DagOn*: Executing direct acyclic graphs as parallel jobs on anything

1st Raffaele Montella
Department of Science and Technologies
University of Naples "Parthenope"
Naples, Italy
raffaele.montella@uniparthenope.it

2nd Diana Di Luccio
Department of Science and Technologies
University of Naples "Parthenope"
Naples, Italy
diana.diluccio@uniparthenope.it

3rd Sokol Kosta
Department of Electronic Systems
Aalborg University Copenhagen
Copenhagen, Denmark
sok@cmi.aau.dk

***Abstract*—The democratization of computational resources, thanks to the advent of public, private, and hybrid clouds, changed the rules in many science fields. For decades, one of the effort of computer scientists and computer engineers was the development of tools able to simplify access to high-end computational resources by computational scientists. However, nowadays any science field can be considered "computational" as the availability of powerful, but easy to manage workflow engines, is crucial. In this work, we present DagOn* (Direct acyclic graph On anything), a lightweight Python library implementing a workflow engine able to execute parallel jobs represented by direct acyclic graphs on any combination of local machines, on-premise high performance computing clusters, containers, and cloud-based virtual infrastructures. We use a real-world production-level application for weather and marine forecasts to illustrate the use of this new workflow engine.**

***Index Terms*—Workflow, Data Intensive, Cloud Computing, Direct Acyclic Graph, Parallel Processing**

I. Introduction

The success of a novel emerging technology could be proven as it is used in a pervasive way by common people having no scientific or technical knowledge about how it works under the hood. Remarkable examples proving this concept could be found in Geographic Information Systems, Cloud Computing and Internet of Things based technologies used by common people in their everyday life.

Applying concept to science, we could state that at the time of writing, any science field that can be considered as *computational science* as opposed to a decade ago, when one of the many effort of computer scientists and computer engineers was how to make simpler the interaction of domain scientists with computational resources.

Following the same path, it is arguable that the next big thing will involve data science and parallel techniques to deal with big data.

Workflow engines play a primary role in computational sciences because the availability of cloud provided elastic and virtually infinite computational resources. In this scenario, the computation *orchestrator* acts as cornerstone in search of performance, affordability, reliability, availability and, above all, reproducibility of any computational experiment.

The massive use of computational models and data analysis techniques have been taking part in the scientific approach since the last few decades. In earth system sciences, material sciences, high energy, biological and medical studies, performing complex computations on heterogeneous facilities repeated on different data-sets has become one of the main investigation tools in support to (and sometimes in substitution of) field or laboratory activities.

The rise of machine learning techniques, above all the deep neural network, unchained the power of big data analysis and, at the same time, sped-up the need to manage computing power in a straightforward fashion.

Workflow engines for data-intensive science have existed since the beginning of the grid computing era [1], such as Triana [2], Taverna [3], Kepler [4], Unicore [5] and Pegasus [6].

In this paper, we present DagOn* (Direct acyclic graph On anything, named after the Phoenicians' god known by ancient Greeks as Triton; note that "*" symbol is the wild card for "anything") an open source Python library and related components enabling the execution of direct acyclic graph (DAG) jobs on anything, ranging from the local machine to virtual HPC clusters hosted on private, public or hybrid clouds.

In the rest of the paper, we will refer to a node of the DAG (which represents the workflow) as *task*. Each task must be wrapped by an execution envelope, ensuring the definition of a sandbox in which the software has to be executed. The task consumes input data and produces output data. The tasks are assembled using a specific scripting language. The parents/siblings relationship can be implicit when each dependence is automatically carried out from data dependencies or it must be declared in an explicit way trough the definition language.

DagOn* is a production-oriented workflow engine targeting computational scientists, and for this purpose it has been developed in order to meet following requirements:

- Integration in the Python environment;
- minimal footprint in terms of storage room used for external software components' execution sandbox;
- avoiding any workflow engine centered management of data (need to share the file system between computational nodes);
- easy definition of tasks with direct use of Python scripts, web interaction, external software components execution, cloud hosted virtual machines or containerized tools;
- execution sites independence.

The development of DagOn* has been motivated by our operational application for routinely produced weather and marine forecasts.

The rest of this document is organized as follows: in Section II, related work is discussed and the approach requirements are described; the motivations are carried the Section III where we state the DagOn* position respect to the related work; Section IV is about design strategies, parallelism, and the distribution approach; Section V describes the DagOn* application lifecycle in details; Section VI is about the real-world operational application that drove the development of DagOn*; and finally, Section VII gives conclusion remarks and draws the path for future research development.

II. Related Work

There is a plethora of available workflow engines devoting their features to enable the user to run complex applications involving up to millions of jobs on different computational resources [7]. Among the most recent literature there are a number of Python-based tools and libraries designed to support the workflows management as PaPy [8] a parallel and distributed data processing pipelines, PyCOMPSs [9] a framework to facilitates the development of parallel computational workflows, Fireworks [10] a dynamic workflow system designed for highthroughput applications and DASK [11] a parallel computing library designed for parallel analytics. The management strategy of each workflow engine is pretty similar [12].

At the best of our knowledge, we can consider Swift (II-A), Parsl (II-B) and Galaxy (II-C) as the workflow environments which can be related to the presented work.

A. Swift

Swift is a scripting language designed for composing application programs into parallel applications which can be executed on multi-core processors, clusters and cloud, using a C-like syntax with dataflow-driven semantics and implicit parallelism [13].

Unlike most the other scripting languages, Swift focuses on the issues that arise from the concurrent execution, composition, and coordination of many independent (and, typically, distributed) computational tasks. This language expresses the execution of programs that consume and produce file-resident dataset.

B. Parsl

Based on Swift [14] model, Parsl [15] is a software component leveraging on Python parallel scripting library, supporting asynchronous and implicitly parallel data-oriented workflows.

Parsl brings advanced parallel workflow capabilities to scripts (or applications) written in Python. Parsl scripts allow selected Python functions and external applications (called Apps) to be connected by shared input/output data objects into flexible parallel workflows.

