A Microservice-Based Building Block Approach for Scientific Workflow Engines: Processing Large Data Volumes with DagOnStar

Dante D. Sánchez-Gallegos
Unidad Tamaulipas, Cinvestav
Victoria, Mexico
dsanchez@tamps.cinvestav.mx

Diana Di Luccio
Dpt. of Science and Technologies
University of Naples "Parthenope"
Naples, Italy
diana.diluccio@uniparthenope.it

J. L. Gonzalez-Compean
Unidad Tamaulipas, Cinvestav
Victoria, Mexico
jgonzalez@tamps.cinvestav.mx

Raffaele Montella
Dpt. of Science and Technologies
University of Naples "Parthenope"
Naples, Italy
raffele.montella@uniparthenope.it

***Abstract*—The impact of machine learning algorithms on everyday life is overwhelming until the novel concept of datacracy as a new social paradigm. In the field of computational environmental science and, in particular, of applications of large data science proof of concept on the natural resources management this kind of approaches could make the difference between species surviving to potential extinction and compromised ecological niches. In this scenario, the use of high throughput workflow engines, enabling the management of complex data flows in production is rock solid, as demonstrated by the rise of recent tools as Parsl and DagOnStar. Nevertheless, the availability of dedicated computational resources, although mitigated by the use of cloud computing technologies, could be a remarkable limitation. In this paper, we present a novel and improved version of DagOnStar, enabling the execution of lightweight but recurring computational tasks on the microservice architecture. We present our preliminary results motivating our choices supported by some evaluations and a real-world use case.**

***Index Terms*—Microservices, Workflows, Virtual Containers, Parallel Processing, Cloud Computing**

I. Introduction

The acquisition, processing, management, and discovery of data are key tasks when observing the earth and its natural phenomena. These tasks also produce information for organizations to support the decision-making process and to conduct scientific studies. The applications that perform these tasks are commonly modeled as stages, which are chained to create processing structures called workflows [1], [2]. These structures are created by using directed acyclic graphs (DAGs), where nodes represent processing stages and0 the edges represent the I/O interfaces used to chain the stages considered in a workflow.

Multiple tools and frameworks are currently available for scientists to design workflows [1]–[3]. These tools are in charge of executing and deploying each stage of a workflow, in an automatic manner, on a high-performance IT infrastructure (e.g., clusters of servers, the cloud or HPC [3], [4]). These tools are also in charge of extracting data required by the stages to transform data into information as well as delivering that information either to another stage according to a well-defined sequence (DAG) in a synchronized manner.

The traditional scientific workflows include multiple heterogeneous tasks created by using diverse types of programming models, which even could be executed in several platforms and/or infrastructures. Moreover, the tasks may be deployed on either distributed or centralized IT infrastructures such as high-performance clusters (HPC), the cloud and single servers [2], [5]. Nevertheless, traditional workflows require a homogeneous environment to ensure a given functionality. For instance, all the applications should be written in a given programming language (e.g., Python [1], [2] and Java [6]), and for a given platform (e.g., a given operating system). These restrictions are commonly imposed to enforce the correct functionality of the parallel model (commonly based on multi-threads at stage level) added to the workflow frameworks for improving the performance of workflow solutions created by scientists. This heterogeneity also produces performance issues affecting the efficiency of the workflow solutions.

In this context, there is a need for solutions that not only enable scientists to organize their applications in the form of workflows and execute them in an automatic manner, but also for solutions that can deal with the heterogeneity of the development and performance of applications included in workflows that arises in a real-world scenario.

In this paper, we present the design, development, and evaluation of a building block approach for workflow engines by using virtual containers (VCs) and microservices. In this approach, the building blocks encapsulate the stages of the workflows into VCs to provide the stages with portability. These building blocks are interoperable software pieces, which

978-1-7281-5686-6/19/$31.00 ©2019 IEEE
DOI 10.1109/SITIS.2019.00066

can be organized in the form of parallel patterns to improve the performance of the stages encapsulated into the building blocks and to solve the performance heterogeneity of the stages of a workflow. These building blocks are managed in the form of microservices to ensure the correct functionality of the parallel model as well as to solve the development heterogeneity issues that arise when managing stages developed by using different programming languages.

To show the feasibility of this approach, we added the building block management in a microservice architecture to a workflow engine called DagOnStar [5]. To show the efficiency of this approach, we developed a prototype with the new version of DagOnStar, which was evaluated in two case studies. The first case is based on workflows that extract real repositories of environmental data collected from IoT Sensors. These data are transformed through stages such as data pre-processing to detect outliers, data processing to find out hidden patterns in the data, and data storing for preserving the processed data.

The second case was conducted by pre-processing and processing satellite imagery captured by a sensor called LandSat8 to correct radiometric and atmospheric issues in the satellite images of this repository.

