Reproducible and Portable Big Data Analytics in the Cloud

Xin Wang, Pei Guo, Xingyan Li,
Aryya Gangopadhyay, Carl E. Busart, Jade Freeman, and Jianwu Wang*

Abstract—Cloud computing has become a major approach to help reproduce computational experiments. Yet there are still two main difficulties in reproducing batch based big data analytics (including descriptive and predictive analytics) in the cloud. The first is how to automate end-to-end scalable execution of analytics including distributed environment provisioning, analytics pipeline description, parallel execution, and resource termination. The second is that an application developed for one cloud is difficult to be reproduced in another cloud, a.k.a. vendor lock-in problem. To tackle these problems, we leverage serverless computing and containerization techniques for automated scalable execution and reproducibility, and utilize the adapter design pattern to enable application portability and reproducibility across different clouds. We propose and develop an open-source toolkit that supports 1) fully automated end-to-end execution and reproduction via a single command, 2) automated data and configuration storage for each execution, 3) flexible client modes based on user preferences, 4) execution history query, and 5) simple reproduction of existing executions in the same environment or a different environment. We did extensive experiments on both AWS and Azure using four big data analytics applications that run on virtual CPU/GPU clusters. The experiments show our toolkit can achieve good execution performance, scalability, and efficient reproducibility for cloud-based big data analytics.

Index Terms—Reproducibility, Cloud computing, Portability, Serverless, Big data analytics.

1 Introduction

Reproducibility is increasingly required by the research community, funding agencies, and publishers [1]. By reproducing an existing computational experiment and obtaining consistent results, we can have more confidence in the research. Further, besides reproducing the exact process, it is also valuable to explore how the experiment behaves with different input datasets, execution arguments, and environments. Cloud computing has been a major approach for reproducibility [2] because cloud services can be leveraged to provision data, software, or hardware needed in reproduction. For instance, paper [3] summarized 13 aspects that cloud computing can help with reproducibility.

In this paper, we mainly address the following challenges in cloud-based reproducibility. First, it is still difficult to achieve end-to-end automated big data analytics execution and reproduction in the cloud. The end-to-end automation should support scale-up and scale-out of distributed hardware environment, software environment provisioning, data and configuration storage for each execution, resource termination after execution, execution history query and reproducibility of existing executions in the same environment or a different cloud environment. Second, because the services provided by each service provider such as AWS and Azure are proprietary, an application developed for one cloud cannot run in another cloud, which is a well-known vendor lock-in challenge. Two scientific problems to be studied by tackling challenges are: 1) what is a proper abstraction and design for better reproducibility support from both user and toolkit perspectives, 2) what is a more efficient way to achieve cloud-based reproducibility for big data analytics. We note our work only supports batch based processing big data analytics jobs, including descriptive and predictive analytics, not interactive jobs like database queries.

Based on the above challenges and scientific problems, we propose an approach and corresponding open-source toolkit [4] for Reproducible and Portable big data Analytics in the Cloud (RPAC). Our contributions are summarized as follows.

- Our proposed approach and toolkit integrate serverless computing techniques to automate end-to-end batch based big data analytics execution. Tasks of big data analytics execution (resource provisioning, application execution, data storage and resource termination) are encapsulated as cloud functions and automatically triggered by proper events. With the full automation support, users can re-run the exact execution or run the application with different configurations, including different scale-out and scale-up factors, via only one command. Our RPAC toolkit supports both AWS and Azure cloud environments.
- For easy reproducibility, we make proper data modeling and abstraction. It first separates essential information required for reproducibility and detailed information required by each cloud provider. Fol-

- *Corresponding author
- X. Wang, P. Guo, X. Li, A. Gangopadhyay and J. Wang are with the Department of Information Systems, University of Maryland, Baltimore County, Baltimore, MD, 21250.
  E-mail: {xinwang11, peigu01, xingyanli, gangopad, jianwu}@umbc.edu.
- C. Busart and J. Freeman are with the DEVCOM Army Research Laboratory, Adelphi, MD, 20783.
  E-mail: {carl.e.busart.civ, jade.l.freeman2.civ}@army.mil.

lowing the separation of concerns principle, it further separates the essential information into three categories (resources, application, personal) for easy reconfiguration. The essential information will also be automatically stored in the cloud by our toolkit as authentic recording of the execution. Later, the storage URL can be published and shared as the single source to reproduce the historical execution.
- To deal with the vendor lock-in challenge, on top of the above abstractions, we propose a Cloud Agnostic Application Model (CAAM) to support execution and reproducibility portability with different cloud providers. CAAM abstracts the application out of its cloud specific logic, and allows reproducing executions in another cloud via only minimal configuration changes from the user.
- We benchmark both CPU-based and GPU-based big data analytics applications using our RPAC toolkit. We measure the overhead of data storing for reproducibility. We also did extensive experiments to benchmark three applications on different cloud providers in terms of execution performance, scalability and reproducibility efficiency.

The rest of the paper is organized as follows. In Section 2, we briefly introduce related techniques our work is built on. Section 3 provides an overview of our proposed approach. Three main parts of our approach, namely data modeling, automated execution and reproduction of big data analytics in the cloud are explained in Section 4, Section 5 and Section 6, respectively. Experiments and benchmarking results are discussed in Section 7. We compare our work with related studies in 8 and conclude in Section 9.

2 Background

2.1 Big Data Analytics

To deal with increasing data volumes in data analytics, many platforms have been proposed to achieve parallelization of the analytics in a distributed environment. We explain three such platforms that our work is built on for reproducibility. As one of the most popular big data platform, Spark [5] follows and extends the MapReduce paradigm [6] and achieve parallelism by distributing input data among many parallel tasks of the same function. To run an application, Spark employs a master process on one node and a worker process on each of other nodes so the worker processes can take tasks from the master process and run them in parallel. Similar to Spark, a Dask [7] application is composed as a task graph that can be distributed within one computer or a distributed computing environment. Dask employs a similar master-worker framework for task scheduling. Horovod [8], as a popular software framework for distributed learning, provides data parallel deep learning optimized for GPU-based data analytics. For coordinating execution between distributed processes on GPU, Horovod can use Message Passing Interface (MPI) for communicating data with high performance. The CUDA-aware MPI is commonly used in HPC to build applications that can scale to multi-node computer clusters [9].

2.2 Reproducibility

There have been many definitions of reproducibility and similar terms like replicability and repeatability [10, 11]. Unfortunately, these definitions are not very consistent, some even contradict with each other [1]. Here, we simply define reproducibility as a capability that obtains consistent results using the same computational steps, methods, and code. As paper [12] said, containerization is one of the valid and common solutions for the reproducible software deployment problem of scientific pipelines. For cloud-based reproducibility, it studies how to re-execute an existing application in the cloud [3]. We categorize reproducibility support into four ways: 1) rerun exactly the same application with the same hardware and software environment, 2) reproduce with a different application configuration to know how the application performs with different datasets or arguments, 3) reproduce with different cloud provider hardware environment (virtual machine type and number, etc.) within the same cloud provider to test scale-up and scale-out; and 4) reproduce with a different cloud provider to avoid vendor lock-in problem. Our toolkit is built to support all four types of reproduction.

2.3 Serverless Computing

As a recent cloud-based execution model, serverless computing provides a few advantages. First, it responds to user service requests without maintaining back-end servers in the cloud. Second, it employs Function as a Service (FaaS) architecture that allows customers to develop separate functions directly rather than standalone cloud applications. As explained in [13], each application logic/pipeline is split into functions and application execution is based on internal or external events. All major cloud providers offer serverless services, including AWS Lambda, Azure Functions and Google Cloud Functions.

3 Overview of Reproducible and Portable Data Analytics in the Cloud

In this section, we provide an overview of how our proposed approach achieves reproducible and portable data analytics in the cloud. With the approach and corresponding open-source toolkit RPAC for reproducible and portable data analytics in the cloud, users can easily re-run previous experiments with the same or different setups including environments, application arguments, input data and cloud providers. Our approach is built on top of serverless computing and we adopt a new way of utilizing serverless computing for large scale computations. So we will explain first how to use serverless large scale computations, then how to use serverless for big data analytics reproducibility.

3.1 Serverless based Reproducibility

As shown in Fig. 1, our proposed approach has two parts: 1) first execution of an application, and 2) reproduction of the existing execution from historical configurations. Both first execution and reproduction are automated via serverless-based approach shown on the right.

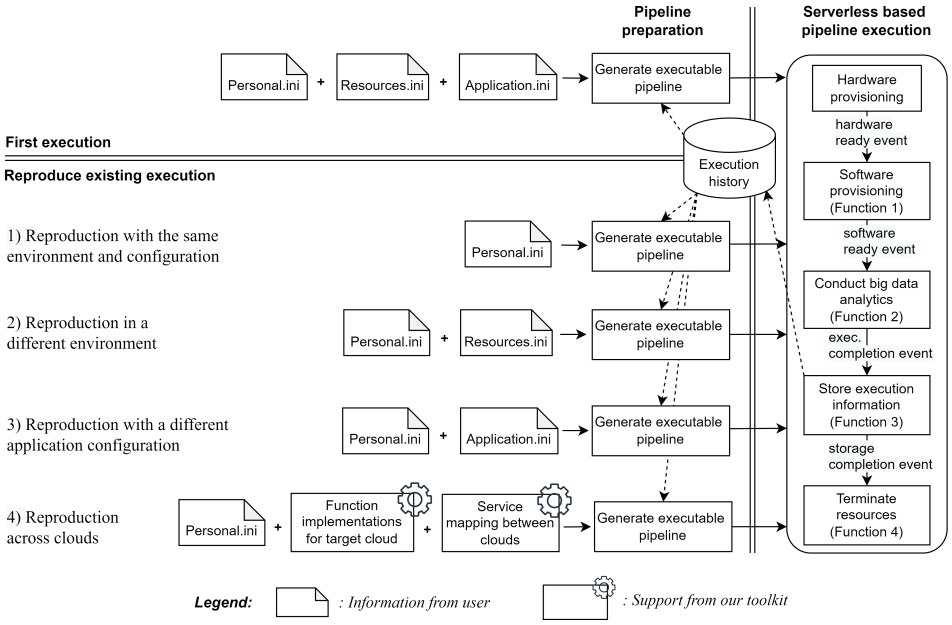


Fig. 1: The overview of our proposed approach for reproducible and portable data analytics in the cloud.