Parsl abstracts the specific execution environment, allowing the same script to be executed on multi-core processors, clusters, clouds, and supercomputers [16].

When a Parsl script is run, the Parsl library causes annotated functions (Apps) to be intercepted by the Parsl execution fabric, which captures and serializes their parameters, analyzes their dependencies, and runs them on selected resources (sites).

The execution fabric brings dependency awareness to Apps by introducing data futures. A data future is a construct that includes the real result as well as the status and exceptions for that asynchronous function invocation.

If one App is responsible for writing a future, other Apps are blocked from reading it until it is written. This feature allows Apps to execute in parallel whenever they do not share dependencies or their data dependencies have been resolved [15].

C. Galaxy

Galaxy is a popular, web-based genomic workbench which enables users to perform computational data processing [17].

Galaxy has established a significant community of users and developers thanks to the approach focused on building a collaborative environment for performing complex analysis, with automatic and unobtrusive provenance tracking [18] allowing transparent sharing of not only the precise computational details underlying an analysis, but also intent, context, and narrative [19].

Galaxy Pages are the principal means to communicate research performed in Galaxy. The users can create their Pages as interactive and web-based documents describing a complete genomics experiment.

The computational experiments are documented and published with all computational outputs directly connected, allowing readers to view any level of experiment detail and reproduce some or all of the experiment extracting methods to be modified and reused [20].

Leveraging on the FACE-IT Globus Galaxy [21], in [22] it has been described how the Galaxy workflow engine has been extended in order to support earth system related applications focusing on agricultural and climate modelling, weather/marine forecasts production and food quality assessment for mussel farms [23].

III. Motivations

DagOn*, the workflow engine proposed in this paper, is the result of our experience in executing massive workflows on a heterogeneous infrastructure powered by diverse and different computing resources spread across local web farms and cloud facilities.

Because our application domain (both production and on demand [24] earth system modelling, Internet of Things data processing [25] and machine learning tasks [26]), Swift could represent a valid alternative to Galaxy.

Nevertheless, the Swift Parallel programming language is perfect for complex workflow description based on data flow, executing software across multiple sites and high performance task spawning, but still lacks in system integration.

Parsl learned the lesson from Swift, offering a workflow engine as Python library able to execute both Python tasks and external software in a straightforward fashion.

We learned the lesson from Swift, developing a data flow oriented workflow engine, but offering the task flow definition as a feature.

As Parsl, we offer our workflow as a Python library, but once a workflow has been defined, its representation as JSON file could be saved to be used via a web graphical user interface as Galaxy. DagOn* represents data files used by different external task using a custom name space identified by the workflow:// schema in order to resolve dependencies automatically.

The management of the staging operations is performed automatically by the workflow engine as the persistence of the scratch directories minimizing the storage footprint.

A DagOn*-based application could be developed using multiple workflows with reciprocal interaction. The developer can implement custom task components extending the ones already available.

As in Galaxy, the developer could implement data types wrapping the name space based data set management.

At the time of writing, DagOn* supports Python naive tasks, web tasks, batch tasks, SLURM tasks, AWS tasks, and container tasks.

IV. Design

DagOn* has two main components: a Python library implementing the application lifecycle at runtime and a service component for workflows monitoring and management (Fig. 1). A DagOn* application is developed as Python script using any Python extension with a regular sequential approach. Parallel tasks are defined using the **Task** component.

The DagOn* Runtime performs the interaction with the actual executors as on premises local or remote resources, virtual machines instanced on public/private/hybrid clouds and containers.

The Task component takes in charge the dependencies, the interaction with the workflow, the file staging system and it participates to the monitoring process in conjunction with the DagOn* Service.

More specialized Task types are designed as extensions of the Task component.

The different task types are classified as follows:

- **Native.** Regular Python function which are executed locally in a concurrent application thread. This kind of tasks have been designed in order to implement lightweight operations that are convenient to be executed in parallel with no need for a specific computing power or storage capacity. This type of task shares the same object scope of the DagOn* application.
- **Web.** Web tasks have been designed in order to enable the developer to make workflows interacting with remote resources accessible using web services. This task type performs SOAP web service consuming and API REST invocations. It waits for an external event and interacts with other DagOn* workflows leveraging on the DagOn* Service. Two or more DagOn* workflow apps can interact with each other using the web services produced by the DagOn* Service and this kind of task. In this scenario the task is able stop its execution until it receives data or other forms of notification via web-socket.
- **Batch.** These are external software components executed using SSH or a local scheduler. This kind of task component, alongside with the Cloud and the Container task types, is the base interface to external executors.
  While Native and Web Tasks are executed within the DagOn* application, Batch represents an external software component that can be executed on the same machine, on another machine in the same cluster or on any remote machine the user has the permissions to access and perform program execution.
  The external software is executed in a scratch directory acting as a sandbox for a customized running environment managed by the DagOn* garbage collector (IV-B). The data dependencies are defined using the DagOn* workflow:// schema (IV-A). The DagOn* design takes the benefit of the object oriented paradigm defining a plug-in architecture for the actual batch executors.
  The current DagOn* implementation enables the developer to use external software component running on the same machine that runs the DagOn* application in a naive way; it issues an SSH remote shell command or submits the job to SLURM [27] as local scheduler.
- **Cloud.** A task can be embedded in a virtual machine image and executed on the cloud. Cloud tasks leverage on cloud-specific libraries as Boto for Amazon Web Services[1] or cloud-generic components as Apache Libcloud footnotehttps://libcloud.apache.org using a cloud interface meta-model paradigm [28].
  When this kind of task is used, a virtual machine image has to be prepared and stored in the cloud image repository. The image has to provide the life support to the external software with the needed configuration for the stage operations. The virtual machine image can be prepared with the support of a shared file system, grid-ftp data transfer[2] or with just the regular secure copy.
  At the time of executing the task, a new virtual machine instance is created using the previously prepared virtual machine image. When the instance is up and running, a local scratch directory is created and data are staged in. Once data is correctly staged in the remote scratch directory, the actual external software is launched. The outputs can be explicitly staged out to a well known data sink or remain in the scratch directory until they are still needed.
  The DagOn* garbage collector (IV-B) takes charge of the virtual machine stop and/or terminate management.
- **Container.** A task can be represented by a container script and executed on a *containerized* infrastructure. Lightweight container technology has recently arisen as an alternative to complete virtualization saving consistent CPU and input/output costs [29]. This kind of task is