In these case studies, the workflows were created by using the building blocks of the DagOnStar prototype. The costs of the workflow deployment and the performance of the resultant workflows were analyzed. Moreover, a comparison of this prototype with engines available in state of the art was also conducted.

The evaluation revealed the feasibility of using a building block approach to deploy workflows on a microservice architecture in an automatic manner, which solves the development of heterogeneity issues. It also revealed the flexibility of this approach to improve the performance of the workflows created with this approach by using master/slave patterns based on building blocks, which solving the performance heterogeneity issues.

The outline of this paper is as follows. In Section II related work to our solution is discussed. Section III is about design strategies of the building blocks. Section IV describes the implementation of a prototype based on the proposed approach. Section V the evaluation run to test DagOnStar. Section VI presents the first case study focused on the processing of IoT sensors. Section VII presents the second case study focused on the processing of satellite imagery. Finally, Section VIII gives conclusion remarks and draws the path for future research development.

II. Related work

The workflow systems enable organizations to deploy pipelines of applications on different types of IT infrastructures. Galaxy [1], Parsl [2], Pegasus [7], Makeflow [8] and Swift [9] are only some examples of this type of system.

In practice, it is common that IT staff performs troubleshooting procedures when the organizations deploying workflows on a given infrastructure (e.g., clusters of computers, virtual machines in the cloud, and the grid). This type of procedure includes tasks such as installing applications and dependencies in virtual/physical machines, setting the environment of each application and arrangements in the configurations of each infrastructure to deploy virtual machines. Troubleshooting results in downtime that could disturb the continuity of studies and the usage of virtual machines could increase the time spent in deploying workflows.

The Virtual Containers (VCs) [10] have become an alternative to virtual machines as this virtualization technique reduces the need for troubleshooting procedures and the deployment time. Therefore, workflows engines, such as Skyport [11], Taverna [12], and Makeflow [8], are currently using virtual containers for end-users to deploy workflow solutions on different types of platforms. Scripting languages (e.g., YAML [13] or Swift [9]) have also been proposed to create workflows by using codification.

Nevertheless, the heterogeneity of the development and performance of the applications is an open issue for traditional workflow engines. Moreover, the reutilization of smaller parts of a workflow (a task or a subset of tasks) is not trivial because of the dependency between tasks.

The microservice architectures [14] have become a popular technology that allows developers to decouple modules from a large service to create tiny independent and isolated services [15]. The microservices already allow users to create different flows by re-using microservices previously created [15] and reduce the effects of the deployment application issues.

Instead of using multi-thread parallelism as used in traditional workflow engines [5], [16], the approach proposed in this paper is based on parallel patterns created by using containerized building blocks, which are focused on addressing the performance heterogeneity issue. In addition, the incorporation of microservices has been proposed for facing up heterogeneity application deployment and enabling the reuse of components.

III. Design principles of the Building block-based approach

In this section, we present the design principles of a Microservice-based building block approach for scientific workflow engines.

A. DagOnStar Overview

We designed DagOnStar [5] to reduce the runtime footprint within the actual application. DagOnStar leverages the application life-cycle, supported by a Python library providing the main system components, while a not mandatory service component is used for workflow monitoring and management.

Figure 1 shows a conceptual representation of DagOnStar architecture. As it can be seen, this architecture considers four main components: i) The *Python library ecosystem* that supports the workflow building. ii) The *workflow engine* maps applications with tasks, creates stages, assigns tasks to stages and builds workflows, including the created stages by using a directed acyclic graph (DAG). Service such as DagOnStar

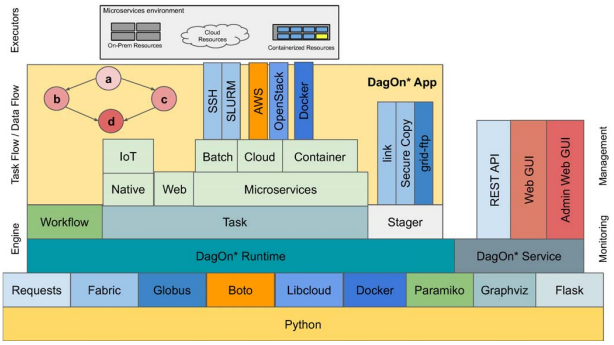

Fig. 1. The DagOnStar architectural schema. Tasks represented as container scripts and executed on a *containerized* infrastructure.

Runtime [17] and DagOnStar Service are in charge of the management and monitoring of building procedures. iii) *Task/data flow management* considers a suite of APIs to prepare the environment (basically defining configurations) according to the DAG of a workflow. iv) The *Executors* are in charge of the workflow deployment on a given infrastructure (e.g., Cloud services, container-based cloud or On-Premise Resources). Executors also establish controls and coordination over the execution of the tasks in the infrastructure (see Figure 1). A DagOnStar application is developed as a Python script where parallel tasks are defined by using the **Task** class. The DagOnStar Runtime performs the interaction with the executors both on local or remote resources instanced by either virtual machines or VCs deployed on public/private/hybrid clouds.