In the beginning, there is no execution history for querying and reproducing. Clients need to prepare configurations to generate the pipeline file for the whole execution. The configuration includes all configurable setups, for example, the application-based information like application programs, arguments, input data, and the cloud-based information like virtual cluster type, size, network setting, memory, with personal credentials. Our toolkit will take this information to create an executable pipeline for a target cloud. With this pipeline, the data analytics application will execute in the cloud environment, output its results to the storage, and automatically terminate resources once the execution finishes. We will explain in detail how we leverage serverless techniques for automated big data analytics in Section 5.

After an application is executed, clients can reproduce it based on its execution history. Our RPAC toolkit will generate a pipeline file based on the execution history and reproduction configurations. If the client wants to reproduce an existing execution with the exact environment and configuration, the pipeline file within the execution history can be used directly by our toolkit for reproducibility. If the client chooses to reproduce existing execution within the same cloud, but with a different environment or application, our toolkit will combine changed configurations of cloud resources or applications with the historical execution information to generate a new pipeline file. If the client prefers reproducing existing execution on a different cloud, our toolkit will provide cloud service mapping and implementations of functions in the target cloud. With user-provided personal information and historical execution, a new pipeline will be generated for the target cloud. Finally, with the pipeline file executable by cloud serverless services, the data analytics will be reproduced in cloud automatically. Details of how our approach achieves reproducibility will be explained in Section 6.

We would like to note the serverless pipeline used here is different from most other workflow or pipeline definitions such as [12][14][15][16]. These definitions only include the processing steps and their dependencies. They do not describe how to provision hardware and software environments because they assume these environments are ready before pipeline execution. Our serverless pipeline includes the full execution life cycle including hardware and software provisioning, big data analytics, execution export and resource release. Our pipeline does not describe internal processing steps, but could be integrated with traditional pipelines as internal logic description in its Function 2: conduct big data analytics.

3.2 Serverless based Large Scale Application in the Cloud

Traditionally, serverless computing is used to execute serverless pipelines and the functions defined in each pipeline directly via cloud services like AWS CloudFormation. In this case, the computation is executed following the pipeline without using any additional cloud resources. Because of the memory and CPU limit for serverless functions, this approach can only handle computations whose resource requirements are light. For instance, OpenWhisk [17] is an event-based serverless computing cloud platform, which allows users to implement their own OpenWhisk APIs for the connections between the event source and trigger, the trigger rule, and the computation actions.

Different from the above way to use serverless, we leverage serverless computing and its FaaS to achieve reproducibility for big data analytics in the cloud. The main difference is that we use the serverless pipeline as a way to orchestrate and manage additional cloud resources for heavy workloads while each step is wrapped as a function. In this way, both the serverless pipeline and its functions do not execute heavy commands directly. Instead, each function's execution only submits commands from serverless to the additional cloud resources. Then when the function is triggered, the commands will be transferred to the additional cloud services and be executed as background processes so they can return without waiting for the finish

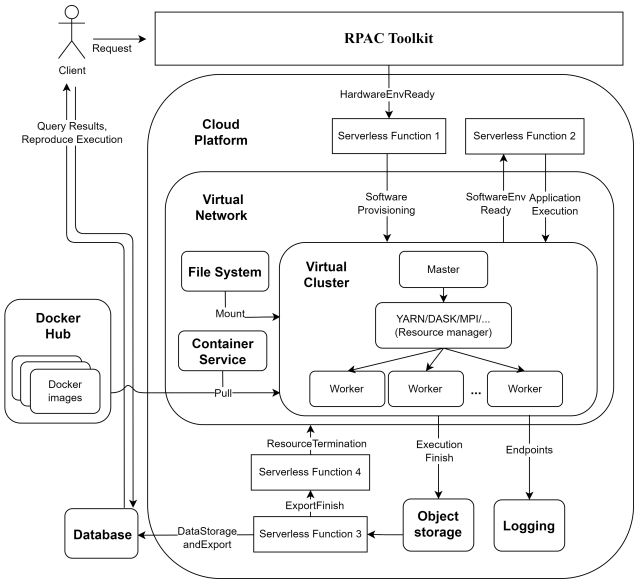


Fig. 2: The interaction between our proposed toolkit and cloud resources.

of the commands. As shown in Fig. 2, the serverless pipeline listens to events sent by our toolkit or other cloud services. By mapping the triggers mentioned in the event with the trigger rule associated with each serverless function, it knows which serverless function will be involved based on the received event. For instance, Function 2 will be triggered when the pipeline receives an event as SoftwareEnvReady. For each function's execution, it only submits commands from serverless to additional cloud services such as AWS EC2, so the resource and time limits for serverless functions will not be violated. Also, all major cloud providers including AWS and Azure, only enforce time limits for serverless functions, not serverless pipelines. So serverless pipelines are capable of large-scale computations that might take a long time. Serverless pipeline is also a reasonable choice from a budgetary cost perspective because serverless service is charged by the number of function invocations and the duration it takes to execute, not the deployment time of the pipeline.

4 Data Modeling and Storage for Reproducibility

To achieve easy configurability by users and future reproducibility across cloud providers, we categorize data based on their usage and employ different levels of data abstraction. Specifically, the data model contains three parts: abstract request information, executable request information and execution history information. We believe the data model can serve as a reference model for different reproducibility toolkits.

4.1 Abstract Request Information

To avoid learning specific specifications and templates for specific clouds, we extract minimal information a user has to provide for application execution or reproduction. Further, as shown in the upper part of Fig. 3, we categorize the information into three separate key-value based configuration files where `ini` is used as file extension to distinguish them from other file types used by our toolkit. Specifically, `resources.ini` stores hardware and software resources information such as virtual machine type, virtual instance number, docker image URL and big data engine; `application.ini` records the program URI of the application, program arguments, and input dataset URIs of the program; `personal.ini` contains the cloud credential information such as SSH key location and cloud credential info (which can also be provided at runtime for security concerns). We use three different files so only a subset of files needs to be edited for each type of reproducibility shown in Fig. 1. A complete and formal listing of the information can be found at Fig. 4 using syntax of Backus-naur form (BNF) [18].

4.2 Executable Request Information

We separate information that is required for actual cloud-based application execution into four files and use `json` as the file extension. Such files have to follow specifications set by each cloud. For such files, our RPAC toolkit generates them automatically based on corresponding abstract `ini` file(s) mentioned above. The first file is `resources.json` which describes hardware and software environment info. This file has to be changed if the cloud provider is switched. The `resources.json` will be generated based on the above `resources.ini` file, the cloud type and the type of big data analytics. Another file is `application.json` which contains application specific information and will be generated by our toolkit based on the above `application.ini` file and the cloud type. Similarly, `personal.json` can be generated from `personal.ini`. As shown in Fig. 3, by combining `resources.json`, `application.json`, `personal.json` and four cloud-specific serverless functions shown in Fig. 1, we get `pipeline.json` that describes the execution logic of the serverless application. Our RPAC toolkit contains template `json` files and serverless function implementations so they can be reused for different data analytics

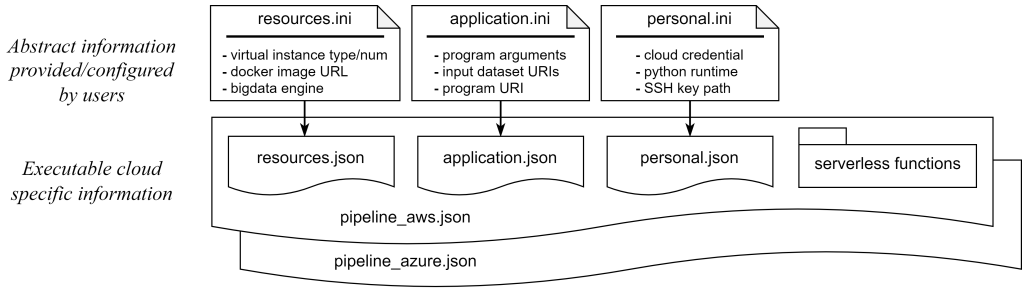


Fig. 3: Data modeling and abstraction for reproducibility.

<abstract_req_4_big_data_analytics> ::= $< resources >, < personal >, < application >$
$< resources >$ ::= $< bigdata\_engine >, < cloud.aws > | < cloud.azure >, < reproduce >$
$< bigdata\_engine >$::= $"none" | "spark" | "horovod" | "dask"$
$< cloud.aws >$::=$< region >, < instance\_number >, < subnet\_id >, < instance\_type >, < vpc\_id >$
$< cloud.azure >$::=$< region >, < instance\_number >, < resource\_group\_name >, < instance\_type >$
$< reproduce >$::=$< reproduce\_storage >, < reproduce\_database >$
$< personal >$::=$< cloud\_provider >, < key\_path >, < key\_name >, < python\_runtime >, < cloud\_credentials >$
$< application >$::=$< docker\_image >, < data\_uri >, < command >, < bootstrap >$

Fig. 4: Normative form for abstract request information.

applications. We illustrate how to map from abstract requirements in Section 4.1 to an executable cloud specific serverless pipeline and its implemented functions in Fig. 5. The pipeline file is generated by the three abstract request information provided by users, which transfers the stateless configurations to executable cloud-specific information. The abstract request information is first transferred to the executable request information while missing parameters can be filled with their default values. All parameters in the executable request information are also sorted out based on the cloud-specific schema. By combining with corresponding serverless functions, the cloud-specific executable request information, like `pipeline_aws.json`, will be generated and executed in our RPAC toolkit. Each serverless function listens to the upcoming events. If a received event (e.g.,*HardwareEnvReady* in Fig. 5) matches, the associated function (e.g., *SoftwareEnvSetup()* in Fig. 5) will be triggered. At the end of the function execution, a new event (e.g., *SoftwareEnvReady* in Fig. 5) will be returned to trigger the downstream functions.

The differences between the two types of request information are summarized below. Abstract request information, as a user-friendly abstraction, contains the minimal information a user has to provide for application execution or reproduction. In comparison, the executable request information describes the execution logic of the serverless application, which is required for actual cloud-based application execution. Our RPAC toolkit will generate the executable request information based on the corresponding abstract information during the execution.

Next, we will explain how the files are used for automated execution in detail in Section 5.1 and how they are reused or transformed for reproduction in Section 6.