[1]https://aws.amazon.com/it/sdk-for-python/

[2]http://toolkit.globus.org/toolkit/docs/latest-stable/gridftp/

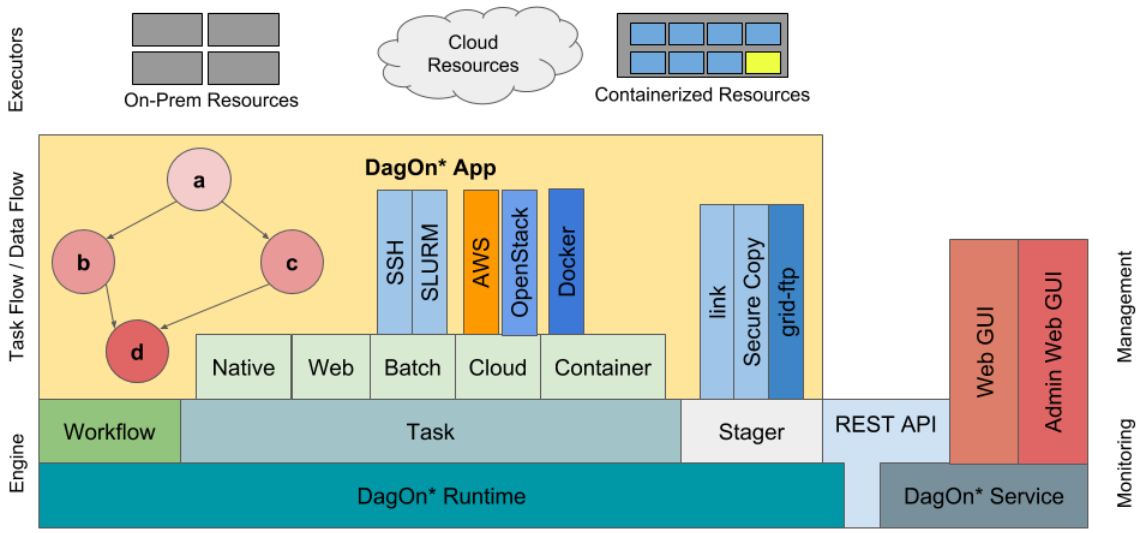

Fig. 1. The DagOn* architectural schema.

one of the most ambitious feature of DagOn*. While its specifications have been formally defined, the actual implementation is limited to a naive approach working with Docker scripts [30] wrapping each task at the workflow level and running each task inside a container. This approach limits the scope of the container while it has consistent costs for the startup and shutdown operations. At the time of executing a containerized task, the container manager takes charge of the container description script in which previously have been coded the features needed bind the external software component, the data transfer mechanism (shared file system, grid-ftp or secure copy), the creation of the scratch directory and the external software component itself. As the container is ready, the stage in process begins and when it is completed the external software is actually launched. As already described for the cloud task type, the outputs can be explicitly staged out to a well known data sink or remain in the scratch directory until they are still needed. The DagOn* garbage collector (IV-B) takes charge of the container shutdown management.

The DagOn* application developer can extend any Task component in order to design the app in terms of Tools, as in Galaxy, providing to the final user a more component oriented approach.

The **Workflow** component is the container for the tasks. The parallel relations between tasks can be defined explicitly (task flow) or using the DagOn* peculiar data dependence model (data flow) based on the workflow:// schema.

This component is in charge of evaluation of all the dependencies between tasks and manages the data movement from a task to another when those dependencies involve data.

The Workflow component performs the initial registration and the final de-registration of the managed workflow on the DagOn* Service. The same DagOn* application can be built using one or more Workflow instances. Each one is uniquely identified by a 128 bit universally unique identifier. In this way multiple workflows belonging to the same or different DagOn* applications can interact with each other via the DagOn* Service and the Web tasks.

The Workflow component has been designed to manage a checkpoint/resume feature.

The **Stager** component is not directly used by the DagOn* application developer, but it is managed by the Workflow component in order to mitigate the need of physically copy data produced by one task and used by another. At the time to perform the actual data stage in, this component selects automatically the data transfer model accordingly with the kind of task. This happens when all the dependencies are resolved and just before the external software component has to be run.

The Stager component performs a symbolic data link in order to avoid useless explicit data copying whether the external software component runs on the local machine or on a remote machine sharing the same file system. We consider this approach simplifies the overall workflow data management.

The Stager component can select the GridFTP copy if available or the regular secure copy of the external software component has to be executed on a remote machine, a virtual machine instanced in a public, private or hybrid cloud, or a container running in a local or dedicated infrastructure.

A. The workflow:// schema

The Batch component takes charge of the management of data dependencies using the DagOn* peculiar paradigm designed around the workflow:// schema.

As in previous similar solutions (i.e. Galaxy), the external software runs in a sandbox implemented as a scratch directory where a customized, high configurable environment is enforced. The shell script wrapping the external software provides the environment configuration such as the local directory structure, libraries and ancillary software components and configuration files.

Input files have to be staged (stage-in operation) in order to be processed. The outputs are produced in the form of files (or directories) considering the scratch directory as the root. Usually, at the end of the process the results have to be staged out (stage out operation) for the next usage.

DagOn* design considers a workflow:// schema as the root of current workflow a virtual file system. Under this conditions, workflow:///**task_unique_name/** is the root of the scratch directory created by the DagOn* Runtime.