The DagOnStar design considers the `workflow://` schema as the root of the current workflow virtual file system. Under these conditions, `workflow:///workflow_unique_name//task_unique_name/` is the root of the scratch directory created by the DagOnStar Runtime. DagOnStar uses this notation to evaluate the task dependencies using a back-reference approach.

An important issue in this type of solution is data management. In DagOnStar, dataflows are constructed by using the `workflow://` schema, which indicated the data dependencies between tasks.

If two or more workflows interact with each other (regardless if they belong to the same application or different one), then the `workflow://` schema remains consistent for identifying any resource as `workflow://workflow_uuid/task_unique_name/`.

We extended the DagOnStar architecture by including microservices and VCs as building blocks.

B. Microservices as building blocks

In our approach, we encapsulated the applications considered in a workflow into VCs. We added I/O interfaces and control structures to the VCs where the applications have been encapsulated into. This VC represents the first Building Block considered in this approach, which we called *BB-VC*.

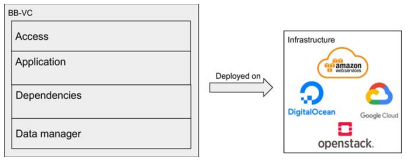

Fig. 2. Conceptual design of a microservice building block in DagOnStar.

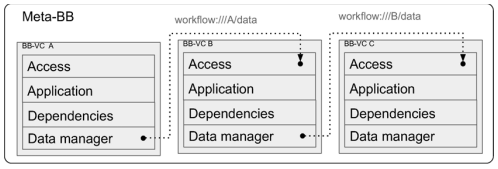

Fig. 3. A microservice (Meta-BB), chaining BB-VC to build a pipeline pattern.

In DagOnStar, the applications are encapsulated into VCs providing the workflow applications with portability property and allowing the deployment of the tasks on multiple platforms. A VC encapsulates the source code or binary of the application, as well as its dependencies libraries, packages, OS, and environment variables and any ancillary component needed to produce the task results. These VCs are constructed in the form of Building Blocks, which we called BB-VC.

Figure 2 depicts the conceptual design of a *BB-VC* in the form of a stack. At the top of the stack, there is a BB access control for receiving requests from DagOnStar. It also includes an application layer where a workflow task is placed in the form of source code or a binary. In the next level, are found the dependencies required by an application to execute a task. Finally, on the lower level, a data management component manages the I/O data of the BB-VC.

The *BB-VCs* are grouped in the form of microservices to create a meta Building Block (*Meta-BB*), which also includes I/O interfaces and control structures. The Meta-BBs allow to workflows designers to re-use previous tasks created in by the workflow engine [15], [18].

A Meta-BB represents a stage of a workflow, whereas the interconnection of multiple Meta-BBs produces a workflow, which can be allocated in different computational resources in a distributed manner.

A Meta-BB represents a front-end for a stage in a workflow. This means this structure is masking the management of BB-VCs deployed on an infrastructure. In order to do this masking procedure, a Meta-BB includes a proxy to distribute a load of tasks to the BB-VCs and a load balancing mechanism to keep a fair workload distribution. A Meta-BB also includes a DagOnStar *executor* to receive tasks assignments and to deliver a notification to the engine. This means that the engine is delegating the control of tasks distribution to the Meta-BB, which executes the tasks as a black box; as a result, a Meta-BB is an independent piece of software that can be deployed on different types of infrastructure, which improves the portability of the workflows created by DagOnStar.

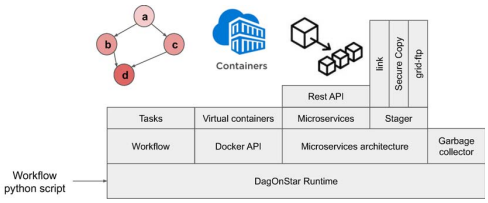

Fig. 4. Deployment of workflows.

The `workflow://` schema previously described is used to chain *Meta-BBs* to build dataflows, which are executed over a microservice ecosystem.

In DagOnStar, a *Meta-BB* has a scratch directory to write their outputs in the form of files (or directories depending on the application outputs) and considering the scratch directory as the root of the local storage supporting the I/O during the task execution. Other *Meta-BB* and tasks, can receive these files as input, which as to be staged in order to be processed. Files are transported between *Meta-BB* by using link/copy where they are in the same machine, or by using a data transference application/protocol (secure copy, grid-FTP [19], or a content delivery network such as SkyCDS [20]) when they are distributed through different computers.

