IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 27 March 2023, accepted 30 April 2023, date of publication 2 May 2023, date of current version 10 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3272619

RESEARCH ARTICLE

CWL-PLAS: Task Workflows Assisted by Data Science Cloud Platforms

ANDREA DETTI[1,2], (Member, IEEE), LUDOVICO FUNARI[1], LUCA PETRUCCI[1], MICHELE DÓRAZIO[1,3], ARIANNA MENCATTINI[1,3], AND EUGENIO MARTINELLI[1,3]

[1]Department of Electronic Engineering, University of Rome Tor Vergata, 00133 Rome, Italy
[2]National Inter-University Consortium for Telecommunications (CNIT), 43124 Parma, Italy
[3]Interdisciplinary Center of Advanced Study of Organ-on-Chip and Lab-on-Chip Applications (IC-LOC), 00133 Rome, Italy

Corresponding author: Andrea Detti (andrea.detti@uniroma2.it)

This work was supported in part by GÉANT Innovation Programme 2021 under the framework of the Platformed Workflow (PLAS) project.

ABSTRACT The Common Workflow Language (CWL) is a platform-independent description language for the representation of data science workflows consisting of a set of tasks that interact with each other to perform scientific analysis. The tasks can be packaged as Linux containers. On the one hand, using containers ensures the reproducibility and portability of workflows. Still, on the other hand, it limits each task to exploiting, at most, the resources of the host where its container runs. In this paper, we propose CWL-PLAS, an extension of CWL that allows a task to instantiate and temporarily use a supporting cloud platform for parallel computing, which is specialized for the task's activity. In this way, tasks can leverage the resources of multiple hosts in parallel, reducing the duration of the workflow. We implemented an open-source workflow manager that supports CWL-PLAS workflows and exploits a Kubernetes back-end. We used this workflow manager to evaluate the performance of CWL-PLAS in a couple of machine learning workflows.

INDEX TERMS Common workflow language, workflow management software, distributed computing, cloud.

I. INTRODUCTION

Big data analysis processes usually consist of a set of tasks, written in different languages, that interact with each other to extract insights and patterns from the data. Tasks take in external data or data produced by upstream tasks and provide their outputs to downstream tasks or ultimately to the user. The set of tasks along with their dependencies is called *workflow*, and there are many workflow managers [1].

Users describe their tasks and related data dependencies through configuration files, which use a workflow description language specific to the workflow manager in use. Consequently, the workflow manager uses these files to automate task execution through an underlying computing infrastructure [2], [3]. One of the main decision-making processes of workflow managers is the scheduling of when a task should start and how many resources to reserve for it. The scheduling decision takes into account that a task cannot start before all its inputs are available. Moreover, other constraints can be included in the problem, such as the availability of limited resources or the need to respect a completion deadline. In addition, a specific objective function is pursued, such as minimizing the time needed to complete the workflow, aka makespan, or monetary cost given a completion deadline, etc. Consequently, many scheduling algorithms have been proposed [4].

Managing data analysis activities through a workflow manager makes them reproducible, portable, maintainable, and shareable, as it allows complex processes consisting of multiple steps to be formalized [5]. Indeed, the use of workflow management systems is increasingly popular for data-intensive analyzes, such as those in bioinformatics, astronomy, and medical imaging processing [6], [7], [8].

The presence of many workflow managers using proprietary description languages results in the difficulty of bringing workflows into different environments, creating a kind of platform lock-in. For this reason, the Common Workflow

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

Language (CWL), a platform-independent workflow description specification, was introduced in 2016 [9], [10].

CWL relies heavily on Linux containers, which are lightweight isolated software packages that contain an application with all its dependencies, such as Docker and Singularity [11], [12]. Each task consists of the execution of a *CommandLineTool* (e.g. Python software) running in a container that includes all necessary binaries and libraries. In this way, the task takes its specific runtime environment with it and can be executed in any Linux-based host, thus ensuring portable and reproducible executions.

The use of CWL requires the user to prepare container images of his tasks and upload them to an accessible repository. Next, the user should describe: i) each task with a `.cwl` file, according to the ''CWL CommandLineTool Description'' standard; ii) the relationships the tasks have in the workflow in another `.cwl` file, according to the ''CWL Workflow Description'' standard. The `.cwl` files use the YAML format, and the CWL standards specify the schema and semantics of their contents. Finally, these `.cwl` files are passed to a CWL workflow manager that executes the workflow using a back-end computer system.

The CWL standard is gaining attention from the scientific community, and several free/open source workflow managers are available that support different backends,[1] including local hosts, public/private clouds offering virtual machines (e.g., AWS, Azure, Openstack, GCP), High-Performance Computing (HPC) systems (e.g., Slurm, PBS), and clusters based on Kubernetes [13], which is the most widely used container orchestration platform today to manage and automate the deployment, scaling, and management of containerized applications across clusters of real or virtual hosts.

In addition, cloud providers have also begun offering the CWL workflow manager as a service (e.g., Amazon Genomics), which is another sign of the growing interest in this technology.

This paper is motivated by the fact that we found that CWL runs the risk of not taking full advantage of the computational resources offered by a cluster of computing resources, such as a Kubernetes cluster. This prompted us to devise an extension of the CWL model, called PLAS (PLAtformed-taskS), which we present in this paper. The overall solution, called CWL-PLAS, aims to reduce the time to complete a workflow, known as makespan, by allowing every task to leverage on-demand parallel computing platforms to distribute its workload over many nodes.

Let us now briefly describe our intuition. A CWL workflow has the shape of a directed acyclic graph (DAG). Tasks that do not have data dependency on each other can be run in parallel by the scheduler, and thus take advantage of distributed computing resources. For example, Fig. 1a shows an example of a workflow consisting of 5 tasks (T1...T5), packaged as Linux containers and whose back-end Kubernetes cluster has 5 nodes where tasks' containers can be executed. Task T1 receives the input data from an external source and is executed by the scheduler on node 1. When the processing of T1 is finished, T2 and T4 use the output data of T1 and the scheduler decides to run them in parallel on nodes 2 and 3. T3 is then executed on node 2 after T2, taking the output of T2 as input. Finally, T5 is executed on node 5, taking the output data of T3 and T4 as input. In this workflow, only T4 can run in parallel with T2 and T3, so the maximum concurrency achieved consists of the parallel exploitation of only two out of five nodes.

The problem with this limited concurrency is based on the fact that packaging a task as a Container limits its execution in a single host, and this limitation is independent of the workflow scheduling strategy, since it considers tasks as a single scheduling unit. Consequently, to allow a single task to take advantage of more than one host, the innovation introduced by CWL-PLAS is to create a new task type, named *platformed-task*, which concurrenlty uses multiple containers to perform its job. From the scheduler's point of view, a platformed-task is seen as a single task, so the proposed scheduling strategies can be easily reused.

The set of containers used by a platformed-task can be crafted by the user, or the user can use existing parallel computing platforms. Fig. 1b shows how the PLAS extension modifies the workflow in Fig. 1a. Without lacking generality, we assumed that T4 is a Machine Learning training task whose trained neural network is passed to T5. For training neural networks, there are several distributed computing frameworks[2] that accelerate the training phase by using different hosts in parallel, such as Apache Spark [14] or Horovod [15]. With CWL-PLAS, the legacy[3] task T4 can be implemented as a *platformed*-task *p*T4, which temporarily installs an Apache Spark or Horovod ''sidecar'' platform on 4 cluster nodes and uses it to train the neural network in parallel and faster. When the platformed-task ends, the sidecar platform is removed.[4] In this case, the maximum concurrency consists of parallel exploitation of the entire set of five nodes.

Prior to this work, to the best of our knowledge, CWL did not include the use of sidecar platforms that would allow a single task to distribute its workload across multiple nodes. In this work, we propose this extension to CWL with the goal of reducing the makespan of workflows by enabling the exploitation of the horizontal scalability feature offered by existing cloud technologies. With a couple of lab experiments, we demonstrate that our proposal goes in this direction and finally provide the CWL community with a way to use it immediately through an open-source implementation [16].

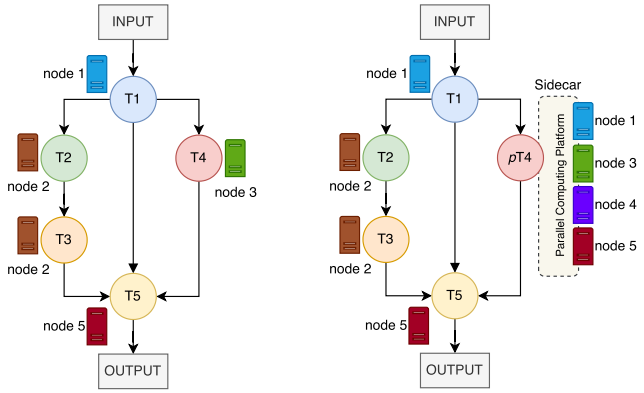
Accordingly, the main contributions of this paper are as follows.

[1] https://www.commonwl.org/implementations/ (Accessed 2023-03-21)

[2] We use the terms platform and framework interchangeably.

[3] We use the term *legacy* to refer to the type of tasks currently included in the CWL standard.

[4] A platform used by a platformed-task is called ''sidecar'' because it resembles a sidecar attached to a motorcycle.



(a) Example of a CWL workflow with legacy tasks

(b) Example of a CWL workflow with platformed-tasks

FIGURE 1. Comparison of CWL workflow with legacy tasks and platformed-tasks. A single task is represented with a circle, and arrows represent dependencies between tasks.