If two or more workflows have to interact (regardless if they belong to the same or different applications) the use of the workflow:// schema remains consistent identifying any resource as workflow:///**workflow_uuid/task_unique_name/**.

The workflow:// schema can be used in conjunction with the DagOn* Service to receive web-socket based notifications.

B. The garbage collector

Batch tasks can be defined by one or more references to data file produced by previously executed tasks. Because multiple tasks can use data from one or more other tasks scratch directories, the DagOn* Runtime takes into account the usage of each task results.

The role of the DagOn* garbage collector is track the storage and computational resources allocated during tasks execution and proceed to dispose them when no longer needed. In case of an external software component executed on the local machine or on a remote machine sharing the same file system, when the task scratch directory is no longer needed (all depending tasks are successfully completed) the directory is removed, making the temporary storage available for new computations.

In the case the of cloud tasks, the garbage collector takes care of virtual machine stop or termination. If the same virtual machine stance have to be used by a single task or multiple ones, advanced disposing policies can be enforced. In a pay per use scenario, where the account is billed on hour basis, the policies can even manage billing issues using customized logic provided by the developer. For example, the garbage collector could remove the remote scratch directory, but take the virtual machine instance alive if there are no more queued similar tasks. In the case the rest of the hour is already payed and a new upcoming task could be executed saving the virtual machine instantiating and startup time.

When a virtual machine instance is not needed anymore and there are no conditions or enforced policies overriding default rules, the garbage collector stops and eventually terminates it.

For containerized task garbage collector behaves similarly as with cloud tasks. The difference is related about where the container wrapping the task is actually contained.

For containerized task the role of the garbage collector is similar to the one performed with cloud tasks.

C. DagOn* Service

As depicted in Fig. 1, the DagOn* Service is a software component not belonging to the DagOn* Runtime (delivered embedded in the DagOn* library). It may be deployed in a DagOn* powered infrastructure in a not mandatory way. This component is devoted to monitoring tasks interacting with the DagOn* Runtime using a REST API. This service can be used to track the status of a workflow or a task of a specified workflow, as described in the Section V about the DagOn* application lifecycle.

The service publishes a directory of active workflows and enables the DagOn* application developer to implement interactions between different workflows of the same application or belonging to different applications using the Web tasks.

A simple portal enables the users to check the workflows overall execution status, tasks success and failures, resource allocation and other insights.

The DagOn* Service has been designed with the idea of managing the user authentication and authorization, the resource allocation and the pay as you go issues.

V. Application Lifecycle

Due to previous experiences with Swift and FACE-IT Galaxy, we elected Python as scripting language to describe our workflow-based applications managed by the DagOn* Runtime. We defined a programming model (see Section V-A) in order to enable the embedding of one or more workflows in regular Python applications. Fig. 2 shows the DagOn* lifecycle using a really simplified application. In order to describe in details the lifecycle, we have to consider the interaction of the application with the DagOn* Runtime (Section V-B) and with the DagOn* Service (Section V-C). From the time axis point of view, in this example, we divide the whole lifecycle in 10 time steps.

A. Programming model

The following listing represents the schema of a simple DagOn* application implementing a typical diamond shaped direct acyclic graph:

```
import dagon
...
task_a=new SBatch("a","...")
task_b=new SBatch("b","... workflow:///a")
task_c=new SBatch("c","... workflow:///a")
task_d=new SBatch("d",
    "... workflow:///b workflow:///c")
...
workflow=Workflow("myapp",settings)
```

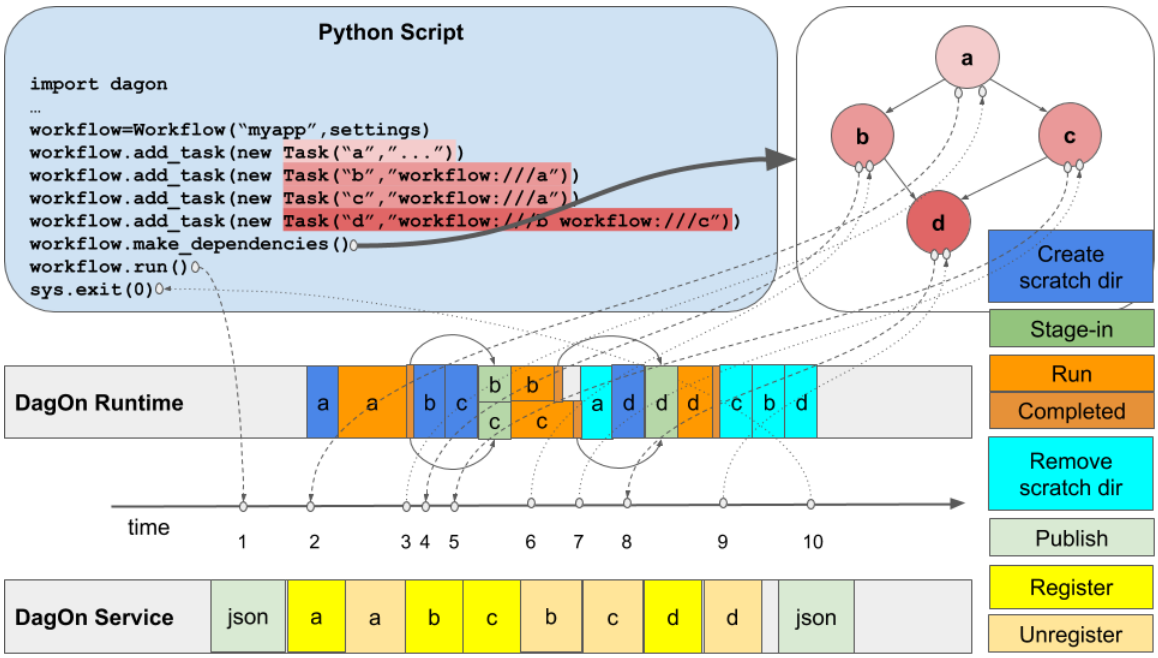

Fig. 2. The DagOn* programming model and the application life cycle as an interaction sequence between the direct acyclic graph parallel job execution, the DagOn* Runtime and the DagOn* Service.

```
workflow.add_task(task_a)
workflow.add_task(task_b)
workflow.add_task(task_c)
workflow.add_task(task_d)
workflow.make_dependencies()
workflow.run()
...
sys.exit(0)
```

The configuration part has been omitted in order to simplify the sample code.