4.3 Execution History Information

Execution history information is critical to share each execution for later analysis and reproduction. As illustrated in Fig. 6, we classify execution history related data into three categories and store them separately. The first category is execution log metadata, such as timestamps, duration, cost, and status, which are stored in the database for query. Key-value based execution parameters including analytics command line and arguments are also stored for easy comparison among executions. This metadata information is unique for each execution, not required for reproducibility, but useful for later analysis such as finding the fastest execution time of the same application on different clouds or cloud resources. For information that can be referred from external resources, such as input datasets, output files and configuration files used for the execution, only their URLs are stored in the database. The second category is object based storage of each execution information for reproducibility. Two items are stored for each execution: 1) abstract request information (`resources.ini`, `personal.ini` and `application.ini` in Config.zip), 2) execution output datasets in Result.zip. Only abstract request information, not cloud specific information, is stored by our RPAC toolkit in order to minimize storage overhead. These data are compressed, categorized and stored in cloud object storage services such as AWS S3 and Azure Blob storage so a unique URL could be obtained for each execution. Because the data has the complete

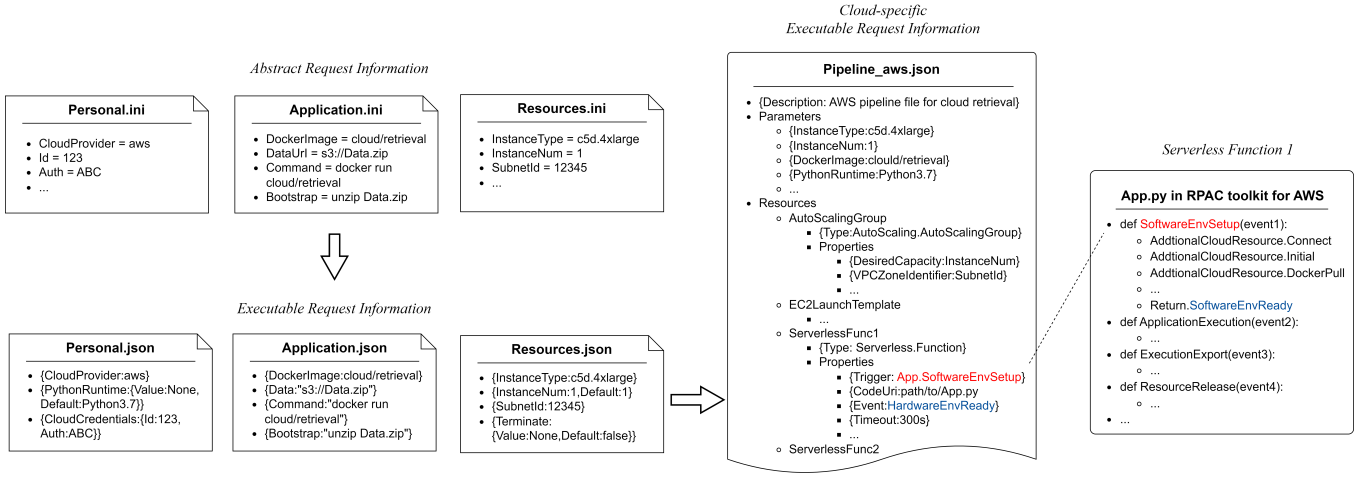
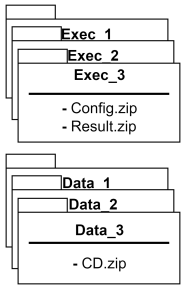


Fig. 5: Mapping from abstract requirement to executable cloud specific serverless pipeline and functions.

Individual object storage of detailed execution history for reproducibility

Database_Table										
id	ts	para	app	dura	cost	ratio	config	data	result	status
0	1632196231	{k1:v1, ...}	*Domain Adaptation*	1.23	4.56	7.89	s3://Exec_1/Config.zip	s3://Data_1/DA.zip	s3://Exec_1/Result.zip	{Success}
1	1639385410	{k1:v1, ...}	*Cloud Retrieval*	1.23	4.56	7.89	s3://Exec_2/Config.zip	s3://Data_2/CR.zip	s3://Exec_2/Result.zip	{Success}
2	1642045810	{k1:v1, ...}	*Causality Discovery*	1.23	4.56	7.89	s3://Exec_3/Config.zip	s3://Data_3/CD.zip	s3://Exec_3/Result.zip	{Fail:ErrorMessage}

Database based meta-data storage of execution history for future analysis



Shared object storage of datasets

Fig. 6: Data modeling of execution history information.

information to achieve reproducibility, the URL could be easily published as public records following the Research Object framework [19] so it can be referred to via a DOI identifier later as the single source for reproducibility. The third category is shared object storage of input datasets. It is stored separately so that multiple executions with the same input data only need one object storage. Also, cloud storage services like AWS S3 and Azure Blob storage allow automatic versioning so minor changes of input datasets do not require a fully separate storage.

5 Automated Big Data Analytics in the Cloud towards Reproducibility

To achieve easy reproducibility, the execution should be as automated as possible to minimize manual operations during reproduction phase. Also, an execution should be easily configurable for different scalability factors, application parameters, even cloud providers. In this section, we discuss our techniques to achieve fully automated big data analytics in the cloud so an application can be executed and later reproduced using only one command.

5.1 Serverless and Docker-based Execution Automation

We leverage serverless computing to achieve overall analytics pipeline description and execution, and docker for software environment setup. Serverless computing offers a few advantages for reproducibility: 1) it saves costs because we do not need to maintain a server in the cloud especially for cases reproduction does not happen frequently; 2) its FaaS model allows us to design and implement separate functions required for automated execution/reproduction; 3) its event-based function composition and execution eliminates the requirement of a separate workflow/pipeline software which is needed for many traditional workflow-based reproducibility [1].

As explained in Section 2.3, serverless computing offers templates to describe cloud service resources required by the application, structured application pipeline, and event-based execution. Each component in the application pipeline is implemented as a serverless function and triggered by the events it listens to. So the pipeline binds cloud services with the specific event in order to trigger the corresponding serverless function. In addition, we can package complicated software dependencies required for an application via docker. The details of the automation are illustrated in Fig. 7.

After receiving the user request, RPAC execution automation starts with pipeline generation and submission. Based on configurations, RPAC generates corresponding pipeline files, deploys its serverless functions, and uploads these configurations to the storage except client personal information. RPAC then submits this pipeline to the cloud and starts serverless execution. The serverless pipeline starts with the on-demand hardware environment provisioning (step a in Fig. 7) via cloud manager services (such as CloudFormation for AWS and Deployment Manager for Azure). The hardware provisioning is more like an on-demand resource request service that is a prerequirement for all serverless functions. So we put the hardware provisioning at the beginning of the serverless pipeline. To conduct big data analytics, we also need to create a virtual cluster by specifying the type and sub-type of virtual machines, the number of virtual machines, network security groups, etc. Cloud manager services allow the information to be submitted based on their semi-structured specification such as JSON and YAML. RPAC will send a reply once the pipeline file is submitted.

The remaining steps of the automated pipeline execution are done via four cloud functions. On top of the virtual hardware environment provisioned, the next automation step is to deploy the required software to run the application (step b in Fig. 7). It is achieved by the first serverless function, which pulls required docker file and starts it. After the hardware and software environments are provisioned, it is ready to execute applications. The second serverless function in Fig. 7 executes the application by deploying user application (e.g., download application codes and unzip them) and running its commands with proper parameters (input data, application specific arguments, etc.). The third function exports all addresses of stored files to cloud database and object storage for future query and reproduction. After the storage completes, a termination event is sent to the last function, which terminates all cloud resources. At this time, the whole pipeline is fully executed, and the client is able to check and query information stored in the database and object storage.

All these functions are triggered automatically when they receive corresponding events. The cloud manager services mentioned before can help the client manually send

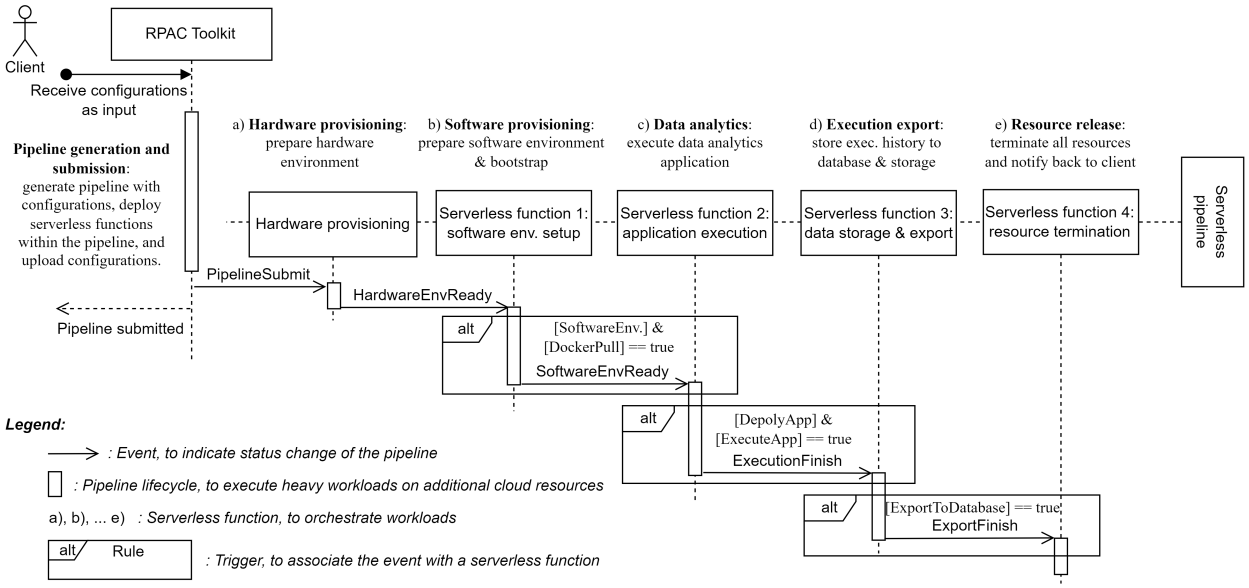


Fig. 7: The system sequence diagram for automated execution of big data analytics in serverless framework.

events to the serverless function from cloud console. In order to achieve full automation, these events can also be delivered to the target function by cloud event handling services (such as EventBridge for AWS and Event Grid for Azure) using a pre-defined event rule. Each serverless function needs to set up an event rule which specifies what type/property of event can trigger this function. For example, the rule of execution export function (step d in Fig. 7) requires the event source from object storage with a rule-defined prefix, like *export*.

