1.2): prefetching file privileges and container placement. prefetching file privileges prefetches file privileges at reduce tasks, and container placement places tasks that share a large amount of data on the same node. In this section, we evaluate the benefits of these optimizations on the 1000Genomes workflow.

**Prefetching file privileges:** This optimization does not have a noticeable impact on performance. The long duration of the tasks of the parallel `individuals` stage ensures that most of the written data are automatically flushed and most of the file privileges are automatically released by the nodes. This renders the optimization ineffective while increasing cost from the extra metadata requests.

**Container placement:** This optimization increases the end-to-end execution time. Since the `individuals` parallel stage is long, reducing parallelism for it results in a much longer execution time, overshadowing any performance gains in the `individuals_merge` reduce stage.

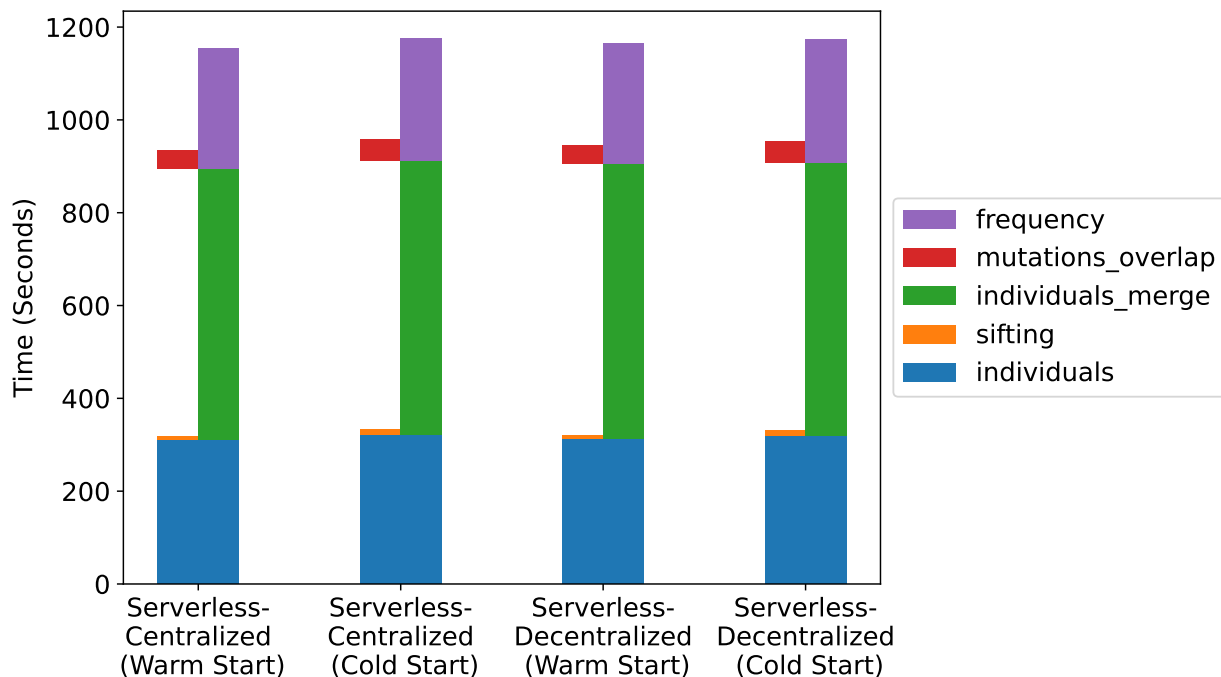


Figure 5.11: The effect of cold starts on the 1000Genomes workflow execution time.

## 5.3 Case Study: SRA Search

We evaluate different orchestration approaches by comparing the end-to-end and per-stage execution time of SRA Search [17], a real-world bioinformatics workflow (Section 5.3.1). We show the effects of the locality optimizations (Section 5.1.2) in Section 5.3.2. Then, we show the effects of the order of execution on parallel tasks in Section 5.3.3. Finally, we show the effect of cold starts in Section 5.3.4.

**Workflow.** SRA Search [17] is an open-source bioinformatics workflow that aligns DNA sequences. Figure 5.12 shows the structure of the workflow. The workflow has three stages: a single task `bowtie2-build` stage, a parallel `bowtie2` stage, and a reduce `merge` stage. An interesting characteristic of this workflow is that the tasks in the parallel stage have a large variation in their execution time.

We use the dataset that is part of the SRA workflow repository [17]. We use 997 input files. Appendix A.1 lists the SRA IDs of the 997 inputs we use. We measured the time it takes to process each input file by the `bowtie2` process. Figure 5.13 shows The CDF of the processing time of all input files in the data set. The figure shows that the processing times vary widely from 114 msec to 121 seconds. We could not use the entire data set in our evaluation because the data set size is 1450 GB and would not fit in the memory of the machines we have in our testbed. To build a representative workload we followed two steps: First, we choose a sample of 50 input files. We choose the files with execution times that represent the complete data set. Figure 5.14 shows the CDF of the execution time for the selected sample. Appendix A.2 lists the SRA IDs we choose for our evaluation. Second, we duplicated the processing of each file 40 times in the same application run. This creates a `bowtie2` stage with 2000 tasks but with data of only 50 input files that can fit in memory. We report the average of 15 trials for each experiment. The standard deviation of the end-to-end execution time is less than 8% for all experiments.

### 5.3.1 Comparing Different Orchestration Approaches

We compare the performance of the SRA search workflow using different orchestration approaches. Figure 5.18 shows the average execution time of each stage and the end-to-end execution time of different alternatives. Results show that serverless-centralized and serverless-decentralized alternatives achieve a comparable performance to the serverful-centralized alternative. We note that the execution time is dominated by the parallel stage. We also notice that the overhead of the orchestration approach is non-noticeable, specially for the serverless decentralized alternative. This is due to the long duration of

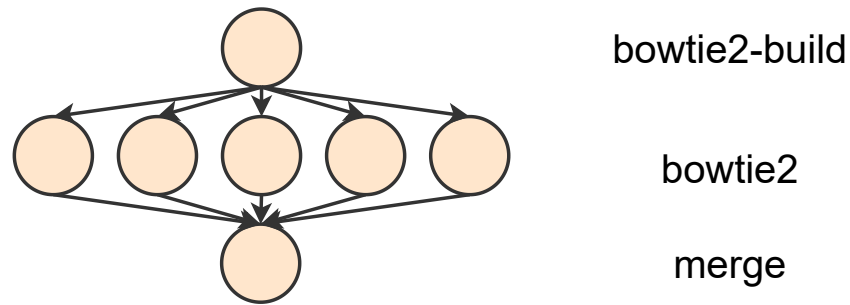


Figure 5.12: The structure of the [SRA](#) Search workflow.

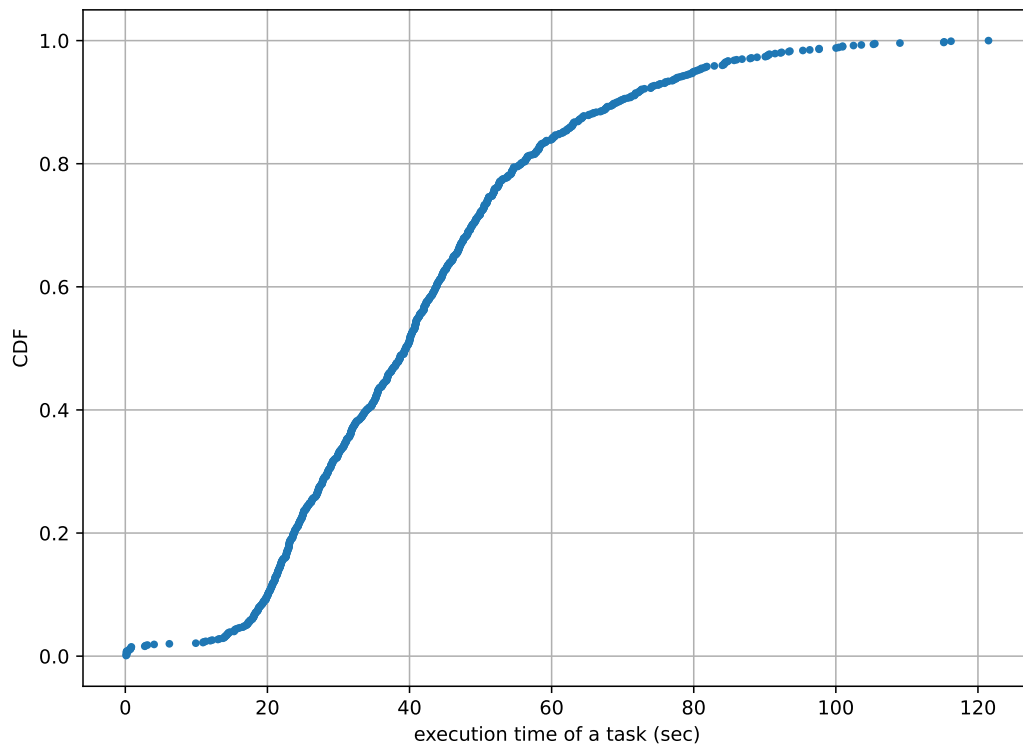


Figure 5.13: CDF of the execution time of the tasks of the parallel stage in the [SRA](#) Search workflow.