C. Building parallel Patterns in Meta-BBs

In order to improve the efficiency of the workflows, developers can create patterns within a Meta-BB for facing up the performance heterogeneity. For instance, Figure 3 shows a conceptual design of the chaining of *BB-VCs* by using `workflow://` scheme to create a pipeline of BB-VCs. The microservice is in charge of the management of data. Notice that DagOnStar enables developers to create different types of patterns.

The implementation of a Meta-BB exposes an HTTP Rest API, which contains all the functions that allow microservices to communicate with each other. It also includes functions to exchange data and to create dataflows. This means Meta-BBs can be chained to produce a workflow modeled as a directed acyclic graph [5] in the very fashion performed in the previous version.

Thanks to the improvements described in this work about the management of BB-VC and Meta-BBs, the DagOnStar tasks can be deployed in a microservice ecosystem, where these tasks are managed and monitored by the core application life support (DagOnStar Runtime, see Figure 4). Deploying tasks as microservices enables the workflow engine to gain executor independence, efficiency and liability taking advantage of the benefits of this technology, such as, but not limited to, fault-tolerant design, decentralized data management, loosely coupling, indecently deploying, self-contained, and limited scope.

D. Life cycle of microservices building blocks

Figure 5 depicts the life cycle of a microservice building block in DagOnStar, which is integrated by seven stages,

Fig. 5. Life cycle of a microservice task.

which are:

1) In the first phase, for each *Meta-BB*, is created a scratch directory in the file system of the machine where it is going to be deployed on.
2) In the second phase, a VC image is created per each BB-VC considered in a Meta-BB. Each image includes the source code of the task and its dependencies on its corresponding BB-VC. In the case of the base image does not exist; it is downloaded from Docker Hub[1] or any other Docker registry configured by the user[2].
3) In this phase, each BB-VC image constructed in the previous phase is linked to the scratch directory with a volume inside the container of this BB-VC. The ports of all BB-VCs are published to be reachable by other microservices and applications.
4) In this phase, the Meta-BB is published in the DagOnStar service by using a unique token, which is used as an ID in the controlling of accesses to the data managed by a microservice. This allows a *Meta-BB* to receive petitions from other *Meta-BB* in the ecosystem.
5) In operation time, the *BB-VCs* included in a microservice are executed as a DagOnStar task. This means it receives input data, processes these data, and writes results as a file through the output interface.
6) The scratch directory is removed when the execution of the tasks has been completed.
7) The *Meta-BB* publication is removed from the ecosystem, removing the BB-VCs associated with that *Meta-BB*.

As can be seen, DagOnStar Stager manages the data between workflows and tasks, which reduces the copying of data produced by one task and used by another.

IV. Prototype Implementation Details

The prototype based on the microservice-based building block approach presented in this paper was implemented mainly in Python programming language using libraries for the management of cloud and Docker tasks. Microservices API was implemented using a Python library called Flask, and are deployed in production by using uWSGI and Nginx as the webserver. Microservices were encapsulated into Docker VCs and deployed on the Docker swarm platform.

V. Experimental Evaluation Methodology

An experimental methodology based on two case studies was conducted to evaluate the performance of the prototype described in the previous section.

[1]https://hub.docker.com/
[2]https://docs.docker.com/registry/

TABLE I
CHARACTERISTICS OF THE MACHINES USED TO CONDUCT THE EXPERIMENTAL EVALUATION.

Host	Sockets	Cores	Threads	RAM	Storage
compute1	2	6	2	64	3.2T
compute2	1	6	1	24	465.8G, 931.5G, 931.5G
compute3	2	8	1	64	931.5G, 931.5G, 931.5G, 931.5G

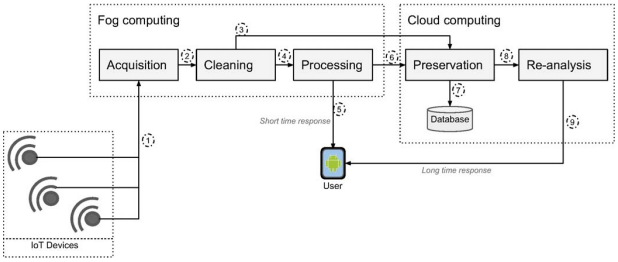

Fig. 6. Workflow for the processing of IoT data.

Two real-world workflow solutions were implemented by using this prototype to conduct two case studies. The first study is based on information for the preparation of IoT data [21]. The second study was based on the processing of satellite imagery captured by the satellite Landsat 8. We implemented a workflow composed of two main microservices: the pre-processing stage and the processing stage.

The prototype was evaluated in the experimental evaluation conducted by using a test environment, including three physical machines. Table I shows the main characteristics of the machines used in the experimental evaluation.