Besides serverless-based execution, our RPAC toolkit also supports cloud SDK-based execution to allow flexible client modes. Their differences are summarized in TABLE 1. The cloud SDK mode is designed based on the cloud-specific software development toolkit (SDK). SDK facilitates the creation of applications by having a compiler, debugger and a software framework based on its functionality. The implementation of this SDK-based mode contains cloud application programming interfaces (APIs) for pipeline management. For example, AWS Boto python SDK can be invoked to describe the status of EC2 using `ec2.describe_instances()`. The execution can be automated by a periodical status pulling loop. This SDK-based mode requires programming knowledge and a complete understanding of the data analytics pipeline, so that developers become preferred users since they can either run the application in a fully automated way or step-wise execution for debugging purposes. By supporting different execution modes, users can make flexible choices. In comparison, serverless based approach is fully automated and more efficient because only the execution is managed via internal event triggering. No communications between client and cloud are needed once the pipeline is submitted.

5.2 Scalable Execution for Three Parallel Frameworks

In this section, we discuss how our approach supports scalable execution via the three parallel frameworks in Section 2.1, namely Spark-based, Dask-based, and Horovod-based analytics. The first two utilize virtual CPU clusters and the third utilizes virtual GPU clusters. By specifying the virtual machine type and number, cloud services can provision a cluster hardware environment. However, software dependencies, process coordination, and even access permission may differ for different big data analytics. Because of these differences, each framework requires its own `resources.json` and implementation of the first serverless function shown in Fig. 7. To reproduce big data analytics, one important part is to record and reuse original big data engine configurations. Paper [20] uses separate files to record Spark memory configuration for reproducibility. Similar to this approach, we set these configurations by recording the information via command line arguments or original big data engine configuration files. Big data engine's configurations can be modified in application reproduction by users in `application.ini` file, such as changing *–driver-memory 60g –executor-memory 60g* for Spark engine. Additional big data engine configurations are set up via separate files like spark-env.sh in the *$SPARK/conf* folder. Our toolkit supports storing such files in the cloud so they can be reused in reproduction.

Beyond the listed frameworks, the additional parallel frameworks can also be deployed by updating the docker images' address in `application.ini`. RPAC will setup this parallel framework in the second serverless function of Fig. 7 and execute analytics within the new environment.

Spark-based big data analytics on virtual CPU nodes. We provide Spark-based parallel framework via the docker-based Spark engine virtual cluster provisioned by direct cloud services like AWS EMR with additional cloud resources like virtual network, container service and file system. By default setting, the resource manager like YARN NodeManager initiates the environment from a pulled docker image, and allocates one virtual instance as the master while others as workers. With serverless based pipeline execution, our toolkit enables automated execution management on master and execution computation on workers defined by serverless function handlers/implementations.

TABLE 1: Comparison of two execution modes of RPAC toolkit for execution and reproducibility.

Execution mode	Techniques	Automation	Preferred users
Cloud SDK	Use the local machine terminal for data analytics. Implementation for the whole execution with cloud SDK.	Full automation by periodical status pulling from client	Developer
Serverless	Use the cloud specific serverless service for data analytics. Managed by binding trigger with serverless function.	Full automation by serverless event triggering within cloud	End user / Developer

Since big data analytics utilizes many compute nodes with complex computation proprieties, it is important to make sure availability and reliability during cloud execution. To achieve a secure and stable scalable execution, we control the access permission of master and workers by using the network security group. During big data analytics, our pipeline assigns one group for the master and another group for workers, and only enables TCP/UDP inbound and outbound rules within them. Also, for computation reliability, the big data analytics pipeline only allows client SSH permission for the master security group.

Dask-based big data analytics on virtual CPU nodes. Besides Spark, our RPAC toolkit also supports CPU-based parallel analytics by using Dask as the resource manager in the virtual cluster. Different from Spark which has dedicated cloud services (such as EMR in AWS), Dask environment can only be provisioned by regular virtual machine services (such as EC2 in AWS).

Each virtual instance in the cluster initiates one docker container and our pipeline assigns one of the containers to be the Dask scheduler and others to be workers. Same with the security group setup with Spark-based analytics, we divide the client access between scheduler and workers for execution reliability. During execution, different from AWS EMR service which automatically initiates Spark processes after hardware provisioning, our RPAC toolkit needs to start Dask processes on both scheduler and worker containers during software provisioning before executing big data analytics on virtual CPU nodes. Besides, same as Spark-based cloud services, the client can also produce interactive visualizations based on Dask diagnostic dashboard in our framework, by using the public DNS name (public IP) of the scheduler instance with its dashboard port.

Horovod-based big data analytics on virtual GPU nodes. To provide a GPU-based parallel framework, we leverage Horovod and regular virtual machine services for analytics. The RPAC toolkit executes multi-instance GPU-based data analytics within our pre-built Docker containers, involving a shared file system and a customized port number for the SSH daemon. In order to categorize functionality between different instances, we set one of them as the primary worker and others as secondary workers. Within the container, the primary worker runs the MPI parallel command for data analytics execution while secondary workers listen to that specific port.

6 Reproduce Big Data Analytics in the Cloud

In this section, we discuss how to achieve different levels of reproducibility within the same cloud and across different cloud providers. To achieve reproducibility, the user only needs to provide the URL of a historical execution stored in cloud storage (more in Section 4) and her own configurations. We will explain how our framework and RPAC toolkit support different ways of reproducibility summarized in Section 2.2.

6.1 Reproducibility in the Same Cloud

Reproduction with the same environment and configuration. This type of reproducibility is simplest because it is the same with the first execution as long as we can retrieve the information used from execution history. As illustrated by the first item in reproducibility phase of Fig. 1, by retrieving `resources.ini` and `application.ini` from execution history and providing proper `personal.ini`, our RPAC toolkit can rerun the experiment the same way it was executed for the first time.

Reproduction in a different environment. Reproduction in a different environment means the virtual environment configuration needs to be changed from a historical execution, which is often useful for scale-up and scale-out experiments. As illustrated by the second reproducibility item in Fig. 1, a new `resources.ini` needs to be provided explaining the new environment setup (mostly virtual machine type and number). Then our RPAC toolkit can use it to generate a new executable `resources.json` and run the experiment in the same cloud.

Reproduction with a different application configuration. Reproduction with a different application configuration is often useful to run the application with a different dataset and/or application argument. As illustrated by the third reproducibility item in Fig. 1, a new `application.ini` needs to be provided explaining the new application setup. Then our toolkit can use it to generate a new executable `application.json` and run the experiment in the same cloud.

We note that the last two reproductions can be easily combined for the requirements of running an application with different configurations and a different environment. To support it, a new `resources.ini` and a new `application.ini` should be provided.

6.2 Cross-cloud Reproducibility

We discuss how the client achieves reproducibility with a different cloud provider. As illustrated in the fourth way of reproduction in Fig. 1, by providing cloud service mapping and corresponding serverless function implementation, our toolkit can transform the general-purpose configurations in execution history into a new executable pipeline file for another cloud.

To extend the reproducibility to another cloud, by leveraging the adapter pattern [21], we propose a portable Cloud Agnostic Application Model (CAAM) in order to

TABLE 2: Cloud service used by reproducible and portable data analytics.

Service category	Service description	Amazon AWS	Microsoft Azure	Google Cloud
Virtual cluster	Virtual machine cluster that enables to host distributed data analytics engines.	EC2 Auto Scaling /EMR	Virtual Machine Scale Set /HDInsight	Autoscaling Groups /Dataproc
Virtual network	Manage and monitor networking functionality for cloud resources.	VPN	Virtual Network	Virtual Private Cloud
Container service	Store, manage, and secure container images in private or public.	ECR	Azure Container Registry	Artifact Registry
Object storage	Store, manage, and secure any amount of data in storage.	S3	Blob storage	Firebase
Database	Scalable and secure NoSQL cloud database.	DynamoDB	CosmosDB	Firebase Realtime Database
Serverless	Run and manage the application with zero server management.	CloudFormation & Lambda Functions	Deployment Manager & Azure Functions	Cloud Deployment Manager & Cloud Functions
Cloud Python SDK	Easy-to-use interface to access cloud services.	Boto/Boto3	.NET Core	Cloud SDK
Authentication	Provide fine-grained access control for cloud resources.	AWS IAM	Azure IAM	Cloud IAM

Algorithm 1: Cloud Agnostic Application Model (CAAM)

Input: resources.ini, application.ini, personal.ini
Output: pipeline.json

```
class AWS:
    method __Init__(config):
        res ← [config.res, config.para_frame, ServMapping]
        resources.json ← GetAwsRes(res)
        application.json ← GetAwsApp(config.app)
        personal.json ← GetAwsPersonal(config.personal)
    method generate():
        pipeline ← [resources.json, application.json, personal.json]
        return GetAwsPipeline(pipeline)

class Azure:
    method __Init__(config):
        res ← [config.res, config.para_frame, ServMapping]
        resources.json ← GetAzureRes(res)
        application.json ← GetAzureApp(config.app)
        personal.json ← GetAzurePersonal(config.personal)
    method generate():
        pipeline ← [resources.json, application.json, personal.json]
        return GetAzurePipeline(pipeline)

class CloudAdapter:
    method __Init__(AdaptedMethods):
        this.__dict__.update(AdaptedMethods)

function CAAM():
    AdapteeMapping ← [aws : AWS, azure : Azure]
    config ← read(Input)
    if arg.reproduce then
        config.update(arg.reproduce)
    end
    cloud_provider ← config.cloud_provider
    if cloud_provider in AdapteeMapping.keys then
        Adaptee ← AdapteeMapping[cloud_provider]
        element ← (execute : Adaptee(config).generate)
        Pipeline ← CloudAdapter(dict.add(element))
    else
        raise NotImplementedError()
    end
    return Pipeline.execute() to Output
```

solve the vendor lock-in and interoperability problem for big data analytics, which is shown in Algorithm 1. When CAAM receives `resources.ini`, `application.ini` and proper `personal.ini`, `CloudAdapter()` invokes each vendor specific method of different cloud. It means as long as there is an adaptee class written for the cloud provider, by calling the `CloudAdapter()` with this cloud provider, the provided general-purpose configurations will be transformed to the executable request information of the target cloud based on its specification sets and execution requirements. By combining the compatible information of resources, application and personal, CAAM generates the overall executable `pipeline.json` and starts to execute the data analytics.

As shown in Algorithm 1, each cloud adaptee needs to implement how to get its `resources.json` based on `resources.ini` from execution history, parallel framework and service mapping shown in TABLE 2. After all `json` files are ready, AWS uses `GetAwsPipeline(AwsConfig)` to generate pipeline file, while Azure uses `GetAzurePipeline (AzureConfig)` for generation. With CAAM, client directly calls `CloudAdapter()` with a specific adaptee method to execute data analytics with one general-purpose configuration. By calling `Pipeline.execution()`, the `generate()` method in corresponding cloud adaptee will generate the pipeline file and execute the big data analytics in cloud. Particularly, adaptee is in a modular design that can be injected into, removed from, or replaced within CAAM at any time.