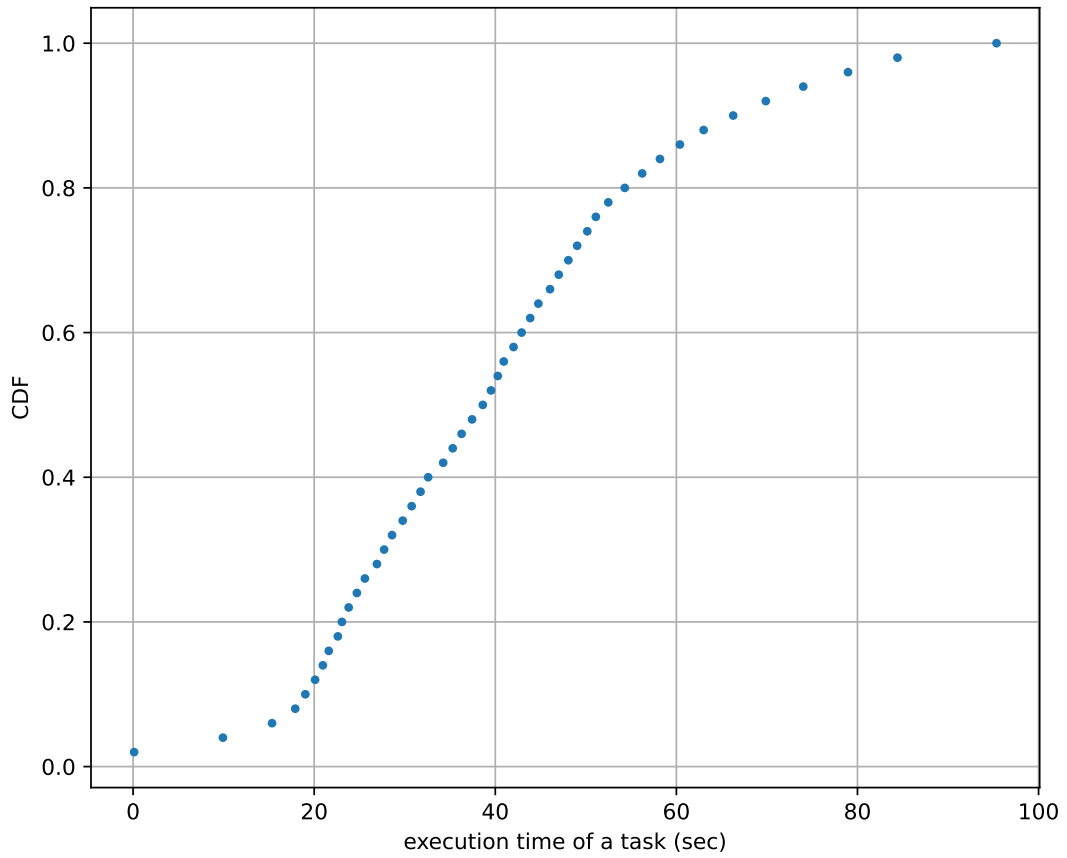


Figure 5.14: CDF of the execution time of the selected tasks of the parallel stage in the [SRA](#) Search workflow.

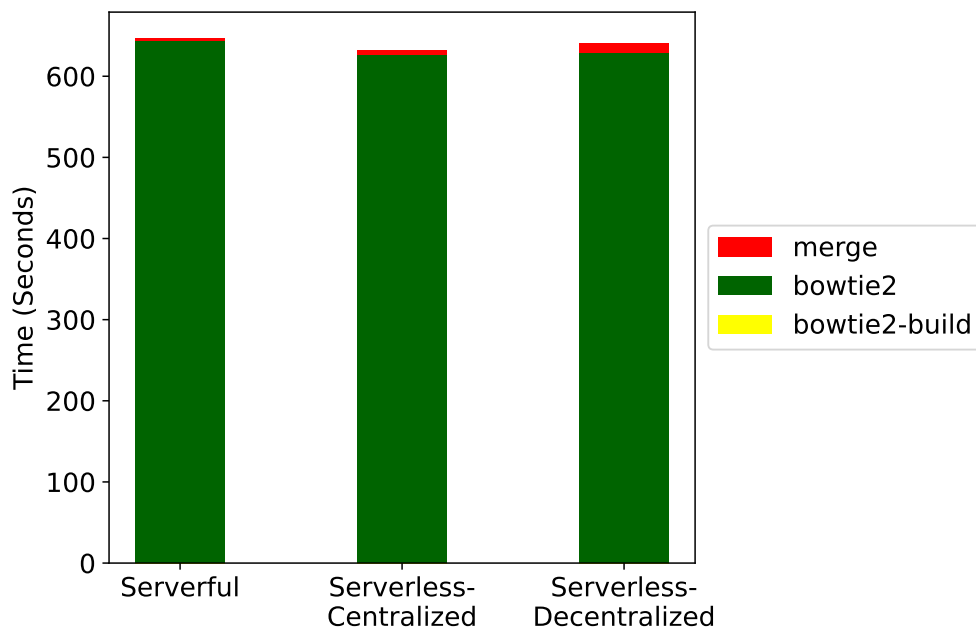


Figure 5.15: The effect of the used orchestration approach on the [SRA](#) Search workflow execution time.

the parallel stage and the variation in the duration of parallel tasks. This indicates that the parallel tasks do not end at the same time. This distributes the requests to the file system over a large period which does not stress the file system. Hence, we do not see any delays in the case of serverless decentralized as was the case in Montage.

### 5.3.2 Locality Optimizations

We evaluate the effect of the prefetching file privileges optimization on the execution time. Figure 5.16 shows the average execution time of each stage and the end-to-end execution time with and without optimization. We note that the container placement optimization impacts performance negatively because the workflow is dominated by the parallel task. Results show that the prefetching file privileges optimization reduces the execution time of the reduce stage `merge` by 29% without a noticeable increase in the parallel stage time. However, the effect on the end-to-end execution time is diminished by the long execution time of the parallel stage. We note that in this case, the optimization may have a negative effect on cost since we are incurring costs from the extra read requests.

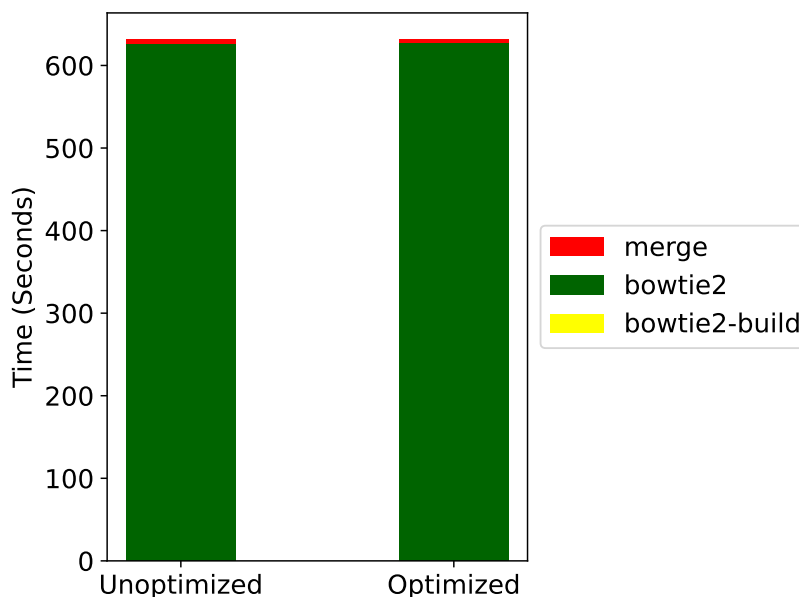


Figure 5.16: The effect of the prefetching file privileges optimization on the [SRA](#) Search workflow execution time.

### 5.3.3 Effect of the Order of Execution of the Parallel Tasks

Given the variability of the execution time of the tasks of the parallel stage, we study the effect of the order of execution on the total time taken by the parallel and reduce stages. We report the total time taken when we execute the tasks in ascending and descending order according to their execution time. Then, we compare them to the original order in the dataset. Figure 5.17 shows the average execution time of each stage in the workflow and the end-to-end execution time of the different orders. We notice that using descending order reduces the execution time of the parallel stage by 7% when compared to the ascending order. This is because descending order ensures that the longest tasks will be well-distributed across the available containers. While in ascending order, the longest tasks are executed after all tasks have been queued, so they might not be well-distributed across workers. Execution order also affects the execution time of the reduce stage. For example, the execution time of the `merge` reduce stage in the descending order is higher by 2 times when compared to the ascending order. This happens as in ascending order the files produced by the short tasks are flushed to disk during the long tasks. This means that when the reduce task executes, it has to force the releasing of a small number of files. In the