- The proposal for an extension of CWL that reduces the time needed to complete single tasks that can leverage parallel computing frameworks and, consequently, the makespan of related workflows (Sec. III-A).
- The design of an architecture that implements this extension and uses a Kubernetes backend (Sec. III-B)
- An open-source implementation of this architecture for Kubernetes cluster [16]
- A performance assessment based on Machine Learning workflows (Sec. IV)

II. RELATED WORK

A. WORKFLOW MANAGEMENT SYSTEMS

Scientific workflows prove effective for programming complex data analysis processes at a high level and executing them on heterogeneous platforms, including supercomputers, or on distributed computing systems such as those offered by Grids and Clouds infrastructures. As a result, many workflow management systems (WMSs) have been developed to define and automate workflow execution.

We have broadly classified WMSs into two groups. The first group consists of WMSs that offer a graphical interface to users to edit and execute workflows. Workflows are made up of sequences of tasks, where each task is an analysis tool available in an internal portfolio of the WMS [19], [20], [26], [28]. Usually, introducing a new tool into the portfolio requires interaction between the user and the WMS administrator, because the tool is command-line software running on the host operating system that must have the necessary execution environment. However, some managers The majority of the listed WMS supports Docker containers as dependency resolver, while some are also compatible with Singularity.

Moreover, Pegasus [18] also provides more tailored and sophisticated scheduling capabilities to reduce data movements and start-up overhead between tasks with small execution time compared to their start-up times merging them into a single higher granularity task.

While the constraint of using analysis tools available in a portfolio is a limitation, it can also allow for an abstract, or semantic, representation of workflow tasks, i.e., the user specifies what is to be done on a dataset without specifying the software that implements this task. This abstract representation makes the workflow capable of being executed on a variety of platforms, which implement the abstract task portfolio with different software and optimized with respect to the computing infrastructure used. This abstraction feature is either native to the WMS or can be added by an overlay layer that maps the abstract tasks to specific implementations, the updating of which (e.g., due to software obsolescence) has no impact on the workflow description making it future-proof [31], [32].

A second group of WMSs is made of systems that enable the implementation of workflows through specific programming languages, e.g., Python, Go, C-like [17], [18], [21], [22], [23], [24], [25], [27], [29], [30] by providing users with classes and function libraries through which WMS services can be accessed, such as input/output data storage or specialized computing functions. Some of these handlers can use Linux containers to package task software, thus not imposing the use of a specific language for task implementation.

Unlike CWL, for both groups, there is no decoupling between the workflow description and the WMS, thus risking platform lock-in, and unlike CWL-PLAS in this paper, containerized tasks, where allowed, can leverage the resources of only one host. In addition, for the second group, the absence of the GUI requires the researcher to know the programming language used by the WMS to define the workflow, even if it uses built-in tasks.

Workflow Description Language (WDL) and Common Workflow Language (CWL) are two vendor-neutral workflow specification languages that aim to make workflows independent of the management systems that run them. CWL emphasizes a bit more reproducibility and portability of workflows, thus requiring more verbosity in workflow description. On the contrary, WDL emphasizes human readability of the workflow and an easy learning curve, but provides users with reduced expressiveness [33]. Both do not provide a semantic abstraction of workflow activities; in fact, the software that performs the task must be specified by the user.

From a conceptual point of view, both languages allow the underlying schedulers to implement parallel execution of different tasks, i.e., tasks that have no data dependency can be executed in parallel on different hosts. In addition to this type of *inter-task parallelism* offered by workflow schedulers, with CWL-PLAS, we promote the concept of *intra-task* parallelism, i.e., a single task can distribute its workload over several hosts in parallel.

Most workflow management systems and specification languages assume that the workflow is run on one site at a time, which can be a local computer as much as a remote HPC

TABLE 1. Comparison with the most popular Workflow Manager System.

	CWL	Execution Envs	UI	Container Support	Workflow as a Service	Intra-task Parallelism
CWL-PLAS	Yes	Cloud, Local	CLI	Docker	Yes	Yes
StreamFlow [17]	Partially (need additional conf. file)	Cloud, HPC, Local	CLI	Docker, Singularity	Partially (not in case of intra-task parallelism)	Yes
Pegasus [18]	No	Grid, Cloud, HPC	CLI	Docker, Singularity, Shifter	Yes	No
Galaxy [19]	Yes	Cloud, HPC, Local	Web, CLI	No	Yes	No
Kepler [20]	No	HPC, Local	Desktop	No	Yes	No
AiiDA [21]	No	HPC, Local	CLI	Docker, Singularity	Yes	No
doit [22]	No	Local	CLI	No	No	No
SciPipe [23]	No	Local	CLI	Singularity	Yes	No
Swift [24]	No	Cloud, HPC	CLI	No	No	No
BEE [25]	Yes	Local, Cloud, HPC	CLI	Docker, Singularity	Yes	No
Arvados [26]	Yes	Local, Cloud, HPC	CLI, Web	Docker, Singularity	Yes	No
Toil [27]	Yes	Local, Cloud, HPC	CLI	Docker, Singularity	No	No
CWL-Airflow [28]	Yes	Local, Cloud, HPC	Web, CLI	Docker, Singularity	Yes	No
Calrissian [29]	Yes	Local, Cloud	CLI	Docker	No	No
Cromwell [30]	No	Local, Cloud, HPC	CLI	Docker	Yes	No

or cloud cluster. Some recent work advocates the concept of multi-site workflow management. For instance, StreamFlow [17], [34] is a multi-site workflow management system that can distribute the tasks of a single workflow across multiple runtime environments, ranging from bare-metal hosts to multi-container Kubernetes environments, possibly geographically distributed. StreamFlow supports CWL for workflow description, but.cwl YAML files must be complemented by StreamFlow-specific configuration files to bind each CWL task to StreamFlow runtime environments. StreamFlow configures and uses these runtime environments through specific *Connectors* that have control rights to the back-end computing platform.

CWL-PLAS can be placed at a lower service level than StreamFlow or other similar multi-site WMSs [25] because it integrates the execution of multi-container tasks directly into the CWL framework without using other overarching solutions. This integration allows cloud service providers to offer ''as-a-service'' execution of CWL workflows, preventing users from interacting with the underlying computing infrastructure. In addition, the approach of using on-demand sidecar platforms whose lifecycle is tied to that of the task using them is another feature of CWL-PLAS toward ''as-a-service'' provisioning, as it supports multi-tenancy more efficiently. In fact, a sidecar platform occupies computing, memory, and storage resources, only for as long as it takes to execute the associated task.[5] For example, this type of workflow-as-a-service model is currently offered by the GÉANT Cloud Flow (GCF) platform [35], which exposes a Workflow Execution Services (WES) compliant with the Global Alliance for Genomics and Health (GA4GH) API [36]. To take advantage of the attractive cross-site services offered by StreamFlow, Task Execution Service supporting CWL-PLAS (e.g., our modified TESK) can be integrated into the StreamFlow framework through a specific Connector.

Tab. 1 compares the main characteristics of the aforementioned most popular workflow management systems with CWL-PLAS. The table shows one of the main strengths of CWL-PLAS, namely the ability to exploit a sidecar platform to enable intra-task parallelism. Another feature we value in the table is the ability of a cloud provider to implement a workflow-as-a-service model with the related WMS, as previously discussed. We highlight [17] is partially capable of executing workflow-as-a-service, because if a user wants to execute tasks that use intra-task parallelism, must have access to the back-end Kubernetes cluster. Furthermore, [19], [25], [26], [27], [29], [37] support the ability to define the

[5]For instance, with StreamFlow those who want to run workflow must have access to the Kubernetes cluster to preventively deploy static Helm platforms in case he wants to use multi-container tasks. This is not a problem in private scenarios, but could be limiting in the case of cloud providers who wish to offer workflow execution as-a-service.

workflow via CWL while [17] supports it partially as it requires additionally configuration file to execute the workflow. It's important to note that CWL-PLAS, at least in our current implementation, has some limitations when considering its features and capabilities. One such limitation is the current lack of HPC support and Singularity containers. In addition, CWL-PLAS does not provide a graphical user interface, but rather a command line interface for interacting with workflows. However, the design of CWL-PLAS doesn't prevent future extensions or software add-ons to include missing features.

B. WORKFLOW SCHEDULING

Workflow scheduling is a decision-making process of a WMS that manages the execution of tasks in a workflow, ensuring that each task is executed in the correct order and at the right time. There are many different types of workflow schedulers available, ranging from simple task scheduling tools to more complex platforms that support parallel execution, resource management, and other advanced features to optimize one or more objectives.

A baseline scheduling strategy, which we can call *best-effort* as it ensures no guarantee, provides that the scheduler does not control any resource quota for the tasks, but merely asks the underlying computing platform (e.g., a single host or Kubernetes cluster) to execute a task when all its input data are ready.

There are many research papers on the topic of workflow scheduling that differ in the objective functions and/or the proposed strategy to pursue the objective efficiently, keeping in mind that, usually, the resulting scheduling problem is NP-complete [4], [38]. Some schedulers aim to minimize the workflow makespan given a resource or monetary budget [39], others to minimize the monetary cost given a completion deadline to respect [40], [41]. To pursue their objective function different schedulers use different strategies. For instance, [40], [41] partition the workflow into paths of dependency tasks based on specific criteria and then a scheduling decision is made for each path. The schedulers proposed in [39] and [42] create groups of tasks (akin super-task), rank them and perform the scheduling decision. Similarly, [18], [19], [20], [24] can cluster small tasks into larger, more compute-intensive ones and this is especially helpful when executing workflows with thousands of tasks, where the start-up time overwhelms the overall computation.

