

# StreamFlow: cross-breeding cloud with HPC

Iacopo Colonnelli<sup>1</sup>, Barbara Cantalupo<sup>1</sup>, Ivan Merelli<sup>2</sup>, and Marco Aldinucci<sup>1</sup>

<sup>1</sup>*Department of Computer Science, University of Torino, Italy*

<sup>2</sup>*Biomedical Technologies (ITB) of the Italian National Research Council (CNR), Italy*

## Abstract

Workflows are among the most commonly used tools in a variety of execution environments. Many of them target a specific environment; few of them make it possible to execute an *entire* workflow in different environments, e.g. Kubernetes and batch clusters. We present a novel approach to workflow execution, called StreamFlow, that complements the workflow graph with the declarative description of potentially complex execution environments, and that makes it possible the execution onto multiple sites not sharing a common data space. StreamFlow is then exemplified on a novel bioinformatics pipeline for single-cell transcriptomic data analysis workflow.

## 1 Introduction

Both in the HPC and cloud realms, workflows play an essential role for applications coordination because they provide means to model and formalise complex processes in multiple steps, e.g. tasks, jobs, OS containers or even Virtual Machines, depending on the target system. Steps are generally arranged in a partial order induced by (true) data dependency. For this, workflows can be naturally represented with direct graphs.

Although workflows are used in different execution environments, such as HPC, cloud and Edge, all of these environments continue their path toward greater specialisation in term of typical features and workloads. While RESTful APIs are becoming the lingua franca to access and compose computation and storage in the cloud, the HPC platforms are bound to batch job schedulers. Starting a web server on an HPC platform is generally not admitted, as it is impractical to access to cloud storage, e.g. to retrieve temporary results. While the execution of *independent* steps in the cloud means they can be executed *in any temporal order in a single processing element*, in the HPC platforms the need for co-allocating *at the same time* multiple processing elements to execute a single job is the rule [5]. This complementarity is the cornerstone of a computing continuum that appears emerging in data-driven applicative domains. We envision this continuum as composed of more and more specialised and therefore heterogeneous environments. For this, also workflows need to embrace heterogeneity, by embedding the capability to execute a single workflow

on multiple different environments. For this to happen, workflows should gain a higher level of abstraction, subsuming the role of coordination language of other lower level and more specialised workflows targeting a specific platform.

In this work, we introduce *StreamFlow*, a novel workflow model that extends a classic workflow system with a declarative description of possibly many environments and with the relations among workflow nodes and execution environments. StreamFlow is not yet another workflow system; it somewhat conceptually aims at complementing a workflow system to raise its level of abstraction, providing the workflow with a “virtual” platform spawning multiple sites. In other words, StreamFlow makes it possible to partition a workflow and describe an execution plan spawning across multiple sites, even if they do not share the same data space. In this, StreamFlow leverages on lower level features such as the deployment of explicitly parallel nodes, e.g. MPI execution, which is targeted via HPC jobs schedulers (supporting OS containers).

The StreamFlow concept is exemplified by way of a proof-of-concept implementation based on the Common Workflow Language (CWL) interface, which is used to specify a novel bioinformatic pipeline (single-cell transcriptomic data analysis). Thanks to StreamFlow, the single-cell pipeline is executed on two sites: a Kubernetes orchestrator on the cloud and an HPC cluster on-premise.

In Sec. 2 we describe related work. Being the literature in workflows massive, we focus on the aspects of interest for this work, inviting the reader to refer to existing surveys for a more general comparison among workflow systems. In Sec. 3, we present the proposed approach, i.e. StreamFlow basic principles, whereas StreamFlow design and implementation are described in Sec. 4. Sec. 5 reports the single-cell transcriptomic data analysis workflow, along with StreamFlow experimentation. Finally, Sec. 6 summarises conclusions and future works.

## 2 Related works

Scientific applications are complex processes that possibly involve a large number of interconnected tasks, process large amounts of data and require high computational power. HPC infrastructures can provide all necessary computational power at the price of some rigidity of their exploitation: they are typically managed as batch and closed systems. Also, cloud environments are currently the reference architecture for executing complex applications, e.g. micro-services, but their effectiveness in term of cost and performance is not always adequate.

Workflows provide powerful abstractions to design scientific applications, also supporting their execution on specific infrastructures. According to this vision, we can consider workflows as an interface between the domain specialists and the computing infrastructure. However, the workflow landscape is very variegated, as it embraces scientific domain tools, mainly focused on resolving typical modelling issues in the domain, and low-level specifications, aimed at executing tasks on multi processes infrastructures.

According to the Workflow Management Coalition glossary[40], a Workflow Management System (WMS) is defined as *a system that creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.*

The WMS area comprises a large number of systems, possibly with very different objectives. Here we are focusing on WMSs that respond to application requirements such as modelling, portability and reproducibility, coupled with performant