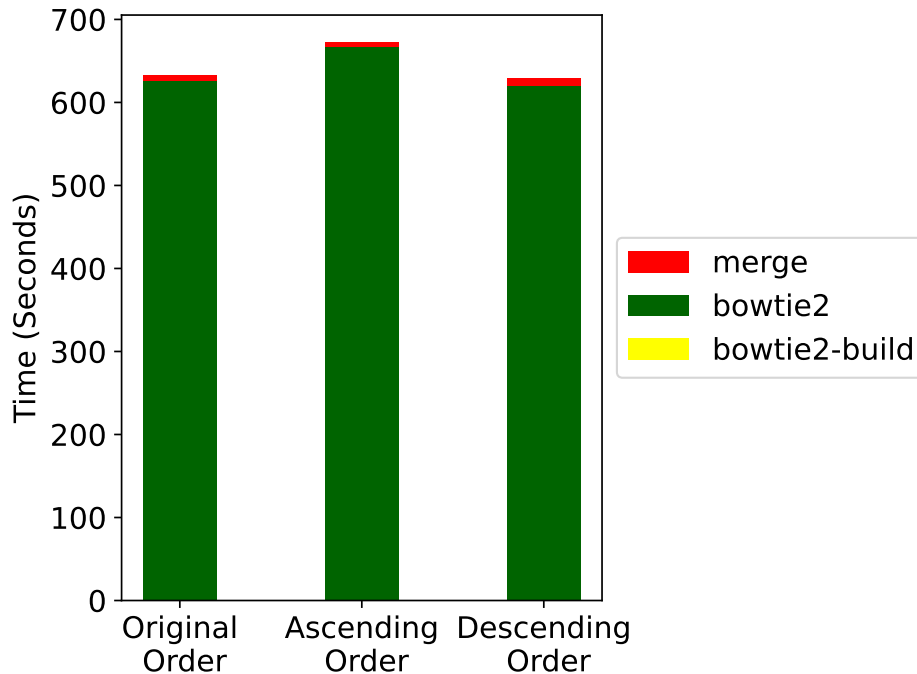


Figure 5.17: The effect of the order of execution on the [SRA Search](#) workflow execution time.

descending order, a large number of files is produced at the end of the parallel stage that did not have enough time to be normally flushed. This forces the reduce stage to wait for the files to be flushed to disk. This behavior also affects the effectiveness of the prefetching file privileges optimization. For instance, the optimization shortened the reduce stage in the ascending order and the descending order by 9.56% and 44.5%, respectively, when compared to an unoptimized version. This is because in the ascending order, most of the files have been flushed during the execution of the long tasks.

Determining the best order to execute the tasks of the parallel stage depends on the type of stages that dominate the workflow. If the workflow is dominated by parallel tasks, then descending order may achieve better performance given that the effect on the reduce stage may not greatly affect the end-to-end execution time of the workflow. If the workflow is dominated by reduce tasks, then ascending order may achieve better performance for the same reasons. At the end, it is a trade-off between gains in the parallel or the reduce stages.

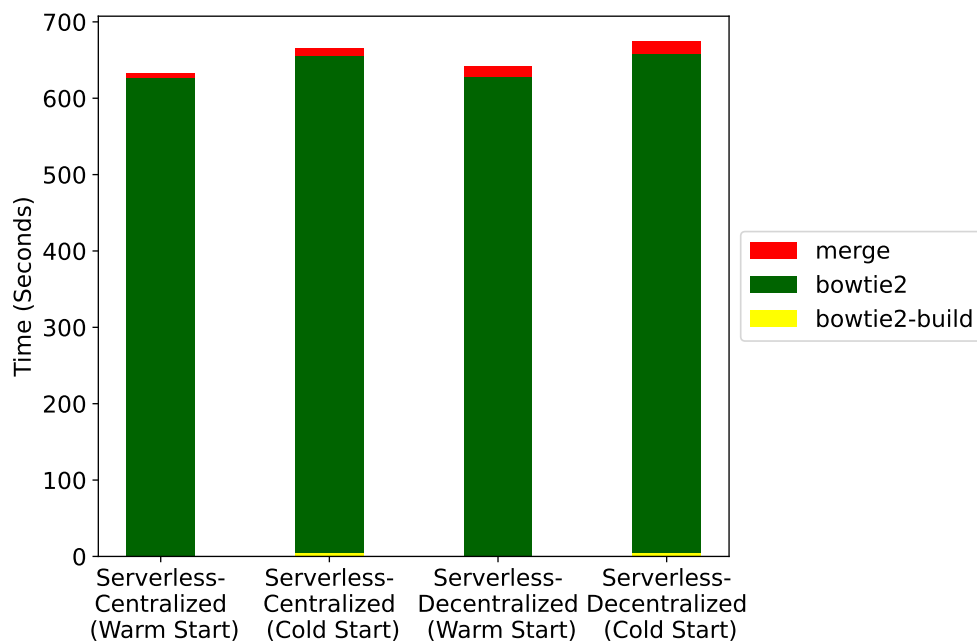


Figure 5.18: The effect of cold starts on the [SRA](#) Search workflow execution time.

### 5.3.4 Effect of Cold Starts

We evaluate the effect of cold starts of containers on performance. Our findings for this workflow match the findings from the previous workflows. Cold starts does not have a large effect on scientific workflow given their long-running nature. For instance, cold starts increased the end-to-end execution time by 5.25% for centralized-serverless and 5.24% for decentralized-serverless as shown in [Figure 5.18](#). Results also show that cold starts have a noticeable effect on only short running stages. For instance, for centralized-serverless, the execution time of `bowtie2-build` and `merge` with cold starts is 13 and 1.75 times the execution time with warm starts, respectively. While it increases the execution time of `bowtie2` by 3.9% when compared to a warm start version.

# Chapter 6

## Cost Analysis

We suggest a model for estimating the costs of executing the serverful and serverless approaches in Section 6.1. Then, we use this model to perform a cost comparison between Serverful-Centralized and Serverless-Centralized (Section 6.2). Our model only captures the service cost and does not include the cost for staff needed to run the application.

### 6.1 Cost Model

In this section, we suggest a model for estimating the costs of executing the serverful and serverless approaches in order to offer a cost comparison between them. Costs in our experiments come from two main sources. The first one is compute costs which is the cost of the computation power that is used to execute the workflow. The second one is storage costs which is the cost of the storage that stores the workflow's data and the cost of the operations done on the data. The pricing of those resources differs from a cloud provider to another. Even for the same cloud provider, pricing changes from time to time. For those reasons, we use standard metrics as a proxy for the cost. The benefit of this approach is that those standard metrics could be used to calculate actual cost at any desired time with any desired cloud provider. Those metrics are:

- **Read Operations:** The number of operations that reads a file or lists files in a directory from the shared storage.
- **Write Operations:** The number of operations that writes a file to the shared storage.

- **VM CPU Hours:** The sum of the number of hours that each CPU in the cluster took to execute the workflow. We use this metric for the serverful-centralized alternative. Since we book the whole cluster during workflow execution, we calculate this metric using Equation 6.1. We use the average execution time of 15 trials in our cost analysis.

$$\#CPU\_Hours = workflow\_execution\_time * \#CPUs \quad (6.1)$$

- **Container CPU Hours:** The sum of the number of hours that each CPU in each serverless container took to execute the workflow. We use this metric for the CaaS version of the serverless-centralized and serverless-decentralized alternatives. We calculate this metric using Equation 6.2. For every container, we use the average execution time of 15 trials in our cost analysis.

$$\#Container\_Core\_Hours = \sum_{i=1}^{\#containers} execution\_time_i * \#CPUs_i \quad (6.2)$$

- **Function CPU Hours:** The sum of the number of hours that each CPU in each serverless function took to execute the workflow. We use this metric for the FaaS version of the serverless-centralized and serverless-decentralized alternatives. We calculate this metric using Equation 6.3. For every task, we use the average execution time of 15 trials in our cost analysis.

$$\#Function\_Hours = \sum_{i=1}^{\#tasks} execution\_time_i \quad (6.3)$$

We note that cloud providers have different costs for a VM CPU hour, a container CPU hour, and a function CPU hour. To be able to compare the cost across these technologies we define two variables.

- $\alpha$ : the ratio of the cost of a container CPU hour to a VM CPU Hour.

$$\alpha = \frac{Cost\_of\_Container\_CPU\_Hour}{Cost\_of\_VM\_CPU\_Hour} \quad (6.4)$$

- $\beta$ : the ratio of the cost of a function CPU hour to a VM CPU Hour.

$$\beta = \frac{Cost\_of\_Function\_CPU\_Hour}{Cost\_of\_VM\_CPU\_Hour} \quad (6.5)$$

## 6.2 Serverful-Centralized Vs Serverless-Centralized

In this section, we do a cost comparison between the serverful-centralized and serverless-centralized alternatives. Since the shared storage is provisioned with the same configurations in both alternatives and they perform the same number of reads and writes, we exclude the number of reads and writes from our cost analysis as the cost is the same in both approaches. The remaining costs are execution cost and orchestration cost.

### 6.2.1 Execution Cost

For the execution cost, we calculate it using Equation 6.1 for the serverful approach, Equation 6.2 for the CaaS version of the serverless approach, and Equation 6.3 for the FaaS version of the serverless approach. Table 6.1 shows the calculated execution cost for the studied workflows.

	Montage	1000Genomes	SRA Search
Serverful (VM CPU Hours)	12.938	58.934	35.928
CaaS Serverless (Container CPU Hours)	1.9	22.0189	34.965
FaaS Serverless (Function CPU Hours)	0.468	17.395	33.9369

Table 6.1: Execution cost for the studied workflows.

Since serverful and serverless approaches use different metrics for comparison we use  $\alpha$  (Equation 6.4) and  $\beta$  (Equation 6.5) variables to compare them.