For any scheduler we know, the smallest unit of scheduling is the task. Thus, a scheduling strategy can improve the utilization of a cluster of resources by running tasks in parallel, i.e. addressing an inter-task parallelism problem. CWL-PLAS enables intra-task parallelisms, i.e., it tries to reduce the completion time of a single task by allowing its workload to be distributed over several nodes of a cloud cluster. Consequently, a scheduler is not intended to pursue the goals of CWL-PLAS, and vice versa, because they address two complementary domains: intra-task (CWL-PLAS) and inter-task (scheduler).

A platformed-task is seen by a scheduler as a single task, so scheduling strategies and platformed-tasks can work perfectly well together. In fact, in our implementation of CWL-PLAS [16], based on cwl-TES [43], we did not even have to extend the scheduling algorithm. The use of platformed-tasks can improve the resource utilization of a cluster when the problem is related to the fact that inter-task dependencies impose that only a few tasks can be executed in parallel by a scheduler, thereby not allowing full exploitation of the cluster nodes. In fact, in these cases, resource utilization efficiency can be recovered with platformed-tasks by distributing the workload of the single task over multiple nodes.

C. CLOUD TECHNOLOGIES

Workflow management systems (WMS) are increasingly embracing cloud technologies to make workflow execution fast and cost-effective. In addition to plain virtual machines, cloud providers offer as-a-service thousands of software and platforms that simplify the deployment of different categories of applications. Regarding the WMS category, databases and file/object repositories are undeniably useful for moving data from one task to another and for storing inputs and outputs. Moreover, two computing paradigms offered by the cloud are well suited to WMSs, namely: container and High-Performance Computing (HPC).

Containers are an exceptionally convenient tool for making software autonomous from the system on which it runs. A Container is a runtime environment, decoupled from the host operating system, in which the user runs his or her software along with all the necessary libraries. This environment is prepared independently by the user in the form of a container *image* and then run on a host. Several containerized software can run on the same host without the risk of any conflicts and without the system administrator having to install any files in the host operating system.

Unlike virtual machines, containers provide a system *isolation* rather than virtualization, and this allows them to have two outstanding features. First, a negligible overhead on host resource consumption compared to the case of native execution, i.e., running container software directly in the host operating system. Second, the performance is almost equal to that of native execution.

Containers can be executed in a single host or in a cluster of nodes. In the latter case, the complexity of dealing with a distributed system is alleviated by container orchestration systems, and Kubernetes is the most widely used [44]. Users request Kubernetes to run a Container and Kubernetes packages it into a resource unit named *Pod*, and then runs it on a cluster node. To support multi-container applications, Kubernetes provides Pod-to-Pod networking and DNS services. The number of Pods executing the same software can be replicated, even automatically. Replicated Pods are used in parallel to serve ''different'' requests, making better use of cluster resources.

Many complex applications are developed as multi-container applications, where each container performs a specific job, and containers interact with each other through the network to serve a user request. This is the case of microservices applications [45] or distributed frameworks for parallel computing, such as Apache Spark [14] or Horovod [15] for machine learning. Since these applications require the presence of many Pods and other Kubernetes resources, their all-at-once deployment is usually supported by another tool called Helm [46]. Helm is a package manager that uses specific files called *charts* to describe the set of Kubernetes resources needed to run a multi-container application. A user uses Helm by passing it the chart of the application he wishes to execute. Consequently, Helm interacts with the Kubernetes control plane to deploy the application resources in the cluster nodes. Helm charts can be made by hand or downloaded from public Helm repositories maintained by software developers/companies that provide charts for their distributed applications. There are about 10,000 public charts today, demonstrating the popularity of Helm.[6]

High-Performance Computing (HPC) platforms are used to run computationally demanding processes in a distributed cluster of nodes [47]. These processes are implemented by programs using parallel computing libraries that run directly in the host OS. The level of parallelism achieved is generally finer than that achievable using Kubernetes Pod replication. In fact, the set of instructions used to serve a ''single'' request can be executed by CPUs from different nodes, while using a memory space made common by a very fast network message-passing system. This extremely fine level of parallelism allows very high performance to be achieved but requires, on the one hand, advanced programming skills and, on the other hand, an underlying HPC platform that allows the execution of distributed instructions in near real-time to avoid bottleneck. For these reasons, HPC platforms usually provide the means to allocate exclusive access to compute resources of nodes and use ultra-low latency network solutions, such as InfiniBand. Note that recently, cloud providers have begun offering HPC as-a-service infrastructure [48].

The HPC and Kubernetes worlds are still some distance apart, partly because they start from a different model of resource sharing. HPC wants performance guarantees and tends to strictly control access to resources (CPU, mem, network) to avoid ''interference'' between processes. Kubernetes is more tolerant and allows concurrent use of hardware resources from containerized software, which inevitably creates uncertainties about execution time. When data transfers are limited and occur after large amounts of computation done by parallel workers (coarse-grained parallelism), these uncertainties/differences in execution time between workers do not slow down the workflow significantly, while they can be dramatic in the case of instruction-level, fine-grained parallelism [49]. In summary, the choice of technology, HPC or Kubernetes, depends on the parallel model of the application. If coarse-grained parallelism is sufficient, Kubernetes has the advantage of making users more autonomous through containers, and infrastructure hardware is usually cheaper. For fine-grained parallelism, HPC systems are mandatory unless massive resource overprovisioning is adopted.

Both HPC and Kubernetes environments can be used as computing backends of CWL workflow managers, and there are implementations that use either or both. In this paper, we focus on Kubernetes and propose an evolution of CWL that speeds up tasks that can be implemented with coarse-grained parallelism.

III. CWL-PLAS

A. EXTENSION OF THE COMMON WORKFLOW LANGUAGE

A CWL task consists of executing a command line tool, such as a Python program, whose input arguments are taken from a variety of sources, including local or remote file repositories. The tool may be present in the host operating system or within a Linux container, temporarily instantiated to execute the command, and later removed. When the execution is complete, one or more output files are produced. When a task belongs to a workflow, its output files can be input arguments for downstream tasks.

Fig. 2 shows an example of a CWL workflow consisting of two tasks. The figure also includes the `.cwl` files that describe the workflow and the tasks.

The `workflow.cwl` file describes a workflow that takes an input file (`input-wf`) from a remote repository and uploads a output file (`output-wf`) to the same repository. The workflow involves the execution of two steps during which tasks 1 and 2 are executed, respectively. The input1 of task1 corresponds to the input file of the workflow, and the output1 of task2 corresponds to the output of the workflow.

The file `task1.cwl` describes a task1 that belongs to the CommandLineTool class.[7] Task execution requires the initialization of a Docker container based on the `myImageA` image. This container provides the execution environment for the command '`python3 myProgA.py`' that is the task1 job. Task1 has two output files: `output1` and `output2`; the former is passed to task2, while the latter is not used downstream. For example, `output2` can be a log file for debugging purposes. We have not reported the file `task2.cwl`, as it has the same content as `task1.cwl`, but it uses `myImageB` and `myProgB.py`.

The CWL-PLAS system extends the CWL task model by introducing the concept of *platformed-task*. Fig. 3 shows an example of a platformed-task with the corresponding `.cwl` file. A platformed-task consists of a container, called *Executor*, and another set of containers that form the sidecar platform. The Executor runs a software tool (e.g., `myParProg.py`) that implements the task job, which interacts with the sidecar platform to parallelize the computation using the different workers of the platform.

[6] https://artifacthub.io/stats (Accessed 2023-03-21)

[7] Other classes exist, but we have limited the description of CWL to what is necessary to present the paper's contribution.