Extensibility on cross-cloud reproduction. Our reproducible and portable big data analytics can be easily extended to additional clouds because most services from different cloud providers can be mapped to each other. TABLE 2 lists all cloud services provided by Amazon AWS, Microsoft Azure and Google Cloud for data analytics. Our toolkit currently only implements cross-cloud reproducibility between AWS and Azure. Extension to Google Cloud can be done by adding an additional cloud specific adaptee and providing corresponding service mapping with function implementation.

7 Evaluation

We implement the reproducible and portable cloud computing and open-source it on GitHub at [4]. Two CPU-based analytics applications (cloud retrieval and causality discovery) and one GPU-based analytics application (domain adaptation) are tested in our experiments. All benchmark evaluations are developed on two cloud providers, Amazon AWS and Microsoft Azure. Seven metrics are used to evaluate our work which include data analytics metrics like execution time, budgetary cost, cost-performance ratio, and cloud reproducibility metrics like overhead.

TABLE 3 lists the exact cloud resources we use for each data analytics. For executing the application with a larger dataset, additional storage like AWS Elastic Block Store (EBS) is also been attached during the resource initialization.

TABLE 3: Comparison of cloud resources.

Framework	Metrics	Cloud	Type	vCPU	Memory (GiB)
CPU-based	scale-up	AWS	c5d.4xlarge	16	32
		Azure	F16s_v2	16	32
	scale-out	AWS	c5d.(x)large	2 (4)	4 (8)
		Azure	Fs_v2	2	4
GPU-based	scale-up	AWS	p3.8xlarge	4	16
		Azure	NC24s_v3	4	16
	scale-out	AWS	p3.2xlarge	1	16
		Azure	NC6s_v3	1	16

One variation is in scale-out of AWS CPU-based evaluation. We use c5d.large cluster for Dask-based analytics, but c5d.xlarge cluster for Spark-based analytics because AWS EMR requires more computational capability.

7.1 Benchmark Analytics and Datasets

To benchmark our toolkit's functionality comprehensively, we employ four applications and each uses a separate big data framework in Section 2.1.

Cloud retrieval. Cloud property retrieval is an important task in remote sensing and Atmospheric science. We used the implementation of paper [22] for our first application. It trains a Random Forest machine learning model for cloud mask and cloud thermodynamic-phase retrieval from satellite observations. Dask framework is used for execution parallelization. The Docker image we built is hosted on DockerHub public repository, with Python 3.6 and sklearn 0.24.2. Total datasets are around 0.5 GB.

Causality discovery. In order to discover the cause-effect relationships in a system with the increasing volume and dimensionality of available data, the two-phase scalable and hybrid causality discovery is proposed by Guo et al. [23]. As a big data analytics, we use the Spark application with Hadoop in the cloud virtual cluster. The Docker image we built is hosted on DockerHub public repository, with Python 3.7 and R 3.4. The data in our execution is 200,000 rows of simulated five variable time-series records, which is around 10 MB.

Domain adaptation. Unsupervised Domain Adaptation (UDA) aims to transfer the knowledge learned from a labeled source domain to an unlabeled target domain. We use the UDA implementation designed by Sun et al. [24] that solves the problem of the unlabeled target domain. To move this data analytics to the cloud, we use the virtual cluster with Pytorch GPU acceleration and Horovod with MPI. The Docker image we built is hosted on DockerHub public repository, with Python 3.6, CUDA 10.1 and cuDNN 7. The data we used is the public Office dataset containing 31 object categories in two domains: Amazon and Webcam, which is around 50 MB in total.

Satellite collocation. Because there are many satellites orbiting the Earth, it is valuable to integrate and/or compare their measurements. Satellite collocation provides a way to pair measurements from two satellite sensors that observe the same location quasi-simultaneously. We implemented and parallelized the method in [25] to generate collocated data from two satellites. Like the cloud retrieval application, we use Dask framework for execution parallelization. The Docker image we built is hosted on DockerHub public repository, with Python 3.8, Pandas 1.5.0 and H5py 3.7.0. The two satellites we used in the experiment include the ABI passive sensing data product from NOAA Geostationary Operational Environmental Satellites (GOES-16+) [26] and the CALIOP active sensing data product from NASA Cloud-Aerosol Lidar and Infrared Pathfinder Satellite Observations (CALIPSO) satellite [27]. The total input data volume is 1.1 TB.

7.2 Evaluation Metrics

Even though there have been many studies on reproducibility, as stated in this recent survey paper [28], there are still no agreed metrics that can quantitatively measure reproducibility and compare different reproducible toolkits. The survey paper thinks performance, scalability and efficiency are possible metrics, but no concrete metric definition was provided. In this work, to promote fair comparison, we provide our own definition of performance, scalability and efficiency for cloud based reproducibility, which results in seven metrics (namely *m1* to *m7* listed below).

7.2.1 Execution Performance Metrics

Following paper [29], we measure execution performance of our cloud based application via *m1: execution time*, *m2: budgetary cost* and *m3: performance-price ratio (PPR)*. We first record execution time for each data analytics benchmark. The execution time is the wall-clock time of analytics pipeline (as shown in Fig. 7), which includes pipeline file preparation, cloud resources deployment and initialization, data analytics execution, execution history upload, and termination.

Budgetary cost contains bill usages for all resources used in each data analytics benchmark, which mainly includes the virtual cluster, container, network, database, and object storage with read and write request usage.

Regarding the performance-price ratio (PPR), it evaluates the performance of each analytics considering the execution time with cost. We use the same formula used in [30] for PPR by calculating the product of execution time and budgetary cost. Lower PPR is more desirable excluding other factors.

7.2.2 Cloud Scalability Metrics

We evaluate the scalability of our work for both *m4: scale-up (vertical scaling)* and *m5: scale-out (horizontal scaling)*.

Cloud scale-up is achieved by utilizing more resources within an existing computation system to reach a desired state of performance. In our evaluation, scale-up is set in a single virtual machine by having more threads in Dask-based analytics, more executor cores in Spark-based analytics, or more GPUs in Horovod-based analytics. For cloud retrieval, we fix the number of threads for each Dask worker, and utilize the number of workers from 1 to 8 during evaluation. It is the same in domain adaptation, except by increasing more threads for GPUs rather than CPUs. For causality discovery, because of the EMR setup, we launch only one worker in each virtual instance and allocate only one executor in this worker. To scale up, we use one executor with increasing the numbers of vCPUs of this executor for parallel execution.

In real world scale-up, it is undesired to launch a powerful instance but only use its partial computational capability.

In order to have a fair comparison, we additionally measure scale-up cost by usage, which times the budgetary cost of one instance by the percentage of CPU that is actually used. It simulates scale-up scenarios that use more and more powerful machines.

Cloud scale-out is usually associated with a distributed architecture, which is achieved by adding additional computational capacity to a cluster. In our evaluation, scale-out is set by increasing more virtual machines in an existing cluster. For cloud retrieval and domain adaptation, we deploy only one worker process per instance, and increase the number of instances from 1 to 8 during evaluation. For causality discovery, we instead use one CPU core in each executor, and increase the number of workers by adding virtual instances.

7.2.3 Reproducibility Efficiency Metrics

For reproducibility, a metric *m6: reproducibility_overhead* is used to understand how much overhead it brings by supporting reproducibility during execution. Since reproducibility support is achieved by storing application configuration and execution history, we calculate the ratio between additional execution time caused by reproducibility data storage and the execution time of execution without reproducibility support. The lower the overhead ratio is, the better.

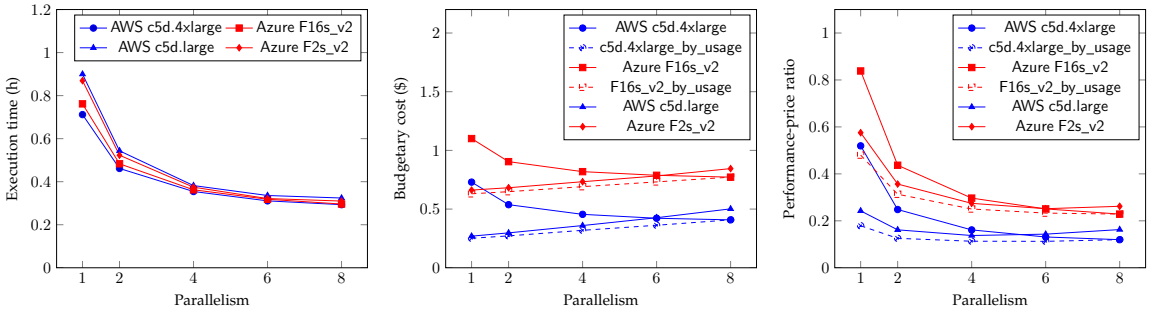
As we mentioned in Section 5.1, an SDK-based pipeline execution mode has also been proposed for big data analytics. Since both SDK-based and serverless-based approaches can be achieved automatically, we also measure *m7: reproducibility_efficiency* to compare their execution time with reproducibility.

7.3 Benchmarking for Execution Performance and Scalability

In this section, we first assess the cloud scalability of our RPAC toolkit based on three metrics: *m1: execution time*, *m2: budgetary cost*, and *m3: performance-price ratio*. In *m4: scale-up* evaluation, the execution is analyzed by gradually utilizing more resources in one instance. In *m5: scale-out*, the evaluation is achieved by gradually adding additional instances of the same type in one cluster. Next, we will explain our benchmarking results of the four applications.