Figure 6.1 shows the relation between  $\alpha$  and the costs of the serverful and the CaaS versions of the three workflows. The horizontal line represents the cost of the serverful approach. The intersection points between the serverful approach and the serverless approaches indicate a point at which the serverful and serverless approaches have the same cost. We note that each application has a different equivalence point. For SRA search, the cost is equivalent between the two approaches when  $\alpha$  is 1.027, for 1000Genomes when  $\alpha$  is 2.67, and for Montage when  $\alpha$  is 6.8.

Figure 6.2 shows the relation between  $\beta$  and the costs of the serverful and the FaaS versions of the three workflows. The results show that the equivalence point for SRA search is when  $\beta$  is 1.058, for 1000Genomes when  $\beta$  is 3.38, and for Montage when  $\beta$  is 27.64.

We note that there is a difference between the equivalence point for  $\alpha$  and  $\beta$ . The equivalence point for  $\beta$  is 1-4 times larger than that of  $\alpha$ . This is because in our evaluation, FaaS uses fewer function hours to complete a workflow that is because FaaS is more

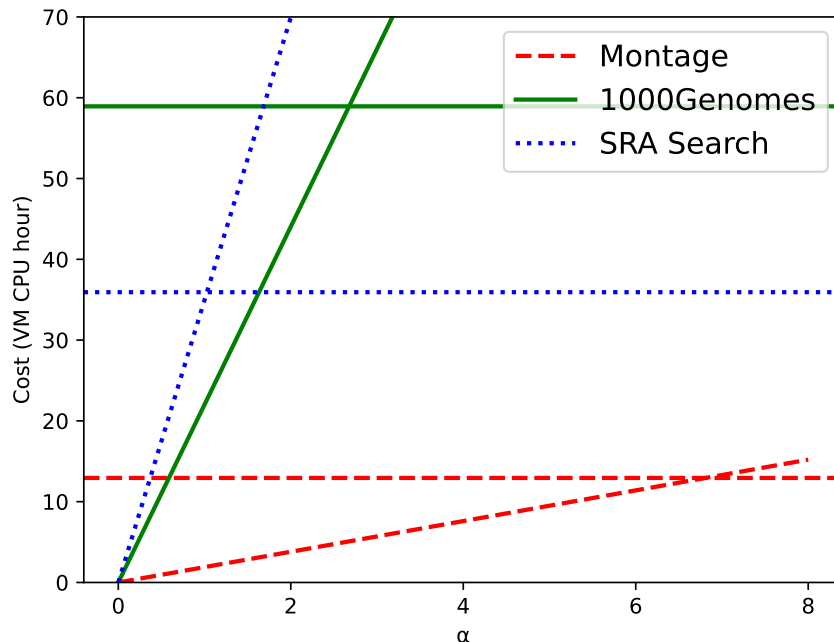


Figure 6.1: The relation between  $\alpha$  and the costs of the serverful and the CaaS version of the serverless approaches. The horizontal lines represent the cost of the serverful approach.

resource-efficient than CaaS. When a task uses only one core, CaaS still charges for all cores allocated to that container. In FaaS this task will be charged for one core only since it only allocates one core. In our experiments, we configured each container in CaaS to use 10 cores. If a stage uses only one core, it is still charged for the 10 cores.

To put these numbers into current cloud providers perspective, at Amazon, a container with 1 CPU core and 1 GB of memory costs \$0.045 per hour, running a FaaS function for an hour with 1 CPU core and 1 GB of memory costs \$0.06, and using a VM with 1 CPU core and 2 GB of memory costs \$0.043. This leads to an  $\alpha$  of 1.04 and  $\beta$  of 1.39. With these numbers, it is cost-efficient to run all our workflows using the serverless approach.

## 6.2.2 An Analysis of the Equivalence Point

Identifying the equivalence point helps practitioners to decide if the serverful or serverless approach is more cost-efficient. Here we present a simple approach to make that decision.

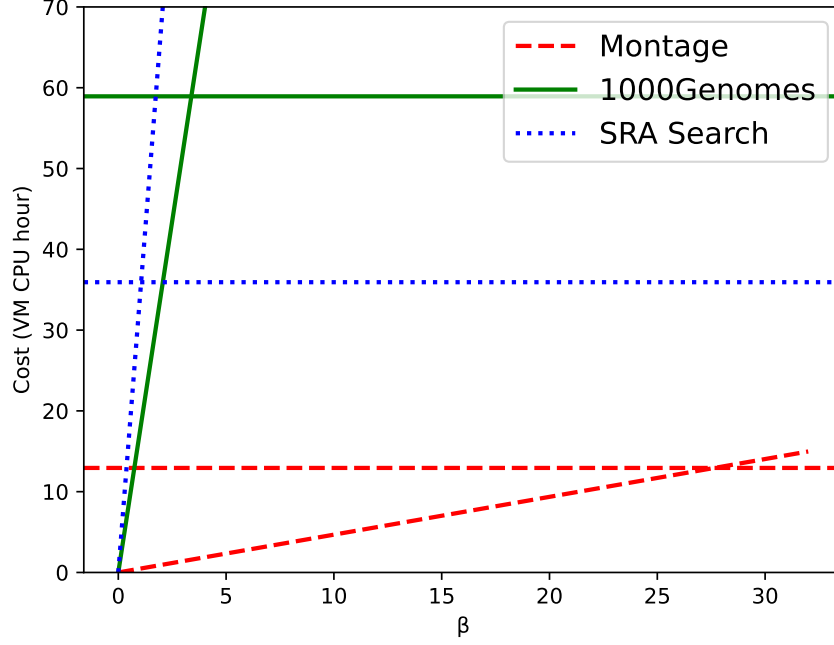


Figure 6.2: The relation between  $\beta$  and the costs of the serverful and the FaaS version of the serverless approaches. The horizontal lines represent the cost of the serverful approach.

We first define two variables:

- $C_{VM}$ : the ratio of the VM CPU hours to the container CPU hours needed to execute a workflow.

$$C_{VM} = \frac{\#VM\_CPU\_Hours}{\#Container\_CPU\_Hours} \quad (6.6)$$

- $F_{VM}$ : the ratio of the VM CPU hours to the [FaaS](#) function CPU hours needed to execute a workflow.

$$F_{VM} = \frac{\#VM\_CPU\_Hours}{\#Function\_CPU\_Hours} \quad (6.7)$$

To understand which alternative is lower in cost to run, we propose the following simple approach:

1. Estimate VM CPU hours, Container CPU hours, and Function CPU hours needed to run the target workflow on the target cloud. This can be done through running a sample workload or doing back-of-envelop-computation using the application characteristics. Use these values to calculate  $C\_VM$  and  $F\_VM$  for that workflow on that cloud provider.
2. Use the cloud provider information sheets to calculate  $\alpha$  and  $\beta$
3. Compare  $\alpha$  and  $\beta$  to  $C\_VM$  and  $F\_VM$ .
  - If  $\alpha < C\_VM$ , the serverful approach is cheaper than [CaaS](#).
  - If  $\beta < F\_VM$  the serverful approach is cheaper than [FaaS](#).

### 6.2.3 Orchestration Cost

For the orchestration cost, we calculate it for a simple orchestrator that runs on a single node (i.e., we ignore replication). Hence, the orchestration cost will be calculated using Equation 6.1. Table 6.2 shows the orchestration cost for the studied workflows. We note that the orchestration cost is directly related to the execution time. Since the execution time is similar in all approaches, the orchestration cost is almost the same for all centralized approaches. The decentralized approach does not incur such costs. However, it incurs costs from extra read and write operations. Specifically, each parallel stage incurs one write operation and a number of read operations that equal the number of tasks it contains.

	Montage	1000Genomes	SRA Search
Serverful (CPU Hours)	1.293	5.86	3.59
Serverless (CPU Hours)	1.368	6.42	3.548

Table 6.2: Orchestration cost for the studied workflows.

# Chapter 7

## Related Work

### 7.1 Utilizing Serverless Environments

Given the potential of serverless computing, several efforts have explored utilizing it for special domains. Lambada [24] explored performing data analytics using serverless computing by introducing a serverless distributed data processing framework. Experiments done with this framework assess the economical and performance viability of using serverless in several scenarios, pinpointing the scenarios where serverless offers advantages. The authors also show the technical challenges that face serverless data analytics. Sprocket [20] is a serverless video processing framework that achieves low latency through intra-video parallelism. It allows developers to create modular, custom video processing pipelines using a simple specification language. The system handles video operations, encoding, decoding, and processing in a highly parallel manner. The paper describes Sprocket’s design on AWS Lambda, demonstrating its performance goals of high parallelism, low latency, and low cost. Cirrus [28] is an ML framework that automates resource management in data centers using serverless computing. It reduces user effort by combining the simplicity of the serverless interface with the scalability of serverless infrastructure. The paper emphasizes the need for a design tailored to both serverless and iterative ML training by showing that Cirrus outperforms designs that focus on a single aspect. For instance, Cirrus is 100x faster than a general-purpose serverless system and 3.75x faster than specialized ML frameworks that run on traditional infrastructures.

These efforts are similar in principle to our work as they also explored the viability of using serverless computing in a specific domain and the challenges faced while doing so. Our work extends these efforts by doing a similar study for scientific workflows.