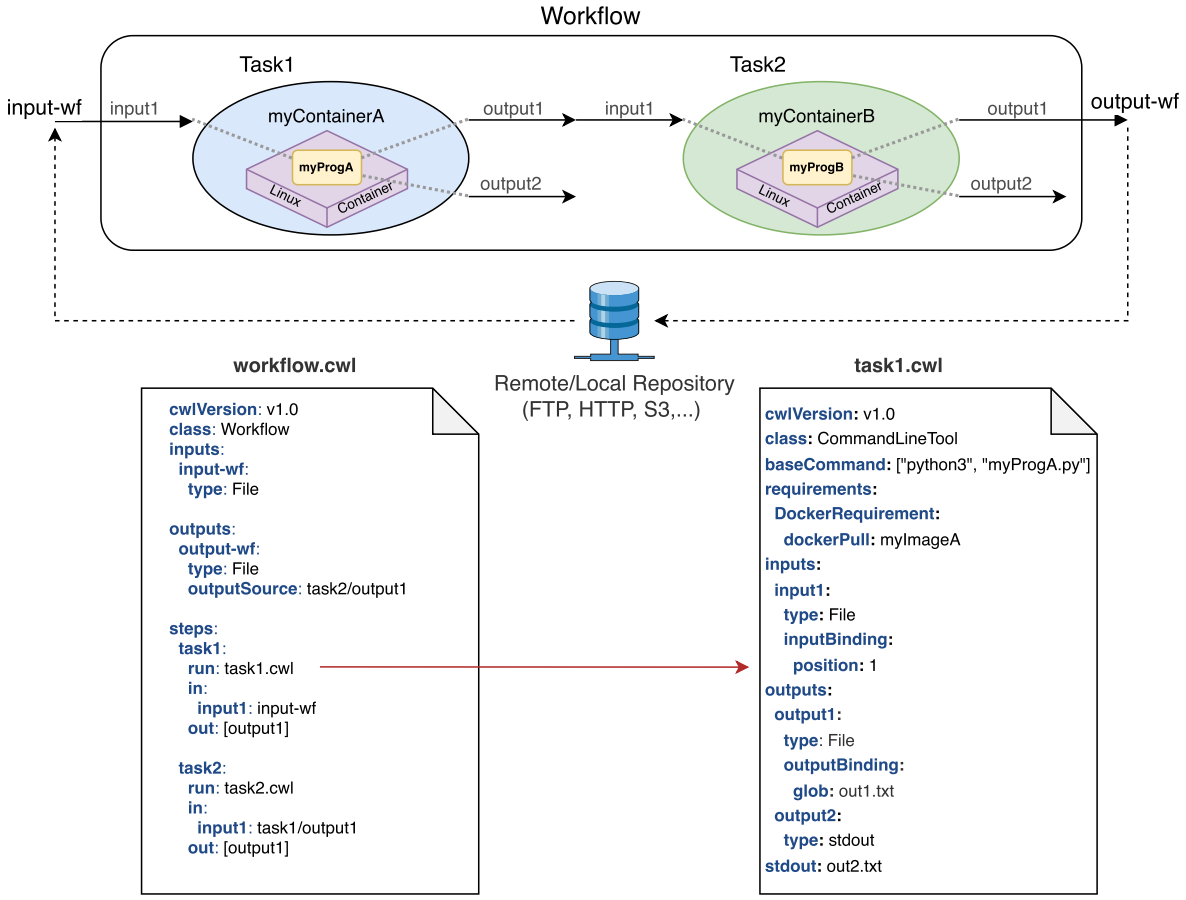


FIGURE 2. CWL workflow with two legacy tasks.

CWL-PLAS uses Helm to control the lifecycle of sidecar platforms. As can be seen in the file `ptask1.cwl` of Fig. 3, CWL-PLAS extends the "CWL Command Line Tool Description" specification by introducing a new requirement class called `HelmRequirement`, whose properties specify repository, name, and version of the Helm chart of the sidecar platform. Moreover, the Docker image of the *Executor* container is provided with the legacy `DockerRequirement` key. This schema is the result of a compromise between (i) limiting the impact on the existing CWL schema, so that current implementations do not have to be significantly modified to support our extension, and (ii) being flexible to introduce support for other sidecar platforms into the CWL, for example, by simply adding a `DockerCompose` requirement in a future implementation that aims to support also Docker-based multi-container sidecar platforms [50].

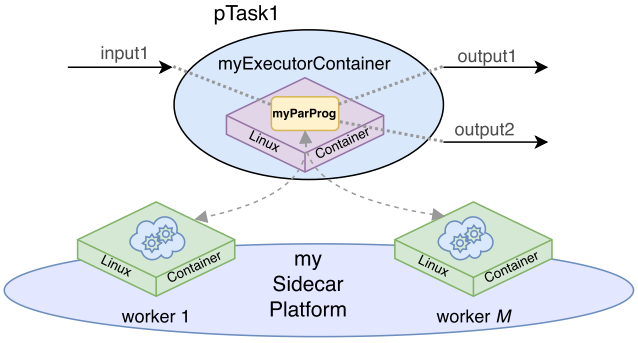
B. THE CWL-PLASK WORKFLOW MANAGER

We have devised an open-source CWL-PLAS workflow manager that uses a Kubernetes back-end [16]. We named this workflow manager CWL-PLASK, where the suffix *K* is a reference to Kubernetes.

Fig. 4 shows the architecture of the CWL-PLASK workflow manager. The architecture is an extension of that used by the Global Alliance for Genomics and Health (GA4GH), and the extension consists of:

- extension of the Task Execution Service (TES) APIs
- software update for handling sidecar platforms within the cwl-TES workflow manager and within a Kubernetes-based TES implementation named TESK [51].

Users access workflow services through a REST API exposed by a server, named cwl-WES, that implements the GA4GH WES-API [36]. The internal engine of cwl-WES is cwl-TES [43], a command-line workflow manager that receives as input a `.cwl` workflow file and requests the execution of embedded tasks to a back-end Task Execution Service (TES) that exposes the GA4GH TES API [52]. In fact, a TES is meant for the execution of single tasks, whereas workflow orchestration is a cwl-TES job. The cwl-TES workload manager uses a file repository for storing input and output files of tasks, and the repository can be an FTP server, an AWS S3 storage, etc. The scheduling strategy used by cwl-TES resembles the best-effort one we mentioned in Sec. II-B: when a task has all its input files ready, it is



ptask1.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: ["python3", "myParProg.py"]
requirements:
 DockerRequirement:
  dockerPull: myExecutorImage
 HelmRequirement:
  chartRepo: "https://myChartRepo"
  chartName: "mySidecarPlatform"
  chartVersion: "1.0.0"
inputs:
 input1:
  type: File
  inputBinding:
   position: 1
outputs:
 output1:
  type: File
  outputBinding:
   glob: out1.txt
 output2:
  type: stdout
stdout: out2.txt
```

FIGURE 3. Platformed-task.

executed by contacting a back-end TES and passing it information about the task to be executed, the URI where to find the input files, and the URI where to put the output files at the end of the task. When the task is completed, the control returns to the cwl-TES, which eventually executes other tasks waiting the output of the task that just finished. We note that this scheduler is agnostic with respect to which server (or servers in the case of platformed-task) the task is executed on, and this lower-level scheduling decision is made by the specific implementation of the Task Execution Service.

A Task Execution Service (TES) can have different implementations depending on the underlying computing resources it uses. To implement CWL-PLASK, we chose and extended TESK [51], an implementation of a TES that uses a Kubernetes cluster for task execution. The TES API is managed by a persistent server, called TESK-API, which runs in a Kubernetes Pod and orchestrates the execution of tasks as required by the upstream cwl-TES. These tasks can be legacy tasks or new platformed-tasks, as in the case of Fig. 4.

When the TESK-API receives a request to execute a task, it starts a Kubernetes Job called Taskmaster.[8] The Taskmaster is a kind of orchestrator dedicated for a single task that allocates and deallocates over time the Kubernetes resources needed to execute the task. The life-cycle of these resources is as follows:

1) a Task Volume is created for use as a shared file repository among the Pods/Jobs of the task;
2) an Input Job is deployed. The Job software retrieves input data from an INPUT repository, copies them to the Task Volume, and then the Job finishes;
3) the sidecar platform for parallel computing is deployed by using a Helm chart;
4) an Executor Job is created. The Job executes the command-line tool that implements the task logic within an Executor Container. To speed up/parallelize the computation, the command-line tool interacts with the sidecar platform. The Executor and sidecar platform use Task Volume to read input files and write output files. When the execution of the command-line tool is completed, the Executor Job terminates, and the Taskmaster removes the sidecar platform;
5) an Output Job is deployed. The Job takes the output files from the Task Volume and uploads them to an OUTPUT repository. When the uploading is complete, the Output Job ends;
6) finally, the Task Volume is removed, and the Taskmaster Job terminates, thereby deallocating all Kubernetes resources used for the task.

Next, TESK-API notifies the upstream `cwl-TES` workflow manager that the task has ended, and the workflow manager will continue the workflow management by requesting the execution of subsequent tasks. When all the tasks have been executed, the workflow manager notifies the user, who will access the output files through an OUTPUT repository.

We conclude the section by commenting on our choice to bind the life-cycle of the sidecar platform with that of the related task. This may seem a sub-optimal choice, considering that the sidecar platform might be reused later by subsequent tasks. In such cases, it might be convenient to leave the platform active, thus saving the delay of instantiating it at task startup (aka *cold-start* delay). Apart from the fact that this *persistency* feature can be added in future implementations of the workflow manager, we have found that, in the long term, all the Kubernetes nodes contain the image of the platform containers, so starting them is very fast and related delay results negligible compared to the task duration, considering also that we are focusing on applications that use coarse-grained parallelism. In addition, removing the sidecar platform at the end of the task fits well with the workflow-as-a-service model that a cloud provider can offer, avoiding

[8] Differently from a Kubernetes Pod that is intended to execute a long-term service, a Job is designed to reliably execute a short-term software program within a Container. When the program is completed, the Job is terminated and Kubernetes keeps track of successful completions.

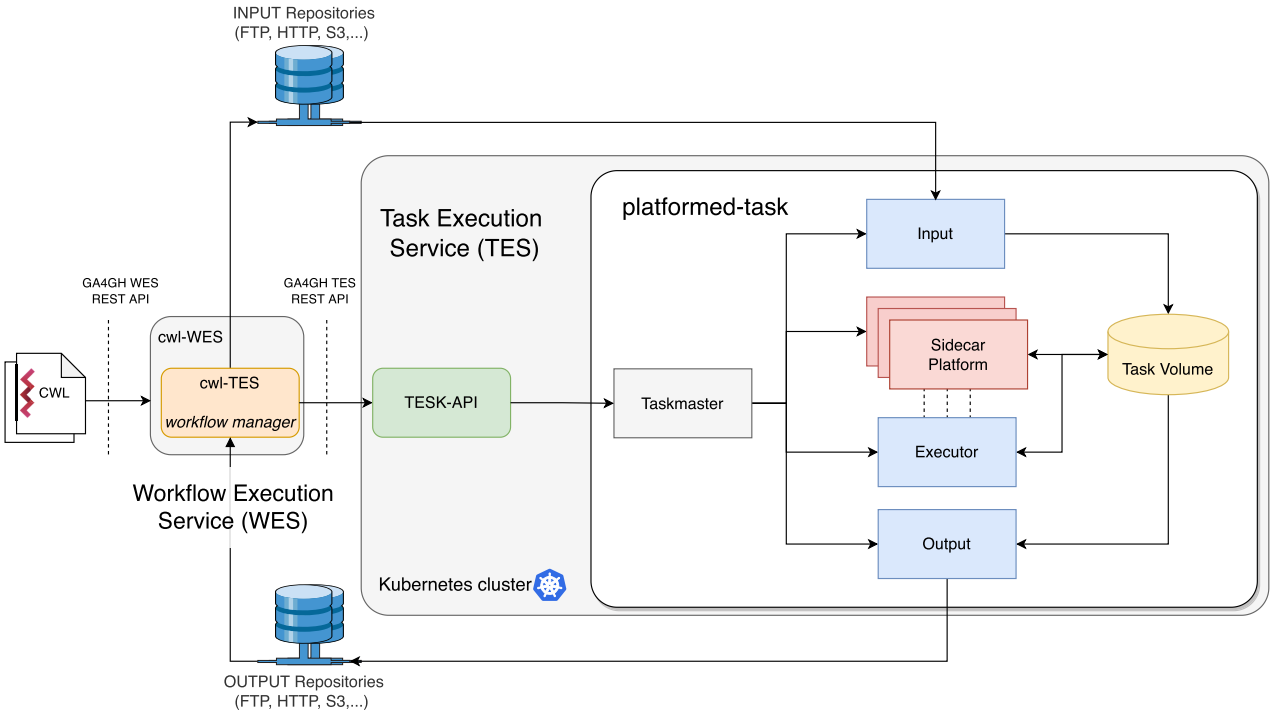


FIGURE 4. CWL-PLASK workflow manager.

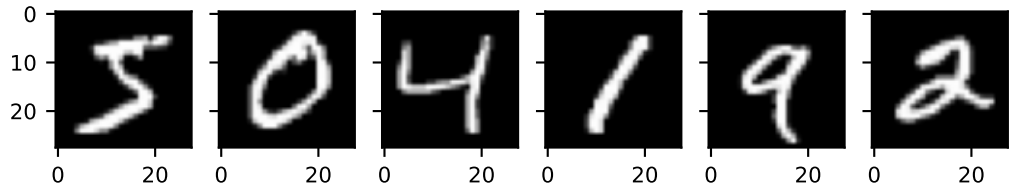


FIGURE 5. Example of data in the MNIST dataset.