The tasks named “a”, “b”, “c” and “d” are defined as SLURM tasks. Within the same workflow, the name of each single task must be unique. In this listing sample, the actual shell script name is not shown, while the arguments needed to execute the external software component are specified using the *workflow://* schema.

The task “a” is the workflow root and produces some data that will be used by the tasks “b” and “c”. Finally, the task “d” ends the workflow consuming data produced by “b” and “c”. In order to simplify the description of the workflow, we omitted that data produced by “d” are staged out to a data sink acting as final user.

Once all tasks are defined, an instance of the Workflow object has been created. This example is related to a workflow with task dependencies defined as data flow. Each task is added to the workflow and then the dependencies are resolved explicitly invoking a method.

In case of task flow oriented workflows, when a task is added to the workflow the list of the backward dependencies and forward dependencies has to be provided in an explicit fashion.

If all dependencies are coherent and consistent, the workflow is ready to be run invoking the homonyms blocking method. Leveraging on the Python environment, a DagOn* application can be divided in more workflows each of them dynamically arranged implementing branching and looping.

B. Runtime

The DagOn* Runtime performs the following basic tasks:

- Create Scratch Directory. A scratch directory is created when an external software component has to be executed on a local or remote machine. This operation ensures the software component has its own isolation at the best of the chosen execution context.
- Stage in. In order to be processed, the needed data have to be available to the external software component. The stage in operation can be a simple local symbolic link or a fine high performance data transfer between remote resources. This operation is managed by the Stager component once the workflow:// schema is enforced in order to define the needed transfer operations.
- Run. This is the crucial task operation. Once the task is completed successfully, the related depending tasks can start their execution.
- Remove Scratch Directory. This operation is managed by the garbage collector.

Step 1 is fully managed with a DagOn* Service interaction. At the timestep 2, the local scheduler SLURM is ready to execute the task “a”.

A scratch directory is created: the job representing the task is executed on the selected resource and terminate with success at the time step 3.

At the timesteps 4 and 5, the local scheduler is ready to run the jobs related to the tasks “b” and “c” and the related scratch directories are created. The tasks “b” and “c” need for data produced by task “a”. The *workflow://* schema references are resolved and the Stager component performs the stage in operation in a concurrent way.

Once data are in the tasks scratch directory, each task is executed ending with success (time steps 6 and 7).

At time step 7, both tasks “b” and “c” are completed: the data produced by task “a” is no longer needed and its scratch directory is deleted.

Meanwhile, the job representing the task “d” is ready to be executed by the local scheduler and its scratch directory is created at time step 8. Task “d” needs for data produced by tasks “b” and “c”. The references are resolved using the *workflow://* schema and the Stager component transfers data in the task “d” scratch directory.

When all data have been transferred, the task “d” is executed and successfully completed (time step 9).

Finally, at time step 10 the scratch directories of the tasks “b”, “c” and “d” are disposed.

C. Service

The interactions between the Runtime and the Service are performed transparently from the point of view of the application developer using REST APIs.

The DagOn* Service performs the following basic tasks:

- Publish. The workflow configuration is acquired by the service and made available using REST APIs. This operation is performed both at the workflow execution start and at its termination.
- Register. A task registers to the service updating its status each time there is a change in the execution lifecycle.
- Unregister. Once a task is completed, it removes itself from the service.

At the time step 1 the DagOn* Runtime registers the workflow on the DagOn* Service using the JSON representation. In this way the service can provide updates about the workflow status.

At time step 2 the task “a” performs its registration to the service and un-registers itself at the step 3 when it is completed. Then, the tasks “b” and “c” perform their register/unregister operations between the time steps 4 and 8. The task “d” notifies about its status changes in the time steps 8 and 9. Finally, at time step 10 the DagOn* Runtime interacts with the Service pushing its status as terminated.

VI. Use case: providing operational weather/marine forecasts

The scientific workflow shown in Fig. 4 has been configured using DagOn* workflow engine. It is operational at Campania Center for Atmospheric and Sea Monitoring and Modelling[3] (CCMMMA) with the aim to provide high resolution meteo-marine forecasts focused on Center and South Thyrrenian Sea,

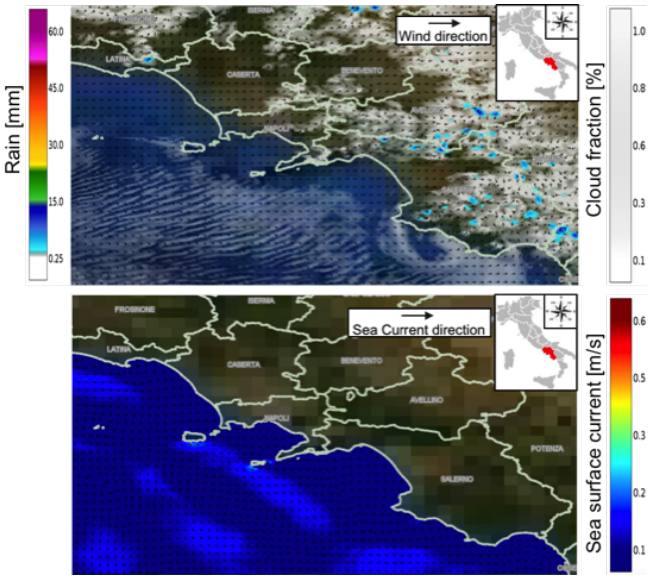

Fig. 3. DagOn* real case application with WRF (weather forecasts map at the top) and ROMS (sea surface current map at the bottom) off-line coupling. The sample forecasts are related to Campania Region (d03 computational domain) at 2018-08-08 Z12 UTC (simulations initialized at 2018-08-08 Z00 UTC).