## 7.2 Scientific Workflows Execution in Serverless Environments

Several efforts have looked into executing scientific workflows in a serverless environment. Malawski et al. [49] have built a [WMS](#) to execute scientific workflows using AWS Lambda and Google Cloud Functions. Their proposed approach is centralized and based on the HyperFlow engine. Moreover, the authors extended HyperFlow with two functions in order to support AWS Lambda and Google Cloud Functions. The authors evaluated the proposed system using a Montage workload. However, the size of the used workload is relatively small (i.e., 165 tasks). Their paper does not compare the proposed approach with other approaches (e.g., the decentralized approach). Rather, it only focuses on evaluating different aspects of the cloud-based functions when executing the workload, such as scalability, portability, and performance variability. Similarly, SWEEP [35] is a centralized, cloud-provider agnostic [WMS](#) that supports both serverless functions and serverless containers. SWEEP was evaluated using Montage and Variant Calling [57]. Burkat et al. [27] investigate the usefulness of serverless containers for executing scientific workflows. Burkat et al. propose a HyperFlow-based [WMS](#) that supports AWS Lambda, AWS Fargate, and Cloud Run. The proposed [WMS](#) adopts a centralized architecture with a one-to-one task to container mapping. Their paper uses four scientific workflows: Ellipsoids, Vina [60], KINC [56], and Soy-KB [45] to compare different CaaS platforms across different metrics, including total execution time and cost.

These efforts focus on developing fully-fledged centralized [WMSs](#) that support various serverless providers. On the other hand, we focus on evaluating the viability of executing scientific workflows in a serverless environment. Also, we study and compare different orchestration approaches to execute scientific workflows. In addition, we discuss several optimizations that can reduce the execution time of scientific workflows in serverless environments.

## 7.3 Accelerating Workflows in a Serverless Environment

Several techniques have been proposed to accelerate the execution of workflows in serverless functions. Palette [18] proposes the use of hints as a simple technique to express locality to the serverless platform. These hints are expressed by users to specify which function invocations will benefit from being deployed in the same instance. Palette load balancer

was implemented and compared to a locality-oblivious load balancer in the open-source version of Azure Functions Host Runtime. Results show that Palette brings noticeable performance benefits compared to the locality-oblivious load balancer.

Mahgoub et al. [47] propose ORION, a technique for performance modeling of serverless workflows to estimate execution time. Based on ORION, the authors propose three optimizations. The first technique is right sizing, which means allocating the right amount of resources for each function invocation to meet the latency requirement with the minimum cost. The second technique is co-locating multiple parallel instances of function to be executed on the same virtual machine (VM) will be beneficial. The last technique is the pre-warming of VMs at which future functions will be executed to avoid cold starts. Results show that ORION can reduce the tail latency (i.e., 95th percentile) by 90% without increasing the cost.

SONIC [46] is a management layer for passing data between different Lambda invocations. SONIC tries to optimize a workflow by selecting the best approach to pass data between different functions in the workflow. SONIC design considers three techniques for passing data: VM-Storage, Direct-Passing, and Remote storage. Since no data passing approach prevails under all scenarios, SONIC selects the data passing approach based on several factors including input data size, intermediate data size, application parallelism level, and network bandwidth. SONIC was integrated with OpenLambda and compared to other alternatives (e.g., AWS-Lambda with S3 and ElasticCache Redis). Results show that SONIC achieves better performance compared to other alternatives.

WiseFuse [48] recognizes two issues that affect the performance of serverless workflows at Azure Durable Functions [51]. The first one is the lack of efficient communication methods between serverless functions in a workflow. The second one is stragglers in parallel stages. To address those issues, WiseFuse introduces three optimizations. The first is Fusion which combines sequential stages into one stage to reduce communication overhead. The second is Bundling which executes a group of parallel tasks in one VM to improve resource sharing and reduce skew. The third one is Resource Allocation which chooses the right size for VMs to reduce latency and cost. Evaluation shows that WiseFuse improves the end-to-end execution time and cost for three serverless applications: Video Analytics, Approximate SVD, and ML Analytics.

These efforts propose optimizations for general workflows represented by DAGs, while our work focuses on scientific workflows, which are more data- and compute-intensive. Furthermore, we propose optimizations related to the deployed distributed file systems (Section 5). In addition, we discuss and evaluate different orchestration approaches to execute scientific workflows.

## 7.4 Serverless Workflows Orchestrators

Several serverless workflows orchestrators exist in both academia and industry. AWS Step Functions [25] is a cloud service provided by AWS that simplifies the process of coordinating and orchestrating multiple AWS services and serverless functions into serverless workflows. It offers a visual and easy-to-use interface for defining, managing, and executing workflows without the need for custom code. Workflows can integrate various AWS services, such as AWS Lambda, Amazon S3, and AWS Batch, making it a versatile tool for a wide range of applications. Azure Durable Functions [51] is an extension of Azure Functions [53] that simplifies the development of stateful and orchestrator-based workflows. It abstracts away the complexities of managing state and concurrency, allowing developers to focus on defining the flow of their application logic using orchestrator functions and entity functions. This can be done in various programming languages and can be integrated with Azure services. Similar orchestrators exist for other cloud providers as well.

A recent study [26] evaluated the performance of popular workflow orchestrators offered by cloud providers. The study showed that some orchestrators add a non-negligible overhead in parallel stages. For instance, with increasing the number of parallel tasks from 5 to 320, the overhead of AWS Step Functions increases exponentially. Moreover, the maximum number of concurrent tasks noticed is around 85. This behavior is likely related to implementation rather than resource constraints as the results are stable. Similar observations about limiting concurrency were noticed in another paper [44] as well. The results for Azure Durable Functions show a lot of variability past 20 concurrent tasks. This could be due to a resource availability or provisioning limitation. These findings suggest that implementation-related factors, resource constraints, and resource provisioning play a significant role in the observed overhead. They also suggest that current serverless orchestrators may not be suitable for executing scientific workflows that exhibit massive parallelism.

## 7.5 Decentralized Orchestration of Serverless Workflows

Unum [44] is a decentralized application-level serverless orchestration system. The Unum evaluation with a synthetic benchmark and a set of applications shows that it performs better and costs less than the centralized AWS Step Functions orchestration approach. The authors argue that decentralized orchestration is better for both serverless providers

and developers. For developers, it brings more flexibility to implement custom patterns as needed and apply application-specific optimization. For providers, it frees them from managing and hosting another complex service. Furthermore, they showed that decentralized orchestration can bring performance and cost improvements without any additional platform-provided infrastructure. This makes decentralized orchestration benefit from any improvements to the existing infrastructure that it uses.

We note that these conclusions cannot be generalized for all centralized and decentralized orchestrators. As our evaluation shows, decentralized orchestration can be slower with massive parallel stages due to overwhelming the storage service.

# Chapter 8

## Discussion and Concluding Remarks

### 8.1 Viability of Using Serverless Computing to Execute Scientific Workflows

Our evaluation shows that serverless computing is a viable approach for executing scientific workflows. Performance-wise, our evaluation shows that the serverless approach achieves comparable performance to the serverful approach. Cost-wise, the serverless approach may bring cost savings compared to the serverful approach.

We note that serverless brings operational benefits as developers focus only on the development of their applications, while the cloud provider is responsible for managing, provisioning, and dynamically scaling resources.

State-of-the-art cloud offerings do not readily support complex workflows. We discuss workflow characteristics and needed platform support in the following subsection.

### 8.2 Characteristics of Scientific Workflows

In this section, we discuss the impact of the characteristics of scientific workflows on executing them on serverless infrastructure.

**Burstiness.** Scientific workflows are bursty compared to typical serverless workloads. This means that they have higher variability in resource demand. This characteristic makes autoscaling a desirable feature that can attract scientific workflows to the serverless

approach. However, this burstiness can stress the serverless platform. We faced this issue in our experiments where we had to scale certain parts of Knative to be able to handle the huge spike in requests. A workflow orchestrator that targets scientific workflows must consider this.

**Long duration.** Tasks of scientific workflows are typically long. Their duration may be more than the current limits imposed by serverless platforms. For instance, during our experiments, we had to increase the timeouts in Knative to be able to run the workflow. However, we believe that this is not a major issue as these limits are increasing with time [64]. Furthermore, some techniques such as checkpointing [64] were introduced to overcome this issue. A second impact of the long execution time is that cold starts are less of an issue for scientific workflows.

**Size of intermediate data.** The large size of intermediate data in scientific workflow has a huge impact on performance. The design of the proposed locality optimizations suggests that it is critical to coordinate the scheduling decisions and the file system optimizations. Such coordination could be done by the cloud provider, the orchestrator, or the user. Optimization opportunities could be identified from analyzing the workflow patterns or from hints from the user.

**Variance in execution time.** Scientific workflows can have a large variance in the execution time of tasks in a parallel stage. This variability should be taken into consideration when making scaling decisions. Spinning a large number of containers to handle these tasks may not necessarily impact the total execution time, as the orchestrator has to wait for the completion of long tasks. This will stress the infrastructure without any benefits. The serverless platform should take this into consideration and only utilize needed resources.

**Interface with storage.** Scientific workflows usually access intermediate data via a POSIX-compliant file system interface. Although less popular than other interfaces, such as object storage, this interface is needed to support legacy workflow applications. This suggests that these interfaces should be optimized by the cloud providers to align them with the access patterns and demands inherent in scientific workflows.

### 8.3 Impact of the Optimizations

The optimizations introduced in this study have a different impact on the total execution time, cost, and resource utilization for different workflows. The "container placement" optimization should be used when the reduction in the execution time of reduce stages is larger than the increment in the execution time of parallel stages. This will reduce the

total execution time of the workflow, similar to the improvement observed in Montage. Otherwise, it will increase the total execution time, which is the case in [SRA Search](#) and 1000Genomes. Furthermore, if the majority of running containers are of a few parallel stages, these containers will be placed on a few nodes resulting in low resource utilization.