```
import tensorflow as tf
...
mnist_model = tf.keras.Sequential([
 tf.keras.Input(shape=(28, 28, 1)),
 tf.keras.layers.Conv2D(32,[3, 3],activation='relu'),
 tf.keras.layers.Conv2D(64,[3, 3],activation='relu'),
 tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
 tf.keras.layers.Dropout(0.25),
 tf.keras.layers.Flatten(),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.5),
 tf.keras.layers.Dense(10, activation='softmax')
])
```

FIGURE 6. Python snippet of MNIST neural network.

leaving unused platform containers running and thus allowing accurate accounting of resources consumed by the user.

IV. PERFORMANCE EVALUATION

We verified the effectiveness of CWL-PLAS by measuring the makespan reduction it provides in a couple of machine learning workflows, named: *MNIST* and *LIVECell*. The MNIST workflow shows how CWL-PLAS reduces the time required to complete a single training task when implemented as a platformed-task. The LIVECell workfolw shows how CWL-PLAS reduces the time required to complete a workflow consisting of platformed and legacy tasks.

The execution of workflows is handled by a CWL-PLASK workflow manager (Fig. 4) that runs on a Kubernetes cluster made up of a control-plane node and 4 worker nodes. These nodes are virtual machines, and those that implement the 4 worker nodes have an NVIDIA Tesla M10 GPU.

For comparison purposes, we implemented each workflow using both legacy CWL and CWL-PLAS. For legacy CWL workflows, each task executes a Python script and runs in an official TensorFlow Docker Container. Instead, CWL-PLAS workflows include platformed-tasks, whose Executor runs a Python script and uses a Horovod sidecar platform [15].

Horovod is a deep learning framework to distribute the training process across multiple workers using different GPUs/CPUs. Each worker trains the same neural network, but with different training and validation sets. After each training batch, the workers share the computed gradients, average

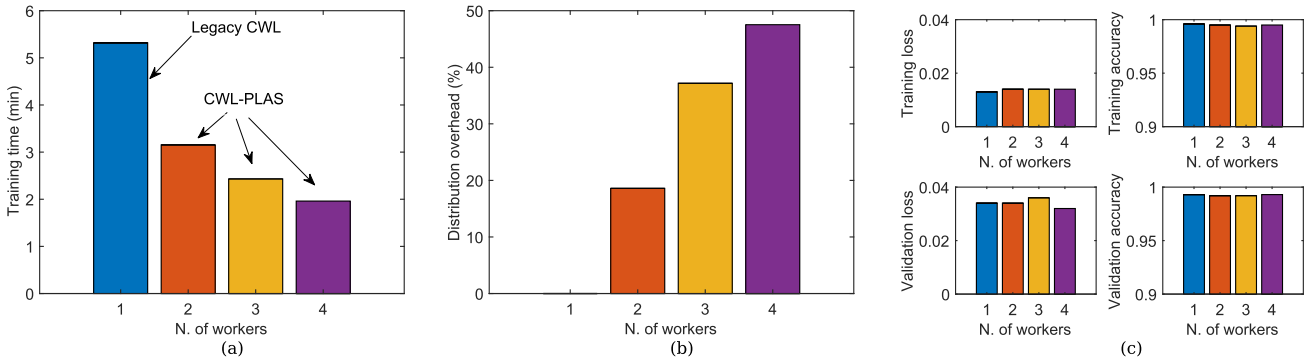


FIGURE 7. Training performance of MNIST workflow in case of legacy CWL (N. of workers=1) and CWL-PLAS workflows with Horovod sidecar platform with 2,3 and 4 parallel workers. Fig. 7a shows the training time, i.e., the time needed to complete the workflow. Fig. 7b shows the learning distribution overhead in Equation 1. Fig. 7c shows the loss and accuracy of the training and validation set computed during the training task.

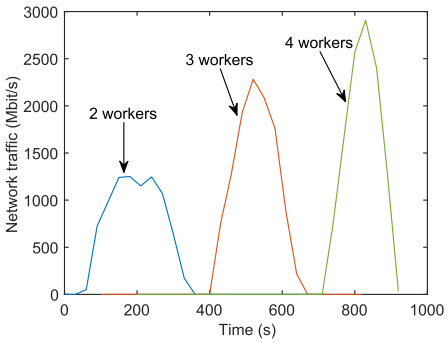


FIGURE 8. Network traffic exchanged by workers during three training sessions of MNIST neural network.

them, and update the weights of the neural network consistently. The number of batches per epoch is scaled by a factor equal to the number of workers. In this way, the duration of an epoch and consequently the duration of training is reduced, although the overall number of batches used per epoch by the entire set of workers remains constant and independent of the number of workers. A distinctive feature of Horovod is the use of a technique called Ring-All-Reduce, which greatly reduces the amount of network communication to share gradients among workers.

A. MNIST WORKFLOW

1) DESCRIPTION

The MNIST workflow is made up of a single task and trains a neural network to classify handwritten digits (0-9) belonging to the popular public MNIST image dataset, a collection of handwritten digits [53]. Fig. 5 shows the raw data in the MNIST dataset. We used the Keras modules of TensorFlow 2.0 to implement the neural network and a Horovod sidecar platform to distribute the computation on different nodes. The Python code in Fig. 6 shows the neural network model consisting of 2 convolutional layers, a `MaxPool` layer, and two final `Dense` layers with intermediate `Dropouts`. The activation function of the intermediate layers is `relu`, while the final activation is `softmax`. We used the `Adam` stochastic gradient descent algorithm and the `SparseCategoricalCrossentropy` as the loss function. The batch size is 128 images, the total number of batches per epoch is 500, and the training lasts 24 epochs.

2) RESULTS

Fig. 7 shows the training metrics of the MNIST workflow versus the number of Horovod workers. For the single-worker case, we used a legacy CWL workflow in which a single container is used to train the neural network. For the cases of 2, 3 and 4 workers, we used CWL-PLAS workflows supported by a Horovod sidecar platform.

Fig. 7a shows the time reduction provided by CWL-PLAS. The legacy CWL workflow took about 5.3 minutes to train the network. Using CWL-PLAS, the training period decreases as the number of parallel workers involved increases. Fig. 7c shows that the loss and accuracy obtained for the training and validation set are practically the same in the different cases, demonstrating the effectiveness of the distributed computation.

One result that may be puzzling is the sub-linear reduction in training time as the number of workers increases, a sign of an inefficiency of the distributed computation. Ideally, we would expect the training time with N workers to be N times smaller than that with only one worker. But we are quite far from this ideal behavior. For instance, the training time with four workers is only 2.7 times shorter than with one single worker. Accordingly, in Fig. 7b we measured the learning "distribution overhead" that is, the increase in the actual training duration compared to the ideal case. In formula:

$$D_O(N) = \frac{T_T(N)}{T_T(1)/N} - 1 \tag{1}$$

where $D_O(N)$ and $T_T(N)$ are the learning distribution overhead and the actual training time for N workers, respectively.

We note that the overhead increases with the number of workers. This inefficiency is mainly due to the fact that, after each training batch, workers share gradients with each other before starting the next batch. This periodic exchange of network traffic creates interruptions in the training process

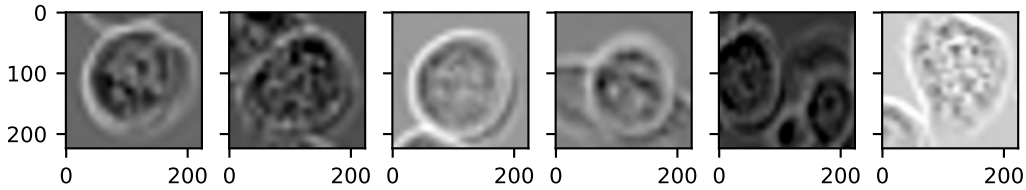


FIGURE 9. Some examples of LIVECell BT-474 cells acquired at hour 0, first three pictures, and after 4 hours, last tree pictures.

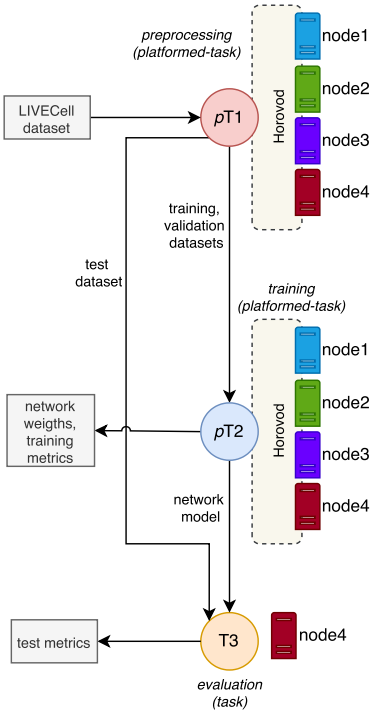


FIGURE 10. LIVECell workflow shows the integration of both legacy CWL and PLAS-CWL tasks. *p*T1 and *p*T2 leverage the parallel distributed computation of CWL-PLAS, while T3 is a legacy CWL task.