Eest Sector, and other services relative to atmosphere and sea management [31]. The configured model chain uses an off-line coupling approach in which the outputs of each involved numerical model are initial or forcing conditions to the next component (see the blocks and the relative connections in Fig. 4).

The model chain starting point is Weather Research and Forecast (WRF) model [32], [33], which provides hourly (external time step of output) weather forecasts necessary to force the air quality model chain components (blue dotted line in Fig. 4) and the hydrodynamic components (pink dotted line in Fig. 4). The WRF model is initialized every three hours (according with the global dataset availability), with initial and boundary conditions produced by the National Centers for Environmental Prediction (NCEP) Global Forecast System (GFS) [34]. An example of the output of this part of the workflow application is visible in the upper side of Fig. 3.

Hydrodynamics is evaluated by two different models driven by the same forcing conditions (10m wind speed and direction provided hourly by the WRF model):

- the third generation model WaveWatch III (WW3) [35], [36], for the sea wave component;
- the Regional Ocean Model System (ROMS) [37], for the sea temperature, salinity and current components.

The water column initial and boundary conditions, necessary to ROMS model, are referred to COPERNICUS service[4].

The ROMS model is coupled with the Lagrangian model WaComM (Water quality Community Model) [38] to forecast the circulation pattern of tracers spilled out by coastal sources.

[3]http://meteo.uniparthenope.it

[4]http://marine.copernicus.eu

The lower side of Fig. 3 represents the output of the ROMS model from our DagOn* workflow.

The models addressed to high resolution air quality forecasts are CALMET [39], [40], a diagnostic 3-dimensional meteorological model, and CHIMERE [41], a model designed to produce hourly forecasts of ozone, aerosols and other pollutants.

Currently this DagOn* workflow empowers three operational applications:

1) The circulation pattern tracers in the Lagrangian WaComM model are associated, using an empirical trend curve, to potential pollutants, with the aim to evaluate their interaction with the mussel farms [42], [43].
2) In the context of coastal management during a sea storm event, the framework Son Of Beach (SoB) takes the hourly offshore wave conditions provided by WW3 model as input. It computes the beach run-up in order to forecast the coastal vulnerability related to the coastal flooding during the extreme weather events [44].
3) CALMET provides the atmospheric pattern driving the air pollution during a geo-localized accidental or maliciously induced wildfire [45].

The computational domain spatial discretization (grid model) supported by the different models and the different forecasting scales justifies the introduction in the workflow of some tools (grey blocks in Fig. 4) to support the data intensive processing (interpolation, rotation, re-sampling on different space and time etc.).

Regardless the scale (spatial resolution) of the different models, we can identify three computational domains with increasing spatial resolution: the d01 extended to European continent (and focusing on the Mediterranean Sea area); the d02 at intermediate scale including the Italian seas; the d03 at regional scale focused on the Center and South Thyrrenian Sea, Eest Sector.

All models have 1-hour external time step (different internal time step (Δt) for each model and domain), and almost all models (WRF, ROMS, WW3, CHIMERE and WaComM) support the restart mode to prevent them to cold start from a quiet physic condition at each simulation. In the described workflow configuration, each 24 simulated hours the results are published on-line and the models continue their run starting from archived restart point.

Tab. I summarizes the data size and the computational power involved in each domain for WRF and ROMS models for one single hour of simulation results. The computational demand is remarkable especially considering that all processing is done daily in a production fashion. This justifies the use of a dedicated on premises computational infrastructure and storage facility that can be elastically extended leveraging on public cloud resources [46].

VII. Conclusion and future directions

In this work DagOn*, a Python based tool for data intensive scientific workflows targeting production applications, has been introduced. The DagOn* programming model enables the advanced user to embed workflows in already existing Python scientific applications.

The DagOn* architecture design (Section IV) and the programming model and application lifecycle (Section V) has been described focusing on its peculiar features as the dependencies management via the workflow:// schema and the garbage collector.

The DagOn* design and development has been primary driven by our previous experiences with FACE-IT Globus Galaxy, Swift and by the needs of our real world use case application for weather and marine forecasts production as shown in Section VI.

DagOn* is far to be considered a mature product ready for prime time and some features still need to be implemented or reinforced as short term goals.

The security, authentication and authorization on a pool of computing resources have to be centralized and managed with a single sign on system component and a credential repository.

The checkpoint and recovery workflow features have to be implemented in the next development iterations. Finally, the DagOn* service needs for improvements in web user interface, REST APIs, usage data analysis and resource allocation management.

The cloud task interface has to be improved in order to enforce the plug-in approach and mitigate the effects due to the leveling down of the features offered by diverse and different cloud providers and cloud management APIs.

The integration between the DagOn* workflows and the container based execution technologies is one of the most intriguing and improvable designed, but partially implemented, features. Several other possible approaches for workflow task execution into containers have been already considered in DagOn* design. A different and more effective way to integrate DagOn* workflows and containers could be implemented and tested as future research directions. One possible log term perspective could be running several instances of external software component on the same container eliminating the startup overheads.

A serious compare and contrast evaluation versus Swift and Parsl is in our future plan, as the evaluation of possible integration with Parsl.

Acknowledgments

This work has been supported by the University of Napoli Parthenope, Italy (project DSTE333 “Modelling Mytilus Farming System with Enhanced Web Technologies” funded by Campania Region/Veterinary sector).

References

[1] R. Lovas, G. Dózsa, P. Kacsuk, N. Podhorszki, and D. Drótos, “Workflow support for complex grid applications: Integrated and portal solutions,” in *Grid Computing*. Springer, 2004, pp. 129–138.

[2] I. Taylor, M. Shields, I. Wang, and A. Harrison, “The triana workflow environment: Architecture and applications,” in *Workflows for e-Science*. Springer, 2007, pp. 320–339.