VI. CASE STUDY I: IOT DATA MANAGEMENT

We developed a workflow for collecting and processing environmental data acquired from IoT devices. We deployed a set of *Meta-BBs* to attend IoT devices in the fog computing, whereas another set was deployed on the cloud to process and manage the acquired data.

The data collected from sensors include device temperature, ambient temperature, humidity, motion, light, and Received Signal Strength Indicator.

The dataset was processed through the workflow depicted in Figure 6 which includes the next tasks deployed as *Meta-BB*:

- *Acquisition*: This microservice is in charge of receive the data captured by IoT devices [22], such as instruments/sensors [23] and smart devices. Each device is registered in this microservice, which enables it to send data to the workflow.
- *Cleaning*: This microservice includes the pre-processing of the data collected, removing outliers, and filling missing values by using Z-score technique [24].
- *Processing*: This microservice process the data to identify unusual behavior with these data (i.e., a fire where the sensor has been allocated in, or the dramatic ascend/descend in temperature caused by anomalies in the environment monitored by the sensor. This microservice produces short time alerts with the most recent data collected by the sensor. These alerts are sent to the application/user indicated in the initial configuration.
- *Preservation*: The microservices were deployed on the cloud to perform the management and preservation of the data collected by the sensors, the cleaned of data and the results of the data processing as well as the logs of the alerts.
- *Re-analysis*: This microservice performs a re-analysis of the data to get an overall summary of the data, performing tasks such as data regression, correlations, and data clustering.

The first three microservices (acquisition, cleaning, and processing) are deployed in fog computing, whereas the preservation and re-analysis are performed in the cloud.

A. Experiments

The workflow constructed in this case study was evaluated in three phases. In the first one, we evaluated the costs of the deployment of *Meta-BBs* in a workflow. The experiments of this phase were performed by measuring the average response time for the construction and deployment of *Meta-BB* images. In the second phase, the deployment of the workflow constructed with *Meta-BBs* was evaluated. The experiments of this phase include the deploying of the studied workflow on three different machines (see Table I), and varying the number of *Meta-BBs* running in parallel for each task in the workflow (see Figure 6). Finally, the performance of the workflow depicted in Figure 6 was evaluated by deploying the workflow with Batch tasks and *Meta-BB* on the infrastructure. It was evaluated the service time spent by this workflow to attend different numbers of concurrent requests sent to the workflow.

B. Results

In this section, are presented and analyzed the results of the performed experiments in the three phases previously described.

1) Analyzing the costs of construction and deployment of building blocks: Figure 7 shows, in the vertical axis, the response time for the deployment of the VC images (BB-VCs) used to execute each task of the workflow (horizontal axis) showed in Figure 6. For each stage of this workflow, Figure 7 shows the response time in two columns: The first one representing the response time when the BB-VCs are created and the second one when the BB-VCs are reused from images previously created. Each column includes three service times: the first one is the building of the basis image (includes an operating system, programming language compiler, or any other tool necessary to run the program). The second one represents the time required by DagOnStar to prepare the image by adding control structures and I/O libraries to the BB-VC. The last one represents the time to encapsulate an application into a BB-VC by adding the application (either binary or source code) and dependencies on the container. The sum of these times represents the costs of building a BB-VC. This means the first column represents the costs of

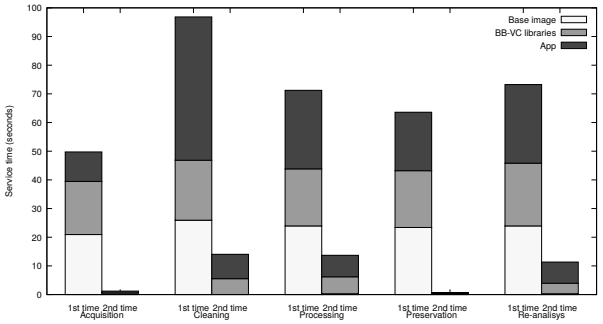

Fig. 7. Average response time for the deployment of tasks in the workflow.

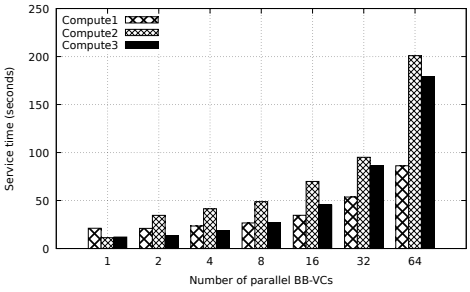

Fig. 8. Average Response time for the deployment of the workflow with different number of parallel *BB-VCs* in different machines.

building a microservice from scratch, whereas the second one represents the re-utilization of a microservice. As expected, the initiation of a microservice is more expensive than re-using an already configured one (the costs depend on the volume of the applications in both cases).