The "prefetching file privileges" optimization should be used when a large number of tasks finish at the end of the reduce stage. In this case, it is likely that the data written by those tasks may not be flushed by the start of the reduce stage. This will shorten the execution time for reduce stages. The impact on the total execution time will be noticeable if the workflow is dominated by the reduce stages (e.g., Montage). If the tasks of the parallel stage are long, this optimization does not have any effect on performance (e.g., 1000Genomes). The long duration of tasks ensures that most of the written data are automatically flushed and most of the file privileges are automatically released by the nodes. This renders the optimization ineffective. Moreover, this optimization might incur higher costs if the duration of the parallel tasks has a large variance. In this case, the reducer might be provisioned early, wait for a while without anything to do as the other parallel tasks are still running, then exit without executing the reducer task, increasing cost. For this scenario, it might be better to just open all the files at once before the parallel stage finishes.

## 8.4 Implementing the Decentralized Orchestration Approach

Due to its distributed nature, implementing, modifying, and debugging the decentralized orchestration approach requires a considerable amount of effort. This is calling for the need for further tools and libraries that offer standard components for implementing common patterns in workflows while transparently handling common issues such as logging. Regarding the implementation of parallel tasks, the used method to check the existence of all output files of parallel tasks can significantly affect the execution time. In our implementation, we check only the number of the output files. However, if a more sophisticated method is used (e.g., checking the name of the files), the performance deterioration will be much worse as it will generate a higher load on the distributed file system.

# References

- [1] 1000Genomes Pegasus Workflow. <https://github.com/pegasus-isi/1000genome-workflow>, 2023.
- [2] Alphafold pegasus workflow. <https://github.com/pegasus-isi/alphafold-pegasus>, 2023.
- [3] Bowtie 2: Fast and sensitive read alignment. <https://bowtie-bio.sourceforge.net/bowtie2/index.shtml>, 2023.
- [4] Casa wind workflow. <https://github.com/pegasus-isi/casa-wind-workflow>, 2023.
- [5] Ceph file system. <https://docs.ceph.com/en/pacific/cephfs/index.html>, 2023.
- [6] Execution instances for cycles workflow. <https://github.com/wfcommons/pegasus-instances/tree/master/cycles>, 2023.
- [7] Galaxy classification workflow. <https://github.com/pegasus-isi/galaxy-classification-workflow>, 2023.
- [8] Gluster file system. <https://www.gluster.org/>, 2023.
- [9] IGSR: The International Genome Sample Resource: Supporting open human variation data. <https://www.internationalgenome.org/>, 2023.
- [10] Lung instance segmentation workflow. <https://github.com/pegasus-isi/lung-instance-segmentation-workflow>, 2023.
- [11] Lustre file system. <https://www.lustre.org/>, 2023.
- [12] Page imputation. <https://github.com/pegasus-isi/page-imputation>, 2023.

- [13] Pegasus orcasound workflow. <https://github.com/pegasus-isi/orcasound-workflow>, 2023.
- [14] Samtools. <https://www.htslib.org/>, 2023.
- [15] Seismology workflow. <https://github.com/pegasus-isi/seismology-workflow>, 2023.
- [16] SRA - NCBI. <https://www.ncbi.nlm.nih.gov/sra>, 2023.
- [17] SRA Search Pegasus Workflow. <https://github.com/pegasus-isi/sra-search-pegasus-workflow>, 2023.
- [18] Mania Abdi, Sam Ginzburg, Charles Lin, Jose M Faleiro, Íñigo Goiri, Gohar Irfan Chaudhry, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, May 2023.
- [19] Samer Al-Kiswany, Lauro B. Costa, Hao Yang, Emalayan Vairavanathan, and Matei Ripeanu. A cross-layer optimized storage system for workflow applications. *Future Generation Computer Systems*, 75:423–437, 2017.
- [20] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC '18*, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.
- [21] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems*, 124:215–229, 2021.
- [22] AWS. Amazon s3 - cloud object storage. <https://aws.amazon.com/s3/>, 2023.
- [23] AWS. Aws fargate: Serverless compute for containers. <https://aws.amazon.com/fargate/>, 2023.
- [24] AWS. Aws lambda. <https://aws.amazon.com/lambda/>, 2023.
- [25] AWS. Aws step functions: Visual workflows for distributed applications. <https://aws.amazon.com/step-functions/>, 2023.

- [26] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 25–30, New York, NY, USA, 2019. Association for Computing Machinery.
- [27] Krzysztof Burkat, Maciej Pawlik, Bartosz Balis, Maciej Malawski, Karan Vahi, Mats Rynge, Rafael Ferreira da Silva, and Ewa Deelman. Serverless containers – rising viable approach to scientific workflows. In *2021 IEEE 17th International Conference on eScience (eScience)*, pages 40–49, 2021.
- [28] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.
- [29] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.
- [30] Mark Van der Boor, Sem C. Borst, Johan S. H. Van Leeuwen, and Debankur Mukherjee. Scalable load balancing in networked systems: A survey of recent advances. *SIAM Review*, 64(3):554–622, 2022.
- [31] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX Annual Technical Conference*, pages 1–14, 2019.
- [32] Abdallah Elshamy, Ahmed Alquraan, and Samer Al-Kiswany. A study of orchestration approaches for scientific workflows in serverless computing. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '23, page 34–40, New York, NY, USA, 2023. Association for Computing Machinery.
- [33] GCP. Google cloud functions. <https://cloud.google.com/functions>, 2023.
- [34] GCP. Google cloud run: Container to production in seconds. <https://cloud.google.com/run>, 2023.

- [35] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. Sweep: Accelerating scientific research through scalable serverless workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC '19 Companion*, page 43–50, New York, NY, USA, 2019. Association for Computing Machinery.
- [36] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013. Special Section: Recent Developments in High Performance Computing and Security.
- [37] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific workflow applications on amazon ec2. In *2009 5th IEEE International Conference on E-Science Workshops*, pages 59–66, 2009.
- [38] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P Berman, and Phil Maechling. Data sharing options for scientific workflows on amazon ec2. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE, 2010.
- [39] Daniel Katz, G. Berriman, John Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, and Roy Williams. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4, 05 2010.
- [40] Knative. Knative: Serverless containers in kubernetes environments. <https://knative.dev>, 2023.
- [41] Janez Kranjc, Vid Podpečan, and Nada Lavrač. Clowdflows: a cloud based scientific workflow platform. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part II 23*, pages 816–819. Springer, 2012.
- [42] Kubernetes. Kubernetes: Production-grade container orchestration. <https://kubernetes.io/>, 2023.
- [43] Young Choon Lee, Hyuck Han, Albert Y. Zomaya, and Mazin Yousif. Resource-efficient workflow scheduling in clouds. *Knowledge-Based Systems*, 80:153–162, 2015. 25th anniversary of Knowledge-Based Systems.

- [44] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing more with less: Orchestrating serverless applications without an orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1505–1519, Boston, MA, April 2023. USENIX Association.
- [45] Yang Liu, Saad Khan, Juexin Wang, Mats Rynge, Yuanxun Zhang, Shuai Zeng, Shiyuan Chen, Joao Vitor Maldonado dos Santos, Babu Valliyodan, Prasad Calyam, Nirav Merchant, Henry Nguyen, Dong Xu, and Trupti Joshi. Pgen: Large-scale genomic variations analysis workflow and browser in soykb. *BMC Bioinformatics*, 17:337, 10 2016.
- [46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.
- [47] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.
- [48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. In *Abstract Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS/PERFORMANCE '22, page 57–58, New York, NY, USA, 2022. Association for Computing Machinery.
- [49] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 110:502–514, 2020.
- [50] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, 2018.
- [51] Microsoft. Durable functions overview - azure — microsoft learn. <https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc>, 2023.

- [52] Microsoft Azure. Azure container instances: Launch containers with hypervisor isolation. <https://azure.microsoft.com/en-us/products/container-instances>, 2023.
- [53] Microsoft Azure. Azure functions. <https://azure.microsoft.com/en-ca/products/functions/>, 2023.
- [54] Nicholas Mills, F. Alex Feltus, and Walter B. Ligon III. Maximizing the performance of scientific data transfer by optimizing the interface between parallel file systems and advanced research networks. *Future Generation Computer Systems*, 79:190–198, 2018.
- [55] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.
- [56] William L. Poehlman, Mats Rynge, D. Balamurugan, Nicholas Mills, and Frank A. Feltus. Osg-kinc: High-throughput gene co-expression network construction using the open science grid. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1827–1831, 2017.
- [57] Suyash Shringarpure, Andrew Carroll, Francisco De La Vega, and Carlos Bustamante. Inexpensive and highly reproducible cloud-based variant calling of 2,535 human genomes. *PloS one*, 10:e0129277, 06 2015.
- [58] Tibor Šimko, Lukas Heinrich, Harri Hirvonsalo, Dinos Kousidis, and Diego Rodríguez. Reana: A system for reusable research data analyses. In *EPJ web of conferences*, volume 214, page 06034. EDP Sciences, 2019.
- [59] Tibor Šimko, Lukas Heinrich, Harri Hirvonsalo, Dinos Kousidis, and Diego Rodríguez. Reana: A system for reusable research data analyses. In *EPJ web of conferences*, volume 214, page 06034. EDP Sciences, 2019.
- [60] Oleg Trott and Arthur J. Olson. Autodock vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry*, Jun 2009.
- [61] Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery.