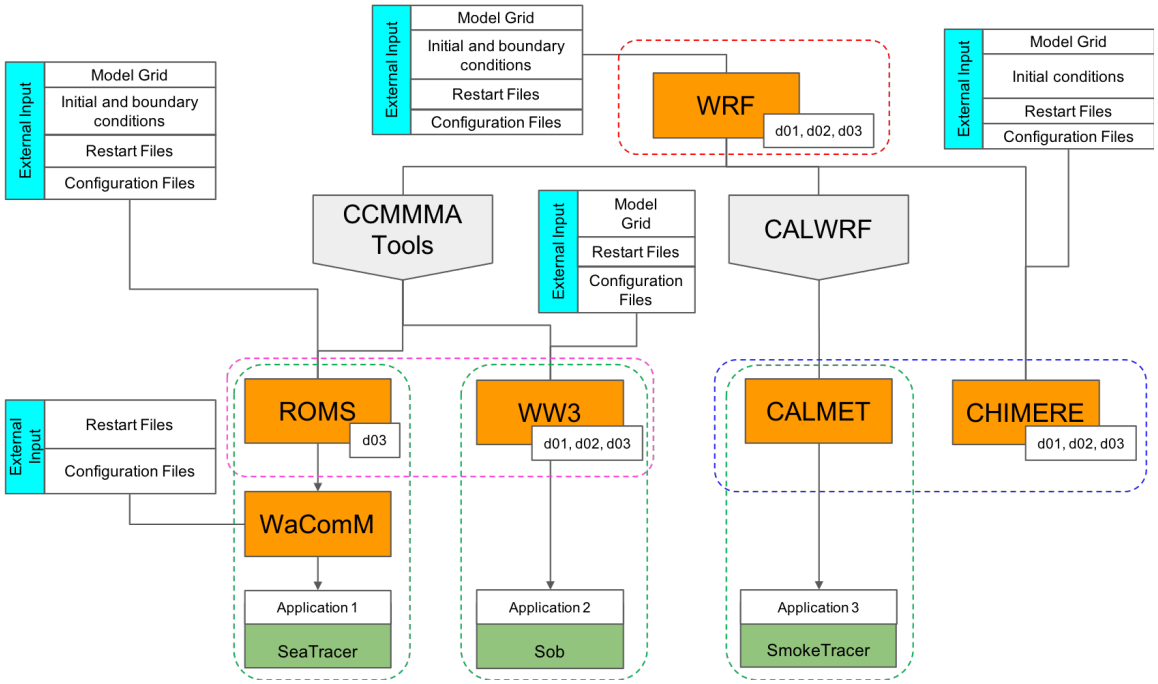

Fig. 4. Block diagram of a scientific workflow to provide operational weather/marine forecast.

Model	Domain	Δt [sec.]	Grid dimension [N. of cells]			Grid spacing [m]	Output File Dimension [Gb] (1-hour of simulation)	N. of output file for each run (7 days)	N. of MPI processes for each Δt	Execution time [mm:ss] (1-hour of simulation)
			West-East	South-North	Bottom-Top					
WRF	d01	150	230	209	28	25000	0.11	169	96 (12 core*8 nodes)	05:05
	d02	30	361	336		5000	0.29			
	d03	6	301	306		1000	0.22			
ROMS	d03	30	1375	1021	30	200	2.28		80 (10 core*4 nodes)	07:12

TABLE I
A PARTIAL SUMMARY OF SIMULATION PRODUCTS' DATA SIZE AND INVOLVED COMPUTING POWER

[3] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. W1, pp. W557–W561, 2013.

[4] D. Barseghian, I. Altintas, M. B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. T. Borer, E. W. Seabloom *et al.*, "Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access and analysis," *Ecological Informatics*, vol. 5, no. 1, pp. 42–50, 2010.

[5] S. Gesing, I. Márton, G. Birkenheuer, B. Schuller, R. Grunzke, J. Krüger, S. Breuers, D. Blunk, G. Fels, L. Packschies *et al.*, "Workflow interoperability in a grid portal for molecular simulations," in *Proceedings of the International Workshop on Science Gateways (IWSG10)*, 2010, pp. 44–48.

[6] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[7] A. Barker and J. Van Hemert, "Scientific workflow: a survey and research directions," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2007, pp. 746–753.

[8] M. Cieslik and C. Mura, "Papy: Parallel and distributed data-processing pipelines in python," *arXiv preprint arXiv:1407.4378*, 2014.

[9] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[10] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier *et al.*, "Fireworks: A dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015.

[11] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, no. 130-136. Citeseer, 2015.

[12] L. Liu, M. Zhang, Y. Lin, and L. Qin, "A survey on workflow management and scheduling in cloud computing," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 837–846.

[13] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[14] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 95–102.

[15] Y. Babuji, K. Chard, I. Foster, D. S. Katz, M. Wilde, A. Woodard, and J. Wozniak, "Parsl: Scalable parallel scripting in python," in *10th International Workshop on Science Gateways*, 2018.

[16] Y. Babuji, A. Brizius, K. Chard, I. Foster, D. Katz, M. Wilde, and J. Wozniak, "Introducing parsl: A python parallel scripting library," 2017.

[17] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[18] E. Afgan, A. Lonie, J. Taylor, K. Skala, and N. Goonasekera, "Architectural models for deploying and running virtual laboratories in the cloud," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on*. IEEE, 2016, pp. 282–286.

[19] C. Sloggett, N. Goonasekera, and E. Afgan, "Bioblend: automating pipeline analyses within galaxy and cloudman," *Bioinformatics*, vol. 29, no. 13, pp. 1685–1686, 2013.

[20] B. Kim, T. Ali, C. Lijeron, E. Afgan, and K. Krampis, "Bio-docklets: virtualization containers for single-step execution of ngs pipelines," *GigaScience*, 2017.