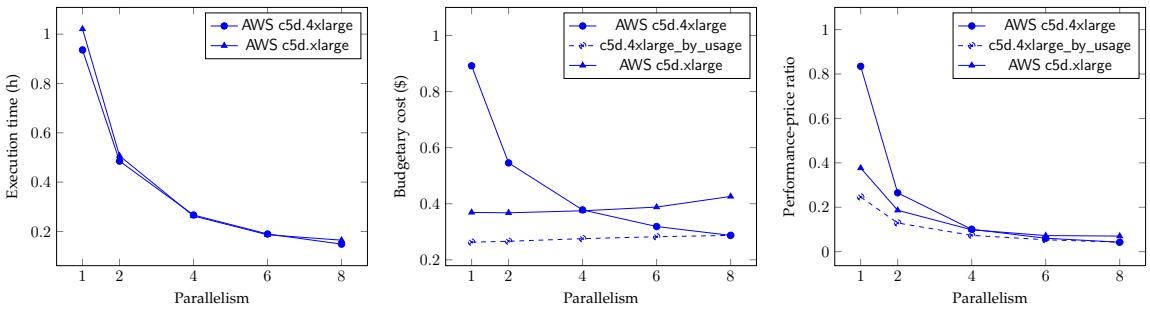
7.3.1 Scalability Evaluation for the Cloud Retrieval Application

The cloud retrieval *m4: scale-up* and *m5: scale-out* evaluations are shown in Fig. 8. As illustrated in Fig. 8a, the *m1: execution time* decreases when the number of executors increases in both AWS and Azure with similar trends. The *m2: budgetary cost* as shown in Fig. 8b, however, decreases in *m4: scale-up* and increases in *m5: scale-out* when the number of executors increase. The reason is that in cluster scale-up, the same resources were used while their execution time was decreasing; and in cluster scale-out, the costs saved by less execution time costs were less than the costs increased with additional resources. If only calculating the cost by usage for *m4: scale-up* case, its trends become similar to those of *m5: scale-out*. Combining cost and time, as illustrated in Fig. 8c, the *m3: PPR* in *m4: scale-up* and scale-up by usage decrease when the numbers of executors increase. However,



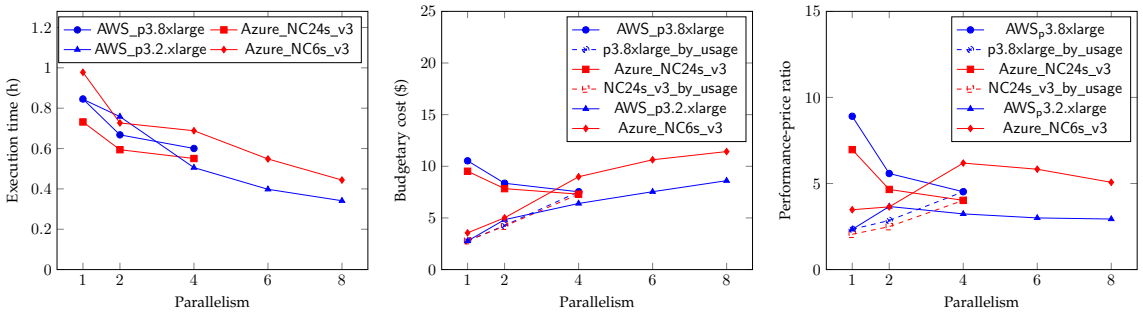
(a) Execution time. (b) Budgetary cost. (c) Performance-price ratio.

Fig. 8: Scalability evaluation of RPAC toolkit for the **cloud retrieval application**: **scale-up** (circle and square mark) and **scale-out** (triangle and diamond mark) for AWS and Azure. Dashed line: cost value calculated by its usage.



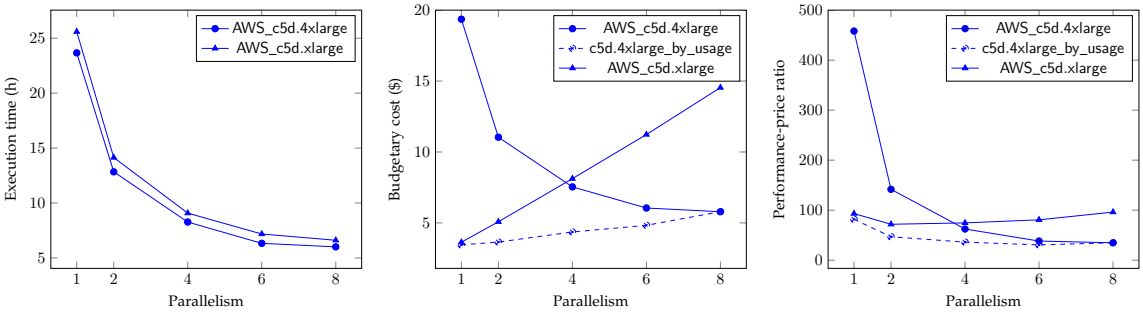
(a) Execution time. (b) Budgetary cost. (c) Performance-price ratio.

Fig. 9: Scalability evaluation of RPAC toolkit for the **causality discovery application**: **scale-up** (circle mark) and **scale-out** (triangle mark) for AWS. Dashed line: cost value calculated by its usage.



(a) Execution time. (b) Budgetary cost. (c) Performance-price ratio.

Fig. 10: Scalability evaluation of RPAC toolkit for the **domain adaptation application**: **scale-up** (circle and square mark) and **scale-out** (triangle and diamond mark) for AWS and Azure. Dashed line: cost value calculated by its usage.



(a) Execution time. (b) Budgetary cost. (c) Performance-price ratio.

Fig. 11: Scalability evaluation of RPAC toolkit for the **satellite collocation application**: **scale-up** (circle mark) and **scale-out** (triangle mark) for AWS. Dashed line: cost value calculated by its usage.

the *m3: PPR* first decreases but later increases a little bit in *m5: scale-out* cases. The figure also shows AWS achieves better *m3: PPR* than Azure, and *m4: scale-up* achieves better *m3: PPR* than *m5: scale-out*. So the best *m3: PPR* for the Dask-based big data application with virtual CPU nodes is achieved by *m4: scale-up* of application with more executors in AWS.

7.3.2 Scalability Evaluation for the Causality Discovery Application

Because Azure HD-Insight cluster does not support Docker-based Spark computation, we only focus on the evaluation of causality discovery for AWS, which is shown in Fig. 9. The trends for this application are very similar to those for the previous application since they both are CPU-based. As illustrated in Fig. 9a, the *m1: execution time* for both *m4: scale-up* and *m5: scale-out* decreases dramatically by at most 80% when the parallelism increases. This change of time appears more significant in causality discovery compared with what is in cloud retrieval. For the *m2: budgetary cost* in Fig. 9b, when the parallelism increases, the *m4: scale-up* decreases, while both *m5: scale-out* and scale-up by usage increase with similar trends. For all three metrics in Fig. 9c, The *m3: PPR* decreases when the numbers of executors increase. As a result, it is better to use a larger number of executors in the Spark-based big data analytics with virtual CPU nodes.

7.3.3 Scalability Evaluation for the Domain Adaptation Application

For domain adaptation, the evaluations are shown in Fig. 10. Because the maximal number of GPUs in one instance is 4 for Azure, we compare *m4: scale-up* only from 1 GPU to 4 GPUs. As illustrated in Fig. 10a, same with the findings from other data analytics, the *m1: execution time* decreases when the numbers of GPUs increase in both AWS and Azure. The *m2: budgetary cost* in Fig. 10b, also have the same regularity compared with CPU-based analytics. For *m3: PPR*, as illustrated in Fig. 10c, more GPUs lead to better ratios for *m4: scale-up* and worse ratios for scale-up by usage. For *m5: scale-out*, *m3: PPR* first gets worse and then improves a little bit. But still launching with only 1 instance can have the best *m3: PPR* for both AWS and Azure execution.

7.3.4 Scalability Evaluation for the Satellite Collocation Application

The above three applications already show the effectiveness of RPAC for parallel frameworks in different clouds. We further evaluate the satellite collection application with over 1 TB input data on AWS, and its longest total execution time is over 25 hours. As shown in Fig. 11, the *m1: execution time* in Fig. 11a of all *m4: scale-up* experiments decrease around 1 to 2 hours compared with all *m5: scale-out* experiments in the same parallelism setting. Thus, parallel execution in one VM with scale-up deployment is preferred, since *m5: scale-out* generates more communication overheads between different nodes. For the *m2: budgetary cost* as illustrated in Fig. 11b, when the number of executors increases, the *m4: scale-up* gets a more reasonable price while the *m5: scale-out* becomes more expensive. Different with previous findings, *m2: budgetary cost* of *m5: scale-out*, *m4: scale-up* and scale-up by usage change very dramatically by at most 75% when the parallelism changes. The reason is that the execution time of

the big data application is much longer than others. Combining cost and time, as illustrated in Fig. 11c, the *m3: PPR* of *m4: scale-up* is decrease when the numbers of executors increase. The *m3: PPR* of scale-up by usage and *m5: scale-out* are first decrease but later increase a little bit. As a result, the better parallelism strategy for the big data application is using more executors in *m4: scale-up* deployment.

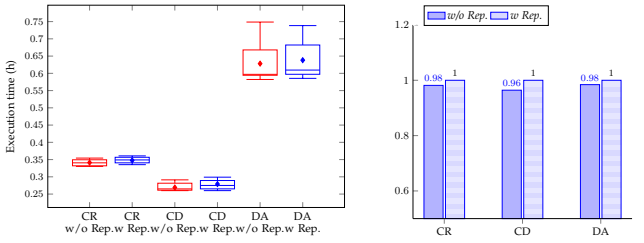
7.4 Benchmarking for Reproducibility Efficiency

In this section, we assess the efficiency of reproducibility for RPAC toolkit in the first three applications. We first evaluate the overhead caused by serverless-based reproducibility, then we compared the efficiency between serverless-based and SDK-based approaches.

7.4.1 Efficiency Comparison for Reproducibility Support

We first measure the *m6: reproducibility_overhead* of our applications with and without reproducibility support. For each application, we measure the AWS scale-up with 4 parallelisms, run each experiment 10 times and collect all results in a box-plot shown in Fig. 12. From the figure, we can see having reproducibility support did not cause much overhead, which is less than 0.01 hours, for all applications. The overhead percentage caused by reproducibility for cloud retrieval (CR), causality discovery (CD), and domain adaptation (DA), are 1.28%, 3.58%, and 2.17%, respectively. Besides, the time range of GPU-based analytics is larger than both CPU-based analytics, which means the execution time of GPU-based computation is more unstable than CPU-based one.

We utilize a statistical hypothesis test approach, called T-test [31], to determine whether the execution time with and without reproducibility support differ statistically. T-test determines a possible conclusion from two different hypotheses. By calculating the corresponding p-value [32], we can measure the probability that an observed difference has occurred just by random chance. Hypothesizing that the reproduce execution provides some overhead over the execution without reproducibility, we calculate the p-value for the two sample t-test with equal variance. The p-values of CR, CD and DA, turn out to be 0.4968, 0.3193 and 0.3634. Since these are not less than $p = 0.05$, we fail to reject the null hypothesis of the tests. As the result, we do not have sufficient evidence to say that the average execution time between the two species (with and without reproducibility support) is different for all three applications.