2) Analyzing the costs of deployment of workflows using Meta-BBs*:* The results of the second evaluation phase are described in this section. In this phase, we evaluated the costs of deploying a workflow on three different machines (see Table I). This workflow was configured to launch different numbers of *BB-VCs* in parallel for each Meta-BB (stage) in the workflow. The impact of the number of *BB-VCs* on the deployment of the workflow was assessed in these experiments.

Figure 8 shows, in the vertical axis, the response time produced by the deployment of the evaluated workflow by using n *BB-VCs* in a parallel pattern called manager/worker (horizontal axis) for three different machines. As can be seen, the Meta-BBs of DagOnStar transparently manages the parallel patterns by changing the parameter of the number of workers (n). Moreover, the workflows could be deployed on different types of servers without IT staff performing a troubleshooting process. As expected, the more BB-VCs deployed on the infrastructure, the more response time observed by end-users to deploy the whole components of the studied workflow. Please note that the highest cost is associated with the building of the BB-VC basis image as the rest of the BB-VCs are clones, and the costs of these structures are not high. For instance, the deployment of a 64 BB-VC system was deployed on infrastructure in about three minutes.

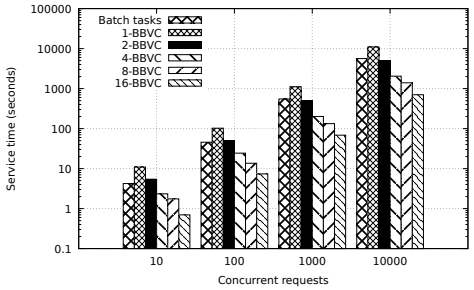

Fig. 9. Average service time for serving different number of concurrent task using different number of services in parallel per performed task.

C. Analyzing the performance of workflows built with Meta-BB

The results of the third phase are described in this section. In this phase, we evaluated the next configurations of the workflow:

- *Batch tasks*: This configuration represents the implementation of the workflow using the regular Batch tasks of DagOnStar.
- 1-*BBVC*: This configuration represents the implementation of the workflow by using microservices building blocks implemented in DagOnStar for each stage.
- n-*BBVC*: This configuration represents the implementation of *BBVC* but deploying parallel patterns for each stage, which improves the service time of processing the workload received by each stage.

To perform these experiments, we configure a bot to create and send artificial requests to the above-described configurations, which accepted and processed these requests as valid requests sent by real end-users. A request represents data sent by an IoT sensor to the workflow to be processed. Therefore, n concurrent requests represent n sensors transmitting data to the workflow (the size of data is homogeneous for all the experiments).

Figure 9 shows on it the vertical axis, the service time produced by each configuration of the workflow for attending a varying of the number of concurrent requests (horizontal axis). The results show that Batch tasks produce the best performance comparing with *1* Meta-BB configuration. The response time of *Batch tasks* for attending 10000 concurrent requests was of 93.51 minutes, and 183.5 minutes for *1* Meta-BB. The performance difference of these configurations represents the overhead produced by the microservice managing BB-VCs. Nevertheless, when comparing the *Batch tasks* with the *three Meta-BBs managing 16 BB-VC*, we can see that 16-BBVC configurations produced a percentage of performance gain 87% for an acceleration of 8x as this configuration processed the workload of 10000 concurrent requests in 11.42 minutes.

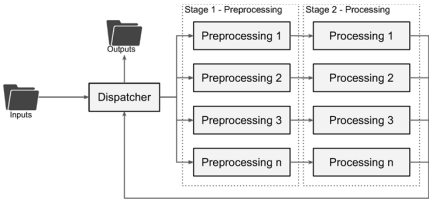

Fig. 10. Workflow for the processing of satellite imagery.

VII. Case Study II: Satellite imagery processing

This case study is based on the processing of satellite imagery captured by the satellite Landsat 8. We implemented a workflow composed of two main microservices(Meta-BB): the pre-processing stage and the processing stage.

A summary of each microservice designed is as follows:

1) *Preprocessing*: The pre-processing microservice calculates the radiation and top atmosphere reflectance (TOA) of each of the eleven bands of Landsat 8 images [25], [26].
2) *Processing*: The processing microservices include the calculation of the traditional indexes (e.g., NDVI, EVI, SAVI, MSAVI, NDMI, NBR, NBR2 and NDSI [25], [26]).

Figure 10 depicts the conceptual design of the workflow for the pre-processing and processing of the satellite imagery. The workflow was designed in the form of a Manager/Worker pattern. In this pattern, an entity called Manager is in charge of distributing, in a load-balanced manner, a set of images through different workers. The workers execute the two stages of the workflow (pre-processing and processing) and send the results back to the Manager.

A. Results

To test the performance of the pattern described in Figure 7, we used a repository of 20 images (of a size of 17.8 GB) captured by LandSat8.

Figure 11 shows in the vertical axis, the response time for the execution of the whole workflow created by using a different number of parallel BB-VCs (horizontal axis).