[21] R. K. Madduri, D. Sulakhe, L. Lacinski, B. Liu, A. Rodriguez, K. Chard, U. J. Dave, and I. T. Foster, "Experiences building globus genomics: a next-generation sequencing analysis service using galaxy, globus, and amazon web services," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 13, pp. 2266–2279, 2014.

[22] 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*, vol. 27, no. 16, pp. 4423–4436, 2015.

[23] R. Montella, A. Brizius, D. Di Luccio, C. Porter, J. Elliot, R. Madduri, D. Kelly, A. Riccio, and I. Foster, "Using the face-it portal and workflow engine for operational food quality prediction and assessment: An application to mussel farms monitoring in the bay of napoli, italy," *Future Generation Computer Systems*, 2018.

[24] E. Chianese, A. Galletti, G. Giunta, T. Landi, L. Marcellino, R. Montella, and A. Riccio, "Spatiotemporally resolved ambient particulate matter concentration by fusing observational data and ensemble chemical transport model simulations," *Ecological Modelling*, vol. 385, pp. 173–181, 2018.

[25] R. Montella, S. Kosta, and I. Foster, "Dynamo: Distributed leisure yacht-carried sensor-network for atmosphere and marine data crowdsourcing applications," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 333–339.

[26] R. Montella, A. Petrosino, and V. Santopietro, "Whoareyou (way): A mobile cuda powered picture id card recognition system," in *International Conference on Image Analysis and Processing*. Springer, 2017, pp. 375–382.

[27] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[28] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A precise metamodel for open cloud computing interface," in *8th IEEE International Conference on Cloud Computing (CLOUD 2015)*, 2015, pp. 852–859.

[29] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makeflow, work queue, and docker," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 2015, pp. 31–38.

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

[31] G. Benassai, P. Aucelli, G. Budillon, M. De Stefano, D. Di Luccio, G. Di Paola, R. Montella, L. Mucerino, M. Sica, and M. Pennetta, "Rip current evidence by hydrodynamic simulations, bathymetric surveys and uav observation," *Natural Hazards and Earth System Sciences*, vol. 17, no. 9, pp. 1493–1503, 2017.

[32] W. C. Skamarock and J. B. Klemp, "A time-split nonhydrostatic atmospheric model for weather research and forecasting applications," *Journal of Computational Physics*, vol. 227, no. 7, pp. 3465–3485, 2008.

[33] J. G. Powers, J. B. Klemp, W. C. Skamarock, C. A. Davis, J. Dudhia, D. O. Gill, J. L. Coen, D. J. Gochis, R. Ahmadov, S. E. Peckham *et al.*, "The weather research and forecasting model: Overview, system efforts, and future directions," *Bulletin of the American Meteorological Society*, vol. 98, no. 8, pp. 1717–1737, 2017.

[34] S. Saha, S. Nadiga, C. Thiaw, J. Wang, W. Wang, Q. Zhang, H. Van den Dool, H.-L. Pan, S. Moorthi, D. Behringer *et al.*, "The ncep climate forecast system," *Journal of Climate*, vol. 19, no. 15, pp. 3483–3517, 2006.

[35] H. L. Tolman, "The numerical model wavewatch: a third generation model for hindcasting of wind waves on tides in shelf seas," 1989.

[36] H. Tolman, "Distributed-memory concepts in the wave model wavewatch iii," *Parallel Computing*, vol. 28, no. 1, pp. 35–52, 2002.

[37] A. F. Shchepetkin and J. C. McWilliams, "The regional oceanic modeling system (roms): a split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean modelling*, vol. 9, no. 4, pp. 347–404, 2005.

[38] 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 *Signal-Image Technology & Internet-Based Systems (SITIS), 2016 12th International Conference on*. IEEE, 2016, pp. 717–724.

[39] J. S. Scire, F. R. Robe, M. E. Fernau, and R. J. Yamartino, "A users guide for the calmet meteorological model," *Earth Tech, USA*, vol. 37, 2000.

[40] L. Morales, F. Lang, and C. Mattar, "Mesoscale wind speed simulation using calmet model and reanalysis information: An application to wind potential," *Renewable Energy*, vol. 48, pp. 57–71, 2012.

[41] S. Mailler, L. Menut, D. Khvorostyanov, M. Valari, F. Couvidat, G. Siour, S. Turquety, R. Briant, P. Tuccella, B. Bessagnet *et al.*, "Chimere-2017: from urban to hemispheric chemistry-transport modeling," *Geoscientific Model Development*, vol. 10, no. 6, pp. 2397–2423, 2017.

[42] A. Galletti, R. Montella, L. Marcellino, A. Riccio, D. Di Luccio, A. Brizius, and I. T. Foster, "Numerical and implementation issues in food quality modeling for human diseases prevention." in *HEALTHINF*, 2017, pp. 526–534.

[43] D. Di Luccio, A. Galletti, L. Marcellino, A. Riccio, R. Montella, and A. Brizius, "Some remarks about a community open source lagrangian pollutant transport and dispersion model," *Procedia Computer Science*, vol. 113, pp. 490–495, 2017.

[44] D. Di Luccio, G. Benassai, G. Budillon, L. Mucerino, R. Montella, and E. Pugliese Carratelli, "Wave run-up prediction and observation in a micro-tidal beach," *Natural Hazards and Earth System Sciences Discussions*, vol. 2017, pp. 1–21, 2017. [Online]. Available: https://www.nat-hazards-earth-syst-sci-discuss.net/nhess-2017-252/

[45] G. Agrillo, E. Chianese, A. Riccio, and A. Zinzi, "Modeling and characterization of air pollution: Perspectives and recent developments with a focus on the campania region (southern italy)," *International Journal of Environmental Research*, vol. 7, no. 4, pp. 909–916, 2013.

[46] R. Montella and I. Foster, "Using hybrid grid/cloud computing technologies for environmental data elastic storage, processing, and provisioning," in *Handbook of Cloud Computing*. Springer, 2010, pp. 595–618.