```
import tensorflow as tf
...
model0 = tf.keras.applications.DenseNet201(
 include_top=True,weights='imagenet',pooling=True)
model0.trainable=False

input_layer = model0.layers[0].input
output_layer = model0.layers[-2].output

ki = tf.keras.initializers.GlorotUniform(seed=73)
br = tf.keras.regularizers.L1(l1=1e-6)
new_out = tf.keras.layers.Dense(
 2,
 activation="softmax",
 kernel_initializer=ki,
 kernel_regularizer=br,
 bias_regularizer=br
) (output_layer)

LIVECell_model=tf.keras.models.Model(
 inputs = input_layer,
 outputs = new_out)

```

FIGURE 11. Python snippet of LIVECell neural network.

that slow it down. The greater the amount of network traffic, the longer the interruptions and the greater the inefficiency.

Fig. 8 shows the network traffic exchanged among workers during the MNIST neural network training. We first trained by using 2 workers, then three workers, and finally 4 workers. The plot is the result of a moving average operation with a 2-minute window. It shows a high level of network involvement, on the order of Gbit/s, which increases as the number of workers increases, and thus the learning distribution overhead also increases accordingly (Fig. 7b).

Obviously, the overhead would decrease if we improved the capacity of the network. In our case, we measured a throughput of 10 Gbit/s between workers; but even with such "good" network throughput, we had non-negligible inefficiency (e.g., 47% overhead) that is indicative of the need to have a very high-speed network to take full advantage of distributed training. Furthermore, in a virtualized environment such as Kubernetes, not only the speed of the network cards plays a determining role, but also the technology used to virtualize the network, which should limit packet processing as much as possible. For example, in our testbed we used the Kubernetes Calico network plugin configured so as not to use VXLAN tunnels. In an initial configuration in which we used VXLAN tunnels, the throughput among workers dropped by a surprising factor of 10, creating an overhead so high that distributed training sometimes looked worse than non-distributed one.

B. LIVECell WORKFLOW

1) DESCRIPTION

The LIVECell workflow aims to train a neural network to extract meaningful information from cell morphology during a biological experiment in a low-cost and non-destructive way [54], [55]. The case study is chosen from the recently published LIVECell dataset [56]. In particular, we selected a subset of images related to a breast cancer cell line, BT-474. These cells are well known in the literature to usually grow in rafts. The task of the neural network is to uncover morphological changes, including cell volume growth, that occur in four hours. For this purpose, we organized the dataset as consisting of two classes and used the neural network for binary classification. Samples from the first class include cells at time zero, whereas samples from the second class

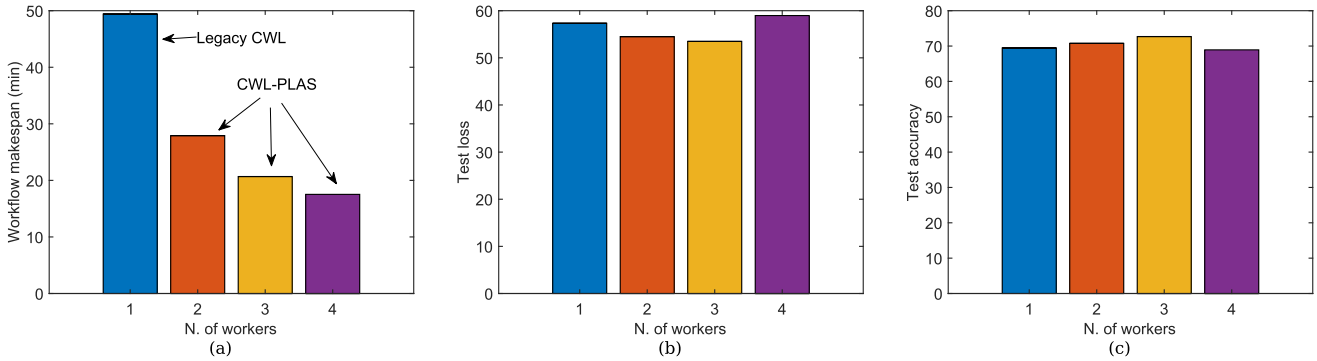


FIGURE 12. Performance of LIVECell workflow in case of legacy CWL (N. of workers=1) and CWL-PLAS workflows with Horovod sidecar platform with 2,3 and 4 parallel workers. Fig. 12a shows the time needed to complete the workflow. Fig. 12b and Fig. 12c show the model loss and accuracy measured on the test set by the last task of the workflow.

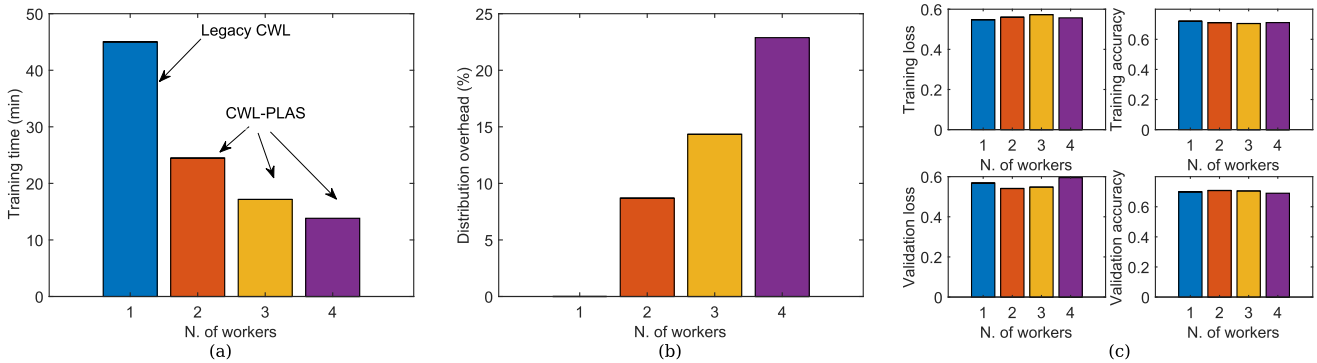


FIGURE 13. Performance of LIVECell training task (pT2) in case of legacy CWL (N. of workers=1) and CWL-PLAS workflows with Horovod sidecar platform with 2,3 and 4 parallel workers. Fig. 13a shows the training time, i.e., the time needed to complete the task. Fig. 13b shows the learning distribution overhead in Equation 1. Fig. 13c shows the loss and accuracy of the training and validation set computed during the training task.

contain cells after four hours. If the network can distinguish between these two classes of cells, it means that it can detect morphological changes. The dataset comprises 8640 and 8022 images for the first and second class, respectively. 70% of the dataset has been used as training set, 20% as validation set, and 10% as test set.

Fig. 10 describes the LIVECell workflow. It consists of three tasks: two CWL-PLAS platformed-tasks (pT1, pT2) and a legacy CWL task (T3). The first two platformed-tasks, pT1 and pT2, leverage the parallel computing capabilities enabled by CWL-PLAS to perform the data preprocessing and training of the neural network model based on the LIVECell dataset, while the third task, T3, is performed by a legacy CWL and evaluates the trained model.

The data preprocessing task takes as input the raw LIVECell images and creates dataset files used to train and validate the model in the second task, and to evaluate it in the final task. To implement this task as a platformed one, we parallelized the data preprocessing operation in many processes so that each process analyzes a subset of the raw data. Instead of developing an ad hoc sidecar platform, we reused a Horovod one, since it already contains the libraries needed for software parallelization, and we embedded the data preprocessing processes in the different Horovod workers running on separate nodes.

The training task is carried out by the neural network model shown in Fig. 11 whose computation is distributed over the workers of a Horovod sidecar platform. We used a transfer-learning approach based on Densenet 201 [57] that is a pre-trained Densely Connected Convolutional Neural Network, already tested over various datasets of cell images [58]. This dense network structure allows the layers of the network to be grouped into blocks (Dense blocks). Within the single block, each layer receives additional inputs from all the preceding layers, thereby allowing to maintain low and high abstraction features even in deeper layers. To adapt the network to our specific binary classification task, we replaced the last fully connected layer with a new classification `Dense` layer composed of two neurons (one for each class). These neurons use the `softmax` activation function, `L1` regularization penalty for the weights of bias and kernel neurons, and the `GlorotUniform` initializer. We used the `Adam` stochastic gradient descent algorithm and `CategoricalCrossentropy` as loss function. We trained only the last layer. The batch size is 32 images, the total number of batches per epoch spans the entire training set, and the training lasts for 10 epochs.

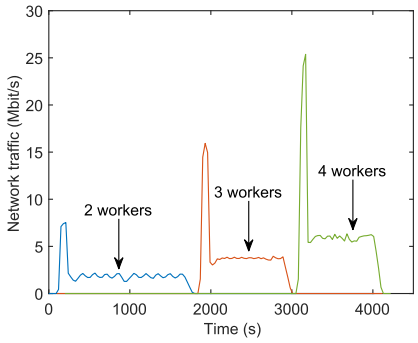


FIGURE 14. Network traffic exchanged by workers during three training sessions of LIVECell neural network.

Finally, the evaluation task takes as input the test dataset from pT1 and the neural network model computed from pT2 and evaluates the performance of the neural network. This software runs in a single container and it is a legacy CWL task.

2) RESULTS

Fig. 12 shows the performance of the workflow, while Fig. 13 focuses on the performance of the training task. As previously done, we used legacy CWL tasks for the case of only one worker, while CWL-PLAS platformed-tasks with Horovod sidecar platform for the cases of 2, 3, and 4 workers.

Fig. 12a shows how the use of CWL-PLAS makes the makespan shorter as the number of workers used by pT1 and pT2 increases. Again, from Fig. 13a, we observe a significant reduction in training time due to the parallel computing capabilities enabled by CWL-PLAS for pT2. For example, the training time with legacy CWL is 45 minutes, compared to the 13 minutes required by CWL-PLAS with 4 workers.

Unlike the MNIST workflow, the learning distribution overhead shown in Fig. 13b is significantly lower for two reasons. First, the neural network is much more complex, so the computation time of training batches is much longer; consequently, the network is involved less frequently. Second, the amount of data shared at the end of a training batch is smaller because workers share only the gradients of the last layer, which is the only one trained. As shown in Fig. 14, this less frequent exchange of gradients with fewer bytes results in lower network traffic, on the order of tens of Mbit/s compared to the Gbit/s of MNIST. This lower network traffic consequently reduces the learning distribution overhead.