- [62] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.
- [63] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.
- [64] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery.

# APPENDICES

# Appendix A

## Dataset of the SRA Search Workflow

### A.1 [SRA](#) IDs in the Dataset

The dataset contains the following [SRA](#) IDs: SRR1518983, SRR1518989, SRR1518977, SRR1518982, SRR1518991, SRR1518985, SRR1518973, SRR1516220, SRR1518981, SRR1518974, SRR1518976, SRR1518979, SRR1518995, SRR1518994, SRR2006285, SRR2006800, SRR2006801, SRR1518379, SRR1518455, SRR1518456, SRR1518450, SRR1518452, SRR1518465, SRR1518467, SRR1518469, SRR1518471, SRR1518473, SRR1518480, SRR1518477, SRR1518479, SRR1518482, SRR1518484, SRR1518537, SRR1518539, SRR1518541, SRR1518542, SRR1518543, SRR1518544, SRR1518546, SRR1518547, SRR1518550, SRR1518552, SRR1103366, SRR1103542, SRR1103543, SRR1103569, SRR1103598, SRR1103614, SRR1103622, SRR1518555, SRR1518558, SRR1518562, SRR1518564, SRR1518565, SRR1518569, SRR1518570, SRR1518572, SRR1518574, SRR1518576, SRR1518581, SRR1518583, SRR1518584, SRR1518621, SRR1518623, SRR1518633, SRR1518625, SRR1518627, SRR1518629, SRR1518634, SRR1518636, SRR1518638, SRR1518640, SRR1518642, SRR1518644, SRR1518645, SRR1518647, SRR1518649, SRR1518651, SRR1518405, SRR1518406, SRR1518407, SRR1518408, SRR1518409, SRR1518412, SRR1518413, SRR1518414, SRR1518415, SRR1518416, SRR1518417, SRR1518418, SRR1518419, SRR1518420, SRR1518421, SRR1518422, SRR1518429, SRR1518430, SRR1518437, SRR1518438, SRR1518440, SRR1518442, SRR1518443, SRR1518444, SRR1518445, SRR1518449, SRR1518451, SRR1518453, SRR1518466, SRR1518468, SRR1518470, SRR1518403, SRR1516219, SRR1518402, SRR1518411, SRR1518446, SRR1518447, SRR1518448, SRR1518481, SRR1518483, SRR1518538, SRR1518540, SRR1518472, SRR1518474, SRR1518475, SRR1518478, SRR1518545, SRR1518548, SRR1518549, SRR1518551, SRR1518997, SRR1518556, SRR1518557, SRR1518554, SRR1518559, SRR1518563, SRR1518568, SRR1518571, SRR1518573, SRR1518575, SRR1518577, SRR1518579, SRR1518580, SRR1518582, SRR1518585,

SRR1518586, SRR1518622, SRR1518624, SRR1518626, SRR1518628, SRR1518630, SRR1518631, SRR1518635, SRR1518637, SRR1518639, SRR1518641, SRR1518643, SRR1518646, SRR1518648, SRR1518650, SRR1518652, SRR1518653, SRR1518654, SRR1518655, SRR1518656, SRR1518938, SRR1518939, SRR1518659, SRR1518660, SRR1518661, SRR1518662, SRR1518663, SRR1518664, SRR1518665, SRR1518666, SRR1518667, SRR1518668, SRR1518669, SRR1518918, SRR1518919, SRR1518920, SRR1518921, SRR1521488, SRR1518922, SRR1518923, SRR1518924, SRR1518925, SRR1518926, SRR1518927, SRR1518928, SRR1518929, SRR1518930, SRR1518931, SRR1518932, SRR1518933, SRR1518934, SRR1518935, SRR1518936, SRR1518940, SRR1518941, SRR1518942, SRR1518943, SRR1518944, SRR1518945, SRR1518946, SRR1518947, SRR1518948, SRR1518949, SRR1518950, SRR1518951, SRR1518952, SRR1518953, SRR1518954, SRR1518955, SRR1518960, SRR1518961, SRR1518962, SRR1518963, SRR1518964, SRR1518965, SRR1518966, SRR1518967, SRR1518968, SRR1518969, SRR1518970, SRR1518971, SRR2088900, SRR3136954, SRR2089331, SRR2143449, SRR2095926, SRR3136938, SRR3136937, SRR2502767, SRR3136955, SRR2096799, SRR2088892, SRR2088899, SRR3136936, SRR2089250, SRR2140566, SRR2140568, SRR2138603, SRR2138599, SRR2140567, SRR2143404, SRR2143424, SRR2089243, SRR3136935, SRR3136939, SRR2502769, SRR3112253, SRR2223155, SRR2144173, SRR2144352, SRR2143520, SRR3098568, SRR2143505, SRR2143879, SRR2143506, SRR2144350, SRR2085794, SRR2085795, SRR1922808, SRR1922805, SRR1922809, SRR1922810, SRR1923031, SRR1922560, SRR1922556, SRR1927230, SRR1922572, SRR1922561, SRR1922713, SRR1922562, SRR1923000, SRR1922800, SRR1922799, SRR1922798, SRR3198326, SRR1922801, SRR1922803, SRR1922802, SRR1922806, SRR1922813, SRR1922814, SRR1922818, SRR1922817, SRR1922821, SRR1922823, SRR1922824, SRR1922837, SRR1923050, SRR1922826, SRR1922827, SRR1922838, SRR1922828, SRR1923047, SRR1922839, SRR1922840, SRR1922841, SRR1923048, SRR1923044, SRR1922874, SRR1922850, SRR1763626, SRR1763629, SRR1763625, SRR2009707, SRR1763627, SRR1763647, SRR1763644, SRR1763648, SRR2010213, SRR1763651, SRR1763630, SRR2009784, SRR1763631, SRR1763632, SRR1763635, SRR1922858, SRR1763636, SRR1763652, SRR1917732, SRR1918767, SRR1918783, SRR1763653, SRR1918771, SRR1763637, SRR1763638, SRR1763640, SRR1763641, SRR1763642, SRR1763671, SRR2009780, SRR1763662, SRR1763664, SRR1763663, SRR2009781, SRR1763665, SRR1763668, SRR2010619, SRR2010659, SRR1918768, SRR1918774, SRR1918766, SRR1918775, SRR1763667, SRR1918772, SRR1918769, SRR1918770, SRR2010658, SRR1918776, SRR1763669, SRR1919630, SRR1919624, SRR1919625, SRR1919632, SRR1919631, SRR1922877, SRR1919633, SRR1919634, SRR1919635, SRR1919931, SRR1919930, SRR1919932, SRR1919988, SRR1919989, SRR1919990, SRR1919982, SRR1919983, SRR1919986, SRR1919984, SRR1922571, SRR1927226, SRR2010618, SRR1918773, SRR2096800, SRR2095925, SRR2143504, SRR2095937, SRR2095962, SRR2095974, SRR2093843, SRR2088777, SRR2089254, SRR3178077, SRR2089347, SRR2143503, SRR2088877, SRR2088902, SRR2093809, SRR2095936, SRR2088901, SRR2095346, SRR2088776, SRR2093807, SRR2088864, SRR2095385, SRR2096963, SRR2138598, SRR2093832, SRR2143447, SRR2093839, SRR2089247, SRR2093834, SRR2095366, SRR2095367, SRR2138597, SRR2089242, SRR2088879,