As it can be observed in Figure 11, the response time is reduced for each workflow when increasing the number of BBVC running in parallel. For instance, the pattern running 8 BBVC produces a response time of 3.83 minutes, whereas the pattern running with only one BBVC produces a response time of 25.03 minutes. This means an improvement in the response time of 84.69% comparing the pattern with eight parallel BBVC with the pattern with only one BBVC.

B. Performance comprison of the approach with state of art solutions

We performed a direct comparison of the proposed approach based on Meta-BBs and BB-VC with other two workflows engines in the literature: Parsl [8] and Makeflow [8]. The workflow used for testing was the one depicted in Figure 7 for the pre-processing and processing of satellite imagery.

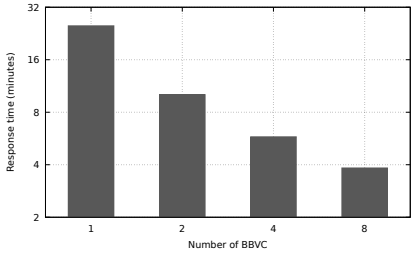

Fig. 11. Response time for the execution of the preprocessing and processing pattern.

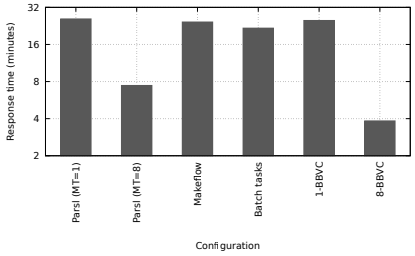

Fig. 12. Response time of the workflow implemented in different workflow engines.

The workflow executed with Makeflow was configured to run over virtual containers (parallel patterns are not available in this solution). For our approach, we tested two configurations of the workflow with one and eight BB-VC in the patterns inside of the Meta-BBs. We also configured Parsl to use two different configurations: the first one using one single thread and the second one using eight threads. We called this configurations as $Parsl(MT = 1)$ and $Parsl(MT = 8)$ respectively.

Figure 12 shows the response time (vertical axis) produced by the studied solutions for the processing of the 20 images.

As it can be seen in Figure 12, the approach using *8-BBVC* configuration produced the best performance compared with both configurations of Parsl and the solution developed with Makeflow. *8-BBVC* produced a response time of 3.83 minutes, whereas $Parsl(MT = 8)$ produced a response time of 7.43 minutes and *Makeflow* produced a response time of 24.34 minutes, which means that *8-BBVC* yielded an improvement in the response time of 48.42% and 84.25% in comparison with $Parsl(MT = 8)$ and *Makeflow* respectively.

VIII. Conclusions and future directions

In this paper, we presented a novel contribution in DagOnStar development, enabling the lightweight workflow engine to support the microservice technology in a VC context. As previously described in [5], DagOnStar has been designed with simplicity in mind focusing on production scenarios in the filed of massive data computational environmental science applications such as extreme weather predictions pollution forecasting [27], crowdsourced environmental data processing

[28] and distributed weather stations network data analysis [21].

The evaluation revealed the feasibility of using a building block approach to automatic deploy workflows in a microservice environment. It also revealed the flexibility of this approach to improve the performance of the workflows created by chaining *Meta-BB* in the form of parallelism patterns.

From the task typology point of view, we are going to integrate the DagOnStar microservice-based task interaction with GPU intensive computing tasks in virtualized [29] and containerized architectures even leveraging on mobile and single-board computing devices [30].

We plan to evolve DagOnStar in order to provide application live support to instrumented data acquisition platforms as vehicles, vessels, and drones [31] scenarios where the data processing workflow throughput is crucial because of the mission critical-like production conditions.

ACKNOWLEDGMENTS

The research project supported this research "DYNAMO: Distributed leisure Yacht-carried sensor-Network for Atmosphere and Marine data crowdsourcing applications" (DSTE373) and it is partially included in the framework of the project "MOQAP - Maritime Operation Quality Assurance Platform" and financed by Italian Ministry of Economic Development.

REFERENCES

[1] R. Madduri, K. Chard, R. Chard, L. Lacinski, A. Rodriguez, D. Sulakhe, D. Kelly, U. Dave, and I. Foster. The globus galaxies platform: delivering science gateways as a service. *Concurrency and Computation: Practice and Experience*, 27(16):4344–4360, 2015.

[2] Y. Babuji, A. Woodard, Z. Li, D. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I Foster, et al. Parsl: Pervasive parallel programming in python. In *28th HPDC*, pages 25–36. ACM, 2019.

[3] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *FGCS*, 25(5):541–551, 2009.

[4] M. P Singh and M. Vouk. Scientific workflows: scientific computing meets transactional workflows. In *NSF Workshop*, pages 28–34, 1996.