(a) The box-plots. (b) The relative difference.

Fig. 12: The box-plots and its relative difference for application execution time with and without reproducibility support.

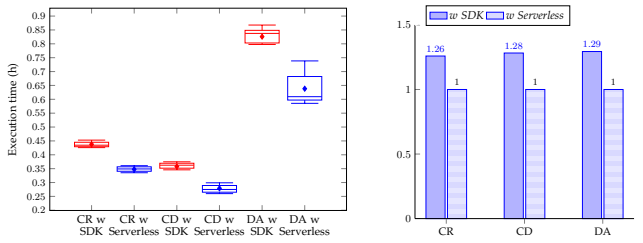
7.4.2 Comparison with SDK-based Reproduction

Besides the serverless-based approach, as shown in TABLE 1, we also implemented an SDK-based automatic execution mode which is achieved by periodical status pulling. In order to explore their difference, we evaluate the *m7: reproducibility_efficiency* of these two approaches with the same applications. Same with the previous measurement setting, we run each experiment 10 times and collect all results in a box-plot as illustrated in Fig. 13. For SDK-based approach, the time window for each status pulling is set to 10 second. The figure shows that serverless-based approach is more efficient than SDK-based approach, and the percentage of overhead reduction for CR, CD and DA, are 25.92%, 28.24% and 29.41%, respectively. The time range of serverless-based approach is larger than SDK-based one especially in GPU-based analytics. The reason is that, in SDK-based approach, the execution status monitoring could be delayed with periodical pulling. With the serverless function and event trigger, serverless-based approach enables big data analytics to be measured more efficiently and with less noise.

We also use T-test to determine whether the execution time using serverless approach and SDK-based approach differs statistically. Hypothesize that the serverless-based approach provides some efficient benefit over SDK-based approach. The p-values of CR, CD and DA, turn out to be $9.31\text{e}{-15}$, $6.00\text{e}{-14}$ and $2.19\text{e}{-06}$. Since these p-values are less than 0.05, we can reject the null hypothesis of the tests. The serverless-based approach is indeed providing statistically significant efficient benefit compared with SDK-based approach.

8 Related Work

There have been many studies on cloud-based reproducibility. Some of them [35, 40–43] only study its conceptual frameworks. In this section, we only discuss those having actual systems/toolkits. As shown in TABLE 4, we categorize related work into four groups based on their systems' capabilities. Besides, we also selected two most related works to compare in detail. The comparison is shown in TABLE 5 where the first one also leverages serverless computing and the second is one of the most recent work on cloud based reproducibility.



(a) The box-plots. (b) The relative difference.

Fig. 13: The box-plots and its relative difference for application execution time with serverless-based and SDK-based approach

TABLE 4: Comparison of related work for cloud-based reproducibility.

Approach	Scalable environment provision	Automated execution and reproducibility	History retrieval based reproducibility	Cross-cloud reproducibility
ReproZip [33], CARE [34], DevOps-based [35], Skyport [36], Hyperflow [37], TOSCA [38, 39]	◐	○	○	○
AMOS [40], WSSE [41], PDIFF [42], ReCAP [43], Tapis [16] OpenWhisk [14, 17], NeuroCAAS [44]	●	○	●	○
PRECIPE [45], Chef [46], Apt [47], Semantic Driven [48, 49]	●	○	○	●
RPAC (this work)	●	●	●	●

8.1 General Comparison with Related Work

Among the related studies in TABLE 4, nearly all related approaches achieve the software environment provision for reproducibility. However, approaches in group 1 mainly use the archived or containerized software environment, which limits the scope of applicability and lacks support for maintaining hardware configurations within cloud. Additionally, they monitor execution status by system commands or cloud APIs based periodical pulling which is less efficient than event-based execution in our work. For example, ReproZip [33] tracks system commands and zips collected information along with all the used system files together for reproducibility. CARE [34] reproduces a job execution by monitoring and archiving all the material required to re-execute operations. For related work in group 2, their proposed approaches encapsulate the code dependencies and software in virtual machine images or graphs, and enable history retrieval for reproduction. For instance, WSSE [41] proposes to generate digital data and source code snapshots to be reproduced and distributed within a cloud-computing provider. The Tapis [16] open-source API platform was proposed for accomplishing distributed computational experiments in a secure, scalable, and reproducible way. With the implemented pipeline with the Python API, the containerized applications can be submitted, scheduled and executed as tasks using a traditional HPC batch scheduler such as SLURM. AMOS [40] uses a VM containing a set of tools previously installed to implement a mechanism that initializes and configures VMs on demand. However, this reproduction is more like a history repetition, which is designed for verification and validation of history execution. They also provide configurable environment variables for automatic resource deployment in a single cloud, but do not support cross-cloud reproducibility. Instead, our proposed RPAC uses a data abstraction for information needed for reproducibility and transforms resource configurations used in one cloud into those in another cloud.

For related studies in TABLE 4, group 3's capabilities are closest to ours. These approaches rely on annotated information provided by a user to assign workflow, and software/hardware environment. For example, PRECIPE [45] provides APIs to access both AWS and private cloud. However, users need to call the functions in order and have to manually terminate resources after the experiment is done, so it does not support automated end-to-end execution and reproducibility. The whole execution has to wait at the client side before the next function can be called. On the contrary, RPAC serverless event triggering enables fewer communications between client and cloud, which improves the efficiency for cloud analytics. Chef [46] achieves virtual execution environment launching and termination via designed knife commands. Chef client is installed in virtual machines to run the pipeline within the virtual machines. So some internal steps of the application can be executed within the virtual machines via its pipeline. However, Chef does not support full automation since its user has to wait at client side to manually terminate resources after the experiment is done. Apt [47] uses user-provided profiles, which consists of a cluster with a control system to instantiate encapsulated experiment environments, for repeating historical research. From this information, they deduce the required execution resources in cloud and then re-provision or configure them through their own APIs. In comparison, we use a serverless-based pipeline and follow cloud function APIs provided by cloud providers so the execution/reproduction process can be managed by the cloud without communications with toolkit. Also, their fully created annotations, even in cross-cloud reproduction, rely heavily on the users instead of execution history. Our work abstracts information required by users from information in execution history, users only need to provide minimal information to reproduce. Our toolkit will transform user-provided information into executable pipeline. For automated execution and reproducibility, none of these approaches can achieve full automation including resources and software provisioning, analytics execution and termination. RPAC's event-based automation is the one-command execution that achieves a more efficient cloud computation and reproduction. For cross-cloud reproducibility, we further use the adapter pattern model to achieve the configuration mapping without taking all inputs from the user.

8.2 Detailed Comparison with Most Related Work

As shown in TABLE 5, Apache OpenWhisk [17][14] is an open-source, distributed Serverless cloud platform. In their

TABLE 5: Detailed comparison among serverless computing related work.

	Pipeline description	Automated deployment, execution and reproduction	Parallel execution on cloud	Capability of cross-cloud deployment/reproducibility
OpenWhisk [14, 17]	Consists of events, triggers and action functions, which are implemented by themselves.	User needs to initiate the cloud cluster, then provides the address of Kubernetes to Openwhisk. No resource termination.	No explicit parallel framework support. Users have to provide environment and implement the pipeline by themselves.	Not supported directly. Users have to rewrite the pipeline by themselves.
NeuroCAAS [44]	Consists of the specification of analysis and infrastructure stack. Toolkit provides formatted pipelines in a public repository.	Pull pipeline from public repository and select listed configurations. The options are limited. No resource termination.	No explicit parallel framework support. Could be scripted and implemented by users in the pipeline.	Not supported directly. Users have to rewrite the pipeline by themselves.
RPAC	Consists three aspects of abstraction and the serverless functions. Users can update configurations from templates.	Full automated Cloud SDK mode and serverless mode. Include hardware provisioning and resource termination.	Provide three parallel frameworks for Spark -based and Dask-based analytics on CPUs, and Horovod-based analytics on GPUs.	By modifying three aspects of abstraction, RPAC enables different levels of reproduction.

serverless design, functions are explicitly defined in terms of the event, trigger, and action, which are implemented by users. Events are generated from event sources, which often indicate changes in data or carry data themselves. The trigger is defined by specifying its name and parameters (key-value pairs). It is associated with an action. The action is defined as functions (code snippets), which encapsulate application logic to be executed in response to events. Before deployment, users need to initiate the cluster in the cloud and provide the hostname and port of the Kubernetes cluster to the toolkit. No resource termination option when deployment finishes. Openwhisk has three deployment options. 1) OpenWhisk can be deployed using Helm charts on any Kubernetes provisioned from a public cloud provider. 2) The deployment can be achieved by OpenWhisk REST API or OpenWhisk CLI. 3) Use the cloud-defined CLI on a cloud provider that already provisions Apache OpenWhisk as a service, which is only supported by IBM cloud as of now. For parallel execution, Openwhisk does not provide direct parallel framework support. To enable scalable execution, users need to initiate a cloud cluster with the parallel software environment and prepare docker images with a parallel framework. Then users need to implement the parallel logic in the pipeline's action. OpenWhisk enables the deployment on different clouds with Helm charts on any Kubernetes. However, an application for one cloud cannot be redeployed in another cloud, unless users 1) initiate instances on another cloud with the Kubernetes cluster, 2) rewrite the events and triggers in the pipeline, and 3) provide the new hostname and port of Kubernetes cluster to the toolkit and redeploy the pipeline. Another example is NeuroCAAS [44]. NeuroCAAS provides formatted pipelines, called blueprints, in a public code repository and defines a resource bank that can make hardware available through pre-specified instances in one specific cloud. The users are able to update the blueprint with new configurations and upload its new version to the public repository for deployment and reproduction. The users need to provide the blueprint's repository address for automated deployment, execution and reproduction. By default, NeuroCAAS fixes a single instance type per analysis in order to facilitate reproducibility. With the blueprint, datasets, and configuration files for one analysis, NeuroCAAS achieves reproducibility for corresponding analyses with the same environment and configuration. For parallel execution, NeuroCAAS does not provide direct parallel framework support. The logic of parallel processing must be explicitly scripted and implemented in the blueprint by users.

9 Conclusions

Reproducibility is an important way to gain the confidence of new research contributions. In this paper, we study how to achieve cloud-based reproducibility for big data analytics. By leveraging serverless, containerization and adapter design pattern techniques, our proposed approach and RPAC toolkit can achieve reproducibility, portability and scalability for big data analytics. Our experiments show our toolkit can achieve good scalability and low overhead for reproducibility support for both AWS and Azure.