SRR2088778, SRR2138601, SRR2093836, SRR2093840, SRR2096200, SRR2143450, SRR2089244, SRR2093830, SRR2089300, SRR2138595, SRR2088810, SRR2138600, SRR2096011, SRR2093772, SRR2143446, SRR2095928, SRR2093831, SRR2088903, SRR2088878, SRR2143571, SRR3177956, SRR3177961, SRR3177954, SRR3178003, SRR3177959, SRR3177953, SRR3177958, SRR3177963, SRR3177964, SRR3177965, SRR3177960, SRR3177966, SRR3177957, SRR3177977, SRR3177968, SRR3177983, SRR3178045, SRR3177967, SRR3178005, SRR3177969, SRR3177979, SRR3177970, SRR3177980, SRR3177981, SRR3177984, SRR3177976, SRR3177973, SRR3177971, SRR3177975, SRR3177997, SRR3177972, SRR3177999, SRR3187399, SRR3178004, SRR3177982, SRR3177994, SRR3177986, SRR3177987, SRR3177998, SRR3177985, SRR3177988, SRR3177995, SRR3177996, SRR3178000, SRR3177991, SRR3177992, SRR3177993, SRR3177989, SRR3178006, SRR3178076, SRR3136982, SRR3145111, SRR3145112, SRR3177771, SRR3177770, SRR3143858, SRR3144016, SRR3177772, SRR3170537, SRR3170538, SRR3170539, SRR3170541, SRR3177774, SRR3177780, SRR3177776, SRR3177777, SRR3170557, SRR3177778, SRR3177783, SRR3170550, SRR3177773, SRR3170551, SRR3177781, SRR3170549, SRR3177784, SRR3170544, SRR3177787, SRR3170548, SRR3170542, SRR3177819, SRR3170543, SRR3170547, SRR3177810, SRR3177785, SRR3170545, SRR3177786, SRR3177791, SRR3170546, SRR3177788, SRR3170552, SRR3177789, SRR3170553, SRR3170555, SRR3177792, SRR3177793, SRR3177794, SRR3177798, SRR3170554, SRR3177795, SRR3177796, SRR3177800, SRR3177797, SRR3177804, SRR3177801, SRR3177802, SRR3177808, SRR3177806, SRR3177809, SRR3177813, SRR3177807, SRR3177811, SRR3177814, SRR3177830, SRR3170556, SRR3177815, SRR3177818, SRR3177817, SRR3148169, SRR3148170, SRR3150340, SRR3150338, SRR3150349, SRR3150339, SRR3150351, SRR3177820, SRR3177821, SRR3150350, SRR3177823, SRR3150352, SRR3177822, SRR3177824, SRR3150357, SRR3150353, SRR3150354, SRR3177825, SRR3177828, SRR3177827, SRR3177829, SRR3150368, SRR3150356, SRR3150355, SRR3150382, SRR3177831, SRR3177832, SRR3150384, SRR3150385, SRR3177833, SRR3177834, SRR3177836, SRR3150386, SRR3177837, SRR3177838, SRR3150387, SRR3178002, SRR3177839, SRR3177840, SRR3177842, SRR3150388, SRR3150389, SRR3177844, SRR3150390, SRR3150391, SRR3177847, SRR3177845, SRR3177843, SRR3150456, SRR3177848, SRR3177850, SRR3177849, SRR3177851, SRR3150498, SRR3177852, SRR3177853, SRR3177854, SRR3177855, SRR3177856, SRR3177860, SRR3177864, SRR3177863, SRR3150392, SRR3177857, SRR3150393, SRR3177859, SRR3177866, SRR3177868, SRR3150403, SRR3150402, SRR3177865, SRR3177867, SRR3150404, SRR3177870, SRR3150479, SRR3150405, SRR3177869, SRR3177896, SRR3177871, SRR3150426, SRR3150428, SRR3177874, SRR3150427, SRR3150429, SRR3177875, SRR3177872, SRR3150430, SRR3150500, SRR3150431, SRR3177876, SRR3150432, SRR3177877, SRR3177878, SRR3150433, SRR3177880, SRR3150455, SRR3150435, SRR3150501, SRR3150534, SRR3177881, SRR3150499, SRR3177882, SRR3150502, SRR3150503, SRR3150504, SRR3150505, SRR3177884, SRR3177883, SRR3177888, SRR3177886, SRR3177887, SRR3177885, SRR3150506, SRR3150507, SRR3177889, SRR3150508, SRR3150509, SRR3177894, SRR3150510, SRR3150511, SRR3150512, SRR3177891, SRR3150513, SRR3177892, SRR3150514, SRR3150515, SRR3150516, SRR3150518, SRR3150520,

SRR3177924, SRR3150517, SRR3177925, SRR3177895, SRR3177898, SRR3150519, SRR3177902, SRR3150521, SRR3177899, SRR3177897, SRR3150523, SRR3392497, SRR3177904, SRR3177905, SRR3177906, SRR3150522, SRR3177908, SRR3177907, SRR3150526, SRR3150527, SRR3150524, SRR3150529, SRR3150525, SRR3150528, SRR3150551, SRR3150532, SRR3177909, SRR3150552, SRR3150530, SRR3150531, SRR3150553, SRR3150533, SRR3150554, SRR3150536, SRR3150535, SRR3150537, SRR3150538, SRR3150539, SRR3150540, SRR3150541, SRR3150543, SRR3150542, SRR3150544, SRR3150546, SRR3150545, SRR3150547, SRR3150548, SRR3150550, SRR3150725, SRR3150701, SRR3150549, SRR3150736, SRR3150702, SRR3150714, SRR3150737, SRR3150713, SRR3177910, SRR3150799, SRR3150738, SRR3177911, SRR3177915, SRR3150794, SRR3150739, SRR3150740, SRR3177912, SRR3150742, SRR3150741, SRR3150793, SRR3150796, SRR3150797, SRR3150795, SRR3177913, SRR3150798, SRR3150800, SRR3150801, SRR3150804, SRR3150802, SRR3150803, SRR3150806, SRR3150805, SRR3150820, SRR3150819, SRR3150808, SRR3150807, SRR3150821, SRR3150809, SRR3150811, SRR3150810, SRR3150816, SRR3150812, SRR3150813, SRR3150817, SRR3150822, SRR3150815, SRR3150818, SRR3150824, SRR3150814, SRR3150857, SRR3150826, SRR3150827, SRR3150828, SRR3150830, SRR3150829, SRR3150831, SRR3150833, SRR3150832, SRR3150834, SRR3150835, SRR3150836, SRR3150837, SRR3150858, SRR3150839, SRR3150838, SRR3150840, SRR3150841, SRR3150842, SRR3150847, SRR3150843, SRR3150844, SRR3150845, SRR3177914, SRR3150848, SRR3177920, SRR3150849, SRR3150851, SRR3150852, SRR3150850, SRR3150853, SRR3150854, SRR3150855, SRR3150856, SRR3151957, SRR3151960, SRR3151958, SRR3151972, SRR3151953, SRR3151954, SRR3151952, SRR3151955, SRR3151961, SRR3151963, SRR3151956, SRR3151964, SRR3151965, SRR3151959, SRR3151966, SRR3151962, SRR3151970, SRR3151976, SRR3151968, SRR3151980, SRR3151973, SRR3151996, SRR3151978, SRR3151971, SRR3151967, SRR3151977, SRR3151974, SRR3152022, SRR3151969, SRR3151982, SRR3151975, SRR3151985, SRR3151979, SRR3151999, SRR3151981, SRR3151997, SRR3151983, SRR3152000, SRR3151988, SRR3151986, SRR3151993, SRR3177916, SRR3151989, SRR3151987, SRR3177918, SRR3151994, SRR3151990, SRR3151984, SRR3151991, SRR3151992, SRR3151998, SRR3151995, SRR3152029, SRR3152030, SRR3152034, SRR3152031, SRR3152032, SRR3152040, SRR3152041, SRR3152035, SRR3152044, SRR3152033, SRR3177921, SRR3152036, SRR3152042, SRR3152037, SRR3152038, SRR3152045, SRR3152039, SRR3152046, SRR3152043, SRR3152078, SRR3152055, SRR3152047, SRR3152048, SRR3152049, SRR3152050, SRR3152051, SRR3177922, SRR3177923, SRR3152059, SRR3152052, SRR3152053, SRR3177938, SRR3152057, SRR3152069, SRR3152056, SRR3152070, SRR3152054, SRR3152079, SRR3152060, SRR3152058, SRR3152061, SRR3177940, SRR3152075, SRR3152062, SRR3152063, SRR3152065, SRR3152067, SRR3152064, SRR3152084, SRR3152066, SRR3152074, SRR3152068, SRR3152071, SRR3152076, SRR3152072, SRR3152073, SRR3152081, SRR3152077, SRR3152085, SRR3152083, SRR3152082, SRR3152086, SRR3152087, SRR3152092, SRR3152088, SRR3152100, SRR3152093, SRR3152089, SRR3152094, SRR3152090, SRR3152091, SRR3152095, SRR3152096, SRR3152097, SRR3152098, SRR3177939, SRR3152101, SRR3152099, SRR3152106, SRR3152116, SRR3152113, SRR3152102, SRR3152103,

SRR3152105, SRR3152104, SRR3152111, SRR3152108, SRR3152107, SRR3152109, SRR3152110, SRR3152139, SRR3152140, SRR3152112, SRR3152115, SRR3152117, SRR3152114, SRR3152130, SRR3152118, SRR3152119, SRR3152120, SRR3177942, SRR3152122, SRR3152123, SRR3152124, SRR3152125, SRR3152121, SRR3152127, SRR3152129, SRR3152126, SRR3152145, SRR3177944, SRR3152131, SRR3152128, SRR3152133, SRR3177941, SRR3152137, SRR3152132, SRR3152135, SRR3152134, SRR3152149, SRR3152138, SRR3177943, SRR3152144, SRR3152146, SRR3152136, SRR3152154, SRR3152155, SRR3152156, SRR3152153, SRR3152141, SRR3152151, SRR3152142, SRR3152143, SRR3177945, SRR3152147, SRR3152152, SRR3152148, SRR3152150, SRR3178075

## A.2 Selected subset of the dataset

We selected the following [SRA](#) IDs to use in our experiments: SRR1518985, SRR2006285, SRR1518456, SRR1518469, SRR1518470, SRR1518579, SRR1518648, SRR1518939, SRR1518659, SRR1518661, SRR1518964, SRR3136954, SRR2085795, SRR1922810, SRR1923031, SRR1922562, SRR1922814, SRR1763644, SRR1919631, SRR1922877, SRR1919983, SRR2096800, SRR2093843, SRR2088777, SRR2088776, SRR3177954, SRR3177964, SRR3178076, SRR3145111, SRR3170551, SRR3170552, SRR3177802, SRR3148170, SRR3150350, SRR3177836, SRR3177885, SRR3392497, SRR3177905, SRR3150539, SRR3150542, SRR3150809, SRR3152022, SRR3151969, SRR3152035, SRR3152033, SRR3152074, SRR3152076, SRR3152085, SRR3152105, SRR3152110