We notice intriguing spikes in traffic at the beginning of a training process. These are due to the initial exchange of the entire set of network weights from worker No. 0 to the other workers to synchronize their neural networks. After this initial phase, only the gradients of the last layer are shared for each batch, and the traffic decreases. These spikes are not evident in MNIST cases because the entire set of gradients is exchanged for each batch, making traffic high at each stage of the training process.

From Fig. 13c we note that the training/validation accuracy obtained is approximately 70 percent, with an even distribution of errors among the classes. This result also holds for the test dataset, as shown in Fig. 12b. The main morphological differences, detectable by the network, among the classes are related to the progression of the cell cycle in four hours. Cell populations show wide distributions in cell cycle status, and this heterogeneity acts as a confounding factor motivating the classification performance achieved. However, achieving binary classification accuracy greater than 50 percent highlights how a properly trained deep neural network may be able to identify variations in cell shape related to the cell cycle. Such variations may represent strategic information for understanding many biological processes, such as the evaluation of therapeutic treatment.

V. CONCLUSION

In this paper, we briefly presented the Common Workflow Language (CWL), a description language for workflows composed of tasks that allows portability and reproducibility, in part due to the adoption of Linux container technology for the packaging of task software.

Being ''confined'' in a container, a task can exploit only the resources of the host where the container is running. This paper proposes CWL-PLAS, an extension of CWL that allows a single task to exploit the resources of multiple nodes in a cluster. We have named this type of task as platformed-task, because its execution is supported by a parallel computing platform instantiated along with the task and specialized for the task activity. Technically, a platformed-task is a legacy CWL task deployed together with a sidecar platform, e.g., for distributed machine-learning, data analysis, etc.

CWL-PLAS allows users to request sidecar platforms that are packaged as Helm chart. However, the concept of platformed-task is more generic. For instance, the CWL specification can be further extended to support platformed-task based on other package managers such as Docker Compose [50].

We expect that in most cases the user will use the publicly available Helm charts to deploy the sidecar platforms needed for his tasks. Consequently, the user only needs to prepare the `.cwl` file of the platformed-task, develop the software that executes the task, and build the Docker image that contains it. This process is similar to that performed by the user in the case of legacy CWL. However, the implementation of the task software might be easier in the case of CWL-PLAS because the user can take advantage of the high-level programming libraries provided by the sidecar platforms. Therefore, CWL-PLAS not only provides performance improvement by leveraging parallel computing frameworks, but also has the potential to simplify the development of task software.

We have implemented a workflow manager that runs CWL-PLAS workflows, called CWL-PLASK. It is based on Kubernetes and is open-source [16].

REFERENCES

[1] *Existing Workflow Systems*. Accessed: Mar. 21, 2023. [Online]. Available: https://s.apache.org/existing-workflow-systems

[2] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Gener. Comput. Syst.*, vol. 25, no. 5, pp. 528–540, May 2009.

[3] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *J. Grid Comput.*, vol. 13, no. 4, pp. 457–493, Dec. 2015.

[4] J. Liu, S. Lu, and D. Che, "A survey of modern scientific workflow scheduling algorithms and systems in the era of big data," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Nov. 2020, pp. 132–141.

[5] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–36, May 2019.

[6] K. A. Ocaña, D. D. Oliveira, F. Horta, J. Dias, E. Ogasawara, and M. Mattoso, "Exploring molecular evolution reconstruction using a parallel cloud based scientific workflow," in *Proc. Brazilian Symp. Bioinf.* Cham, Switzerland: Springer, 2012, pp. 179–191.

[7] J. C. Jacob, "Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 2, pp. 73–87, 2009.

[8] A. T. Kouanou, D. Tchiotsop, R. Kengne, D. T. Zephirin, N. M. A. Armele, and R. Tchinda, "An optimal big data workflow for biomedical image analysis," *Informat. Med. Unlocked*, vol. 11, pp. 68–74, Jan. 2018.

[9] *Common Workflow Language*. Accessed: Mar. 21, 2023. [Online]. Available: https://www.commonwl.org/

[10] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanìc, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and T. C. Community, "Methods included: Standardizing computational reuse and portability with the common workflow language," *Commun. ACM*, vol. 65, no. 6, pp. 54–63, Jun. 2022.

[11] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[12] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS ONE*, vol. 12, no. 5, May 2017, Art. no. e0177459.

[13] *Kubernetes: Production-Grade Container Orchestration*. Accessed: Mar. 21, 2023. [Online]. Available: https://kubernetes.io/

[14] *Apache Spark*. Accessed: Mar. 21, 2023. [Online]. Available: https://spark.apache.org/

[15] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.

[16] *CWL-PLAS*. Accessed: Mar. 21, 2023. [Online]. Available: https://github.com/PlatformedTasks

[17] I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, "StreamFlow: Cross-breeding cloud with HPC," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 4, pp. 1723–1737, Oct. 2021.

[18] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. F. da Silva, G. Papadimitriou, and M. Livny, "The evolution of the pegasus workflow management software," *Comput. Sci. Eng.*, vol. 21, no. 4, pp. 22–36, Jul. 2019.

[19] V. Jalili, E. Afgan, Q. Gu, D. Clements, D. Blankenberg, J. Goecks, J. Taylor, and A. Nekrutenko, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update," *Nucleic Acids Res.*, vol. 48, no. W1, pp. W395–W402, Jul. 2020.

[20] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Proc. 16th Int. Conf. Sci. Stat. Database Manage.*, Jun. 2004, pp. 423–424.

[21] S. P. Huber, "AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance," *Sci. Data*, vol. 7, no. 1, pp. 1–18, Sep. 2020.

[22] E. N. Schettino. (Jun. 2021). *Pydoit/Doit: Task Management & Automation Tool (Python)*. Accessed: Mar. 21, 2023, doi: 10.5281/zenodo.4892136. [Online]. Available: https://zenodo.org/record/4892136#.ZFEDGh9ByM8

[23] S. Lampa, M. Dahlö, J. Alvarsson, and O. Spjuth, "SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines," *GigaScience*, vol. 8, no. 5, May 2019, Art. no. giz044.

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

[25] J. Tronge, P. Grubel, T. Randles, Q. Wofford, R. Davis, S. Anaya, and Q. Guan, "BEE orchestrator: Running complex scientific workflows on multiple systems," in *Proc. IEEE 28th Int. Conf. High Perform. Comput., Data, Anal. (HiPC)*, Dec. 2021, pp. 376–381.

[26] *Arvados Unified Data and Workflow Management*. Accessed: Mar. 21, 2023. [Online]. Available: https://arvados.org/

[27] J. Vivian, "Toil enables reproducible, open source, big biomedical data analyses," *Nature Biotechnol.*, vol. 35, no. 4, pp. 314–316, Apr. 2017.

[28] M. Kotliar, A. V. Kartashov, and A. Barski, "CWL-airflow: A lightweight pipeline manager supporting common workflow language," *GigaScience*, vol. 8, no. 7, Jul. 2019, Art. no. giz084.

[29] *Calrissian*. Accessed: Mar. 21, 2023. [Online]. Available: https://github.com/Duke-GCB/calrissian

[30] *Cromwell*. Accessed: Mar. 21, 2023. [Online]. Available: https://cromwell.readthedocs.io/en/stable/

[31] Y. Gil, V. Ratnakar, J. Kim, P. Gonzalez-Calero, P. Groth, J. Moody, and E. Deelman, "Wings: Intelligent workflow-based design of computational experiments," *IEEE Intell. Syst.*, vol. 26, no. 1, pp. 62–72, Jan. 2011.

[32] Y. Gil, "Mapping semantic workflows to alternative workflow execution engines," in *Proc. IEEE 7th Int. Conf. Semantic Comput.*, Sep. 2013, pp. 377–382.

[33] A. E. Ahmed, J. M. Allen, T. Bhat, P. Burra, C. E. Fliege, S. N. Hart, J. R. Heldenbrand, M. E. Hudson, D. D. Istanto, M. T. Kalmbach, G. D. Kapraun, K. I. Kendig, M. C. Kendzior, E. W. Klee, N. Mattson, C. A. Ross, S. M. Sharif, R. Venkatakrishnan, F. M. Fadlelmola, and L. S. Mainzer, "Design considerations for workflow management systems use in production genomics research and the clinic," *Sci. Rep.*, vol. 11, no. 1, pp. 1–18, Nov. 2021.

[34] I. Colonnelli, B. Cantalupo, R. Esposito, M. Pennisi, C. Spampinato, and M. Aldinucci, "HPC application cloudification: The streamflow toolkit," in *Proc. 12th Workshop Parallel Program. Run-Time Manage. Techn. Many-Core Archit. 10th Workshop Design Tools Archit. Multicore Embedded Comput. Platforms (PARMA-DITAM)*, 2021, pp. 65–78.

[35] *GÉANT Cloud Flow (GCF) Platform*. Accessed: Mar. 21, 2021. [Online]. Available: https://clouds.geant.org/community-cloud/

[36] *CWL Workflow Execution Service*. Accessed: Mar. 21, 2021. [Online]. Available: https://github.com/elixir-cloud-aai/cwl-WES

[37] D. Talia, "Workflow systems for science: Concepts and tools," *Int. Scholarly Res. Notices*, vol. 2013, Jan. 2013, Art. no. 404525.

[38] M. Hosseinzadeh, M. Y. Ghafour, H. K. Hama, B. Vo, and A. Khoshnevis, "Multi-objective task and workflow scheduling approaches in cloud computing: A comprehensive review," *J. Grid Comput.*, vol. 18, no. 3, pp. 327–356, Sep. 2020.

[39] M. A. Rodriguez and R. Buyya, "Budget-driven scheduling of scientific workflows in IaaS clouds with fine-grained billing periods," *ACM Trans. Auto. Adapt. Syst.*, vol. 12, no. 2, pp. 1–22, Jun. 2017.

[40] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 158–169, Jan. 2013.

[41] C. Bai, S. Lu, I. Ahmed, D. Che, and A. Mohan, "LPOD: A local path based optimized scheduling algorithm for deadline-constrained big data workflows in the cloud," in *Proc. IEEE Int. Congr. Big Data (BigDataCongress)*, Jul. 2019, pp. 35–44.

[42] M. Naghibzadeh, "Modeling workflow of tasks and task interaction graphs to schedule on the cloud," in *Proc. CLOUD Comput.*, Mar. 2016, p. 81.

[43] *GA4GH CWL Task Execution*. Accessed: Mar. 21, 2023. [Online]. Available: https://github.com/ohsu-comp-bio/cwl-tes

[44] D. Bernstein, "Containers and cloud: From LXC to Docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[45] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[46] *Helm*. Accessed: Mar. 21, 2023. [Online]. Available: https://helm.sh/

[47] F. Nielsen, *Introduction to HPC With MPI for Data Science*. Cham, Switzerland: Springer, 2016.

[48] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–29, Jan. 2019.

[49] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms," in *Proc. IEEE/ACM Int. Workshop Containers New Orchestration Paradigms Isolated Environ. HPC (CANOPIE-HPC)*, Nov. 2019, pp. 11–20.
[50] *Docker Compose*. Accessed: Mar. 21, 2023. [Online]. Available: https://docs.docker.com/compose/
[51] *TESK*. Accessed: Mar. 21, 2023. [Online]. Available: https://github.com/elixir-cloud-aai/TESK
[52] *Task Execution Service (TES) API*. Accessed: Mar. 21, 2023. [Online]. Available: https://github.com/ga4gh/task-execution-schemas
[53] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.
[54] M. D'Orazio, F. Corsi, A. Mencattini, D. Di Giuseppe, M. Colomba Comes, P. Casti, J. Filippi, C. D. Natale, L. Ghibelli, and E. Martinelli, "Deciphering cancer cell behavior from motility and shape features: Peer prediction and dynamic selection to support cancer diagnosis and therapy," *Frontiers Oncol.*, vol. 10, Oct. 2020, Art. no. 580698.
[55] A. Mencattini, A. Spalloni, P. Casti, M. C. Comes, D. Di Giuseppe, G. Antonelli, M. D'Orazio, J. Filippi, F. Corsi, H. Isambert, C. Di Natale, P. Longone, and E. Martinelli, "NeuriTES. Monitoring neurite changes through transfer entropy and semantic segmentation in bright-field time-lapse microscopy," *Patterns*, vol. 2, no. 6, Jun. 2021, Art. no. 100261.
[56] C. Edlund, T. R. Jackson, N. Khalid, N. Bevan, T. Dale, A. Dengel, S. Ahmed, J. Trygg, and R. Sjögren, "LIVECell—A large-scale dataset for label-free live cell segmentation," *Nature Methods*, vol. 18, no. 9, pp. 1038–1045, Sep. 2021.
[57] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4700–4708.
[58] R. Ma, J. Miao, L. Niu, and P. Zhang, "Transformed ℓ_1 regularization for learning sparse deep neural networks," *Neural Netw.*, vol. 119, pp. 286–298, Nov. 2019.

ANDREA DETTI (Member, IEEE) is currently a Professor of wireless networks and cloud computing with the Department of Electronic Engineering, University of Rome Tor Vergata. He is the coauthor of more than 80 papers in journals and conference proceedings and participated in several EU-funded projects with coordination and research roles. His current research interests include computer networks and cloud computing.

LUDOVICO FUNARI received the master's degree in ICT and internet engineering, in October 2019. He is currently pursuing the Ph.D. degree with the University of Rome Tor Vergata. He has worked on European projects, such as the EU H2020 "Fed4IoT" Project, as an Italian National Inter-University Consortium for Telecommunications (CNIT) Researcher, and on the MIUR Research Project "Liquid_Edge" with the University of Rome Tor Vergata. His research interests include the IoT and cloud and edge computing.

LUCA PETRUCCI received the master's degree in computer science engineering from the University of Rome Tor Vergata, in April 2019. He is currently pursuing the Ph.D. degree with the University of Rome Tor Vergata. From April 2016 to December 2019, he was a Researcher with the Italian National Inter-University Consortium for Telecommunications (CNIT), where he developed his bachelor's thesis and master's thesis, respectively, concerning the EU projects BEBA and 5G-PICTURE.

MICHELE DÓRAZIO is currently a Postdoctoral Researcher with the Department of Electronic Engineering, University of Rome Tor Vergata. He has coauthored more than 14 publications in international journals and conference proceedings. His main research interests include machine learning and deep learning algorithms applied to biomedical applications, with a specific focus on lab-on-chip data analysis.

ARIANNA MENCATTINI is currently an Associate Professor with the Department of Electronic Engineering, University of Rome Tor Vergata. She has coauthored more than 120 papers in international journals and conferences. Her main research interests include the metrological aspects of image and video processing techniques for the development of computed-assisted systems.

EUGENIO MARTINELLI is currently a Full Professor with the Department of Electronic Engineering, University of Rome Tor Vergata, where he is the Head of the Bioinspired Electronic Engineering Group and the Co-Director of the Interdisciplinary Center of Organ-on-Chip and Lab-on-Chip applications (IC-LOC). He was responsible (PI) for several national and international research projects for the development of sensorial systems and data analysis for space, food, and biomedical applications. He has authored more than 240 publications in international journals and congresses (with more than 5000 citations and H-index equal to 40) and holds six patents.

Open Access funding provided by 'Università degli Studi di Roma "Tor Vergata"' within the CRUI CARE Agreement