For future work, we will mainly focus on the following three aspects. First, we will optimize the executions in terms of time, cost or ratio by mining execution history, and further optimize the overhead of reproducibility via better data abstraction, modeling and storage. Second, we will extend our work to easily publish data analytics as public records following the Research Object framework [19] so they can be referred via DOI identifiers later. Third, we will study how to utilize execution history data to achieve automated execution optimization based on users' objectives (time, cost or ratio) and datasets.

Acknowledgment

This work is supported by the National Science Foundation (NSF) Grant No. OAC–1942714, National Aeronautics and Space Administration (NASA) grant No. 80NSSC21M0027 and U.S. Army Grant No. W911NF2120076.

References

[1] National Academies of Sciences, Engineering, and Medicine and others, *Reproducibility and replicability in science*. National Academies Press, 2019.

[2] R. Qasha, J. Cała, and P. Watson, "A framework for scientific workflow reproducibility in the cloud," in *2016 ieee 12th international conference on e-science (e-science)*. IEEE, 2016, pp. 81–90.

[3] B. Howe, "Virtual appliances, cloud computing, and reproducible research," *Computing in Science & Engineering*, vol. 14, no. 04, pp. 36–41, 2012.

[4] "Reproducible and portable big data analytics in cloud," https://github.com/big-data-lab-umbc/Reproducible_and_portable_app_in_cloud, 2021, doi: 10.5281/zenodo.7548181, accessed: 2021-10-01.

[5] "Apache Spark Project," http://spark.apache.org, 2021, accessed: 2021-5-28.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] "Dask: Scalable analytics in Python," https://dask.org/, 2021, accessed: 2021-5-28.

[8] "Horovod: Distributed training framework for TensorFlow, Keras, PyTorch, and Apache MXNet." https://horovod.readthedocs.io/, 2021, accessed: 2021-5-28.

[9] J. Kraus, "An introduction to cuda-aware mpi," *NVIDIA Developer Blog*, 2013.

[10] L. A. Barba, "Terminologies for reproducible research," *arXiv preprint arXiv:1802.03311*, 2018.

[11] Association for Computing Machinery, "Artifact Review and Badging," https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2021, accessed: 2021-4-13.

[12] F. Bartusch, M. Hanussek, J. Krüger, and O. Kohlbacher, "Reproducible scientific workflows for high performance and cloud computing," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 161–164.

[13] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017*

IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017, pp. 405–410.

[14] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 287–288.

[15] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster *et al.*, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.

[16] J. Stubbs, R. Cardone, M. Packard, A. Jamthe, S. Padhy, S. Terry, J. Looney, J. Meiring, S. Black, M. Dahan *et al.*, "Tapis: An api platform for reproducible, distributed computational research," in *Future of Information and Communication Conference*. Springer, 2021, pp. 878–900.

[17] The Apache Software Foundation, "Apache OpenWhisk." http://openwhisk.org/, 2016.

[18] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, 2003, pp. 129–131.

[19] S. Bechhofer, I. Buchan, D. De Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfield, I. Dunlop *et al.*, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 599–611, 2013.

[20] S. Perera, A. Perera, and K. Hakimzadeh, "Reproducible experiments for comparing apache flink and apache spark on public clouds," *arXiv preprint arXiv:1610.04493*, 2016.

[21] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, J. Vlissides *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[22] C. Wang, S. Platnick, K. Meyer, Z. Zhang, and Y. Zhou, "A machine-learning-based cloud detection and thermodynamic-phase classification algorithm using passive spectral observations," *Atmospheric Measurement Techniques*, vol. 13, no. 5, pp. 2257–2277, 2020.

[23] P. Guo, A. Ofonedu, and J. Wang, "Scalable and hybrid ensemble-based causality discovery," in *2020 IEEE International Conference on Smart Data Services (SMDS)*, 2020, pp. 72–80.

[24] B. Sun and K. Saenko, "Deep coral: Correlation alignment for deep domain adaptation," in *European conference on computer vision*. Springer, 2016, pp. 443–450.

[25] R. Holz, S. Ackerman, F. Nagle, R. Frey, S. Dutcher, R. Kuehn, M. Vaughan, and B. Baum, "Global moderate resolution imaging spectroradiometer (modis) cloud detection and height evaluation using caliop," *Journal of Geophysical Research: Atmospheres*, vol. 113, no. D8, 2008.

[26] NOAA GOES Satellite Data, "NOAA Geostationary Operational Environmental Satellites (GOES) 16, 17 & 18," https://registry.opendata.aws/noaa-goes/, 2021, accessed: 2022-10-06.

[27] D. M. Winker, M. A. Vaughan, A. Omar, Y. Hu, K. A. Powell, Z. Liu, W. H. Hunt, and S. A. Young, "Overview of the calipso mission and caliop data processing algorithms," *Journal of Atmospheric and Oceanic Technology*, vol. 26, no. 11, pp. 2310–2323, 2009.

[28] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–36, 2018.

[29] K. Li, "Quantitative modeling and analytical calculation of elasticity in cloud computing," *IEEE Transactions on Cloud Computing*, vol. 8, no. 4, pp. 1135–1148, 2017.

[30] J. Wang, P. Korambath, I. Altintas, J. Davis, and D. Crawl, "Workflow as a service in the cloud: architecture and scheduling algorithms," *Procedia computer science*, vol. 29, pp. 546–556, 2014.

[31] T. K. Kim, "T test as a parametric statistic," *Korean journal of anesthesiology*, vol. 68, no. 6, p. 540, 2015.

[32] T. Dahiru, "P-value, a true test of statistical significance? a cautionary note," *Annals of Ibadan postgraduate medicine*, vol. 6, no. 1, pp. 21–26, 2008.

[33] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 2085–2088.

[34] Y. Janin, C. Vincent, and R. Duraffort, "Care, the comprehensive archiver for reproducible execution," in *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, 2014, pp. 1–7.

[35] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[36] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai *et al.*, "Skyport-container-based execution environment management for multi-cloud scientific workflows," in *2014 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE, 2014, pp. 25–32.

[37] M. Orzechowski, B. Baliś, R. G. Słota, and J. Kitowski, "Reproducibility of computational experiments on kubernetes-managed container clouds with hyperflow," in *International Conference on Computational Science*. Springer, 2020, pp. 220–233.

[38] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Tosca: portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[39] R. P. P. Qasha, "Automatic deployment and reproducibility of workflow on the cloud using container virtualization," Ph.D. dissertation, Newcastle University, 2017.

[40] R. J. Strijkers, R. Cushing, D. Vasyunin, C. de Laat, A. Belloum, R. J. Meijer *et al.*, "Toward executable scientific publications." in *ICCS*, 2011, pp. 707–715.

[41] J. T. Dudley and A. J. Butte, "In silico research in the era of cloud computing," *Nature biotechnology*, vol. 28, no. 11, pp. 1181–1185, 2010.

[42] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workflow reproducibility analysis," *Concurrency and Computation: Practice and Experience*, vol. 28, pp. 995–1015, 2016.

[43] M. K. H. Ahmad, "Scientific workflow execution re-

producibility using cloud-aware provenance," Ph.D. dissertation, University of the West of England, 2016.
[44] T. Abe, I. Kinsella, S. Saxena, E. K. Buchanan, J. Couto, J. Briggs, S. L. Kitt, R. Glassman, J. Zhou, L. Paninski *et al.*, "Neuroscience cloud analysis as a service: An open-source platform for scalable, reproducible data analysis," *Neuron*, vol. 110, no. 17, pp. 2771–2789, 2022.
[45] S. Azarnoosh, M. Rynge, G. Juve, E. Deelman, M. Niec, M. Malawski, and R. F. Da Silva, "Introducing precip: an api for managing repeatable experiments in the cloud," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2. IEEE, 2013, pp. 19–26.
[46] J. Klinginsmith, M. Mahoui, and Y. M. Wu, "Towards reproducible escience in the cloud," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 582–586.
[47] R. Ricci, G. Wong, L. Stoller, K. Webb, J. Duerig, K. Downie, and M. Hibler, "Apt: A platform for repeatable research in computer science," *ACM SIGOPS Operating Systems Review*, vol. 49, pp. 100–107, 2015.
[48] I. Santana-Perez, R. F. Da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho, "A semantic-based approach to attain reproducibility of computational environments in scientific workflows: A case study," in *European Conference on Parallel Processing*. Springer, 2014, pp. 452–463.
[49] I. Santana-Perez, R. F. Da Silva, M. Rynge, E. Deelman, M. S. Pérez-Henández, and O. Corcho, "Leveraging semantics to improve reproducibility in scientific workflows," in *The reproducibility at XSEDE workshop*, 2014.

Xin Wang received her PhD in Information Systems from the University of Maryland, Baltimore County in 2022. Her PhD research interests include distributed computing (systems), blockchains, big data analytics, federated learning, cloud computing and reproducibility.

Pei Guo is working as a Data Scientist at Wyze Labs. She received her PhD in Information Systems from the University of Maryland, Baltimore County in 2021. Her PhD researches focused on spatiotemporal causal modeling on large-scale data, big data application parallelizing and cloud computing. Her contribution to the paper was done during her PhD study at UMBC.

Xingyan Li is a PhD student working in the Big Data Analytics Lab at the Department of Information Systems, University of Maryland, Baltimore County. Her research interests include data science, machine learning and deep learning.

Aryya Gangopadhyay is a Professor in the Department of Information Systems and the Director of the Center for Real-time Distributed Sensing and Autonomy (cards.umbc.edu) at University of Maryland, Baltimore County. His research interests include Machine Learning and cybersecurity. His research has been funded by NSF, ARL, IBM, and the US Department of Education.

Carl Busart received the B.S. and M.S. degrees from Johns Hopkins University, an MBA from the University of Maryland, College Park, and the D.Eng. degree from George Washington University. He is a branch chief at the U.S. Army Research Laboratory and a member of the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). His research interests include artificial intelligence / machine learning (AI/ML) and secure design.

Jade Freeman is the Chief of Battlefield Information Systems Branch at DEVCOM Army Research Laboratory. Her research interest includes information systems for decision support, human-information interactions, and information theory.

Jianwu Wang is an Associate Professor of Data Science and the Director of the Big Data Analytics Lab at the Department of Information Systems, University of Maryland, Baltimore County. His research interests include Big Data Analytics, Scientific Workflow, Distributed Computing, Service Oriented Computing. He has published 120+ papers with more than 2700 citations (h-index: 25).