[5] R. Montella, D. Di Luccio, and S. Kosta. Dagon*: Executing direct acyclic graphs as parallel jobs on anything. In *WORKS 2018*, pages 64–73. IEEE, 2018.

[6] I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer, 2007.

[7] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[8] M. Albrecht, P. Donnelly, and D. Bui, Peter P. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on SWEET*, page 1. ACM, 2012.

[9] M. Wilde, M. Hategan, J. M Wozniak, B. Clifford, D. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[10] C. Zheng and D. Thain. Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *8th VTDC*, pages 31–38. ACM, 2015.

[11] 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 *5th DataCloud*, pages 25–32. IEEE Press, 2014.

[12] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In *SSDBM*, pages 471–481. Springer, 2010.

[13] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml) version 1.1. *yaml. org, Tech. Rep*, page 23, 2005.

[14] N. Dragoni, S. Giallorenzo, A. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[15] N. T. de Sousa, W. Hasselbring, T. Weber, and D. Kranzlmüller. Designing a generic research data infrastructure architecture with continuous software engineering. In *CSE 2018*. CEUR-WS. org, 2018.

[16] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *2007 IEEE Congress on Services (Services 2007)*, pages 199–206. IEEE, 2007.

[17] R. Montella, D. Kelly, W. Xiong, A. Brizius, J. Elliott, R. Madduri, K. Maheshwari, C. Porter, P. Vilter, M. Wilde, et al. Face-it: A science gateway for food security research. *Concurrency and Computation: Practice and Experience*, 27(16):4423–4436, 2015.

[18] D. Sánchez-Gallegos, JL Gonzalez-Compean, S. Alvarado-Barrientos, Victor J Sosa-Sosa, J. Tuxpan-Vargas, and J. Carretero. A containerized service for clustering and categorization of weather records in the cloud. In *2018 CSIT*, pages 26–31. IEEE, 2018.

[19] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *2005 ACM/IEEE Supercomputing*, page 54. IEEE Computer Society, 2005.

[20] J. L. Gonzalez, J. C. Perez, Vor J Sosa-Sosa, Luis M Sanchez, and B. Bergua. Skycds: A resilient content delivery service based on diversified cloud storage. *Simulation Modelling Practice and Theory*, 54:64–85, 2015.

[21] D. Sánchez-Gallegos, D. Di Luccio, JL. Gonzalez-Compean, and R. Montella. Internet of things orchestration using dagon* workflow engine. In *2019 WF-IoT*, pages 95–100. IEEE, 2019.

[22] G. Laccetti, R. Montella, C. Palmieri, and V. Pelliccia. The high performance internet of things: using gvirtus to share high-end gpus with arm based cluster computing nodes. In *PPAM*, pages 734–744. Springer, 2013.

[23] R. Montella, G. Agrillo, D. Mastrangelo, and M. Menna. A globus toolkit 4 based instrument service for environmental data acquisition and distribution. In *3rd Upgrade-cn*, pages 21–28. ACM, 2008.

[24] Dhiren Ghosh and Andrew Vogt. Outliers: An evaluation of methodologies. In *Joint statistical meetings*, pages 3455–3460, 2012.

[25] Gyanesh C., Brian L. M., and Dennis L. H. Summary of current radiometric calibration coefficients for landsat mss, tm, etm+, and eo-1 ali sensors. *Remote Sensing of Environment*, 113(5):893 – 903, 2009.

[26] Alexander Ariza. Descripción y correccin de productos landsat 8 ldcm (landsat data continuity mission), 2013.

[27] R. Montella, D. Di Luccio, P. Troiano, A. Riccio, A. Brizius, and I. Foster. Wacomm: A parallel water quality community model for pollutant transport and dispersion operational predictions. In *2016 SITIS*, pages 717–724. IEEE, 2016.

[28] R. Montella, D. Di Luccio, L. Marcellino, A. Galletti, S. Kosta, G. Giunta, and I. Foster. Workflow-based automatic processing for internet of floating things crowdsourced data. *FGCS*, 94:103–119, 2019.

[29] R. Montella, S. Kosta, D. Oro, J. Vera, C. Fernández, C. Palmieri, D. Di Luccio, G. Giunta, M. Lapegna, and G. Laccetti. Accelerating linux and android applications on low-power devices through remote gpgpu offloading. *Concurrency and Computation: Practice and Experience*, 29(24):e4286, 2017.

[30] R. Montella, C. Ferraro, S. Kosta, V. Pelliccia, and G. Giunta. Enabling android-based devices to high-end gpgpus. In *ICA3PP*, pages 118–125. Springer, 2016.

[31] R. Montella, S. Kosta, and I. Foster. Dynamo: Distributed leisure yacht-carried sensor-network for atmosphere and marine data crowdsourcing applications. In *2018 IC2E*, pages 333–339. IEEE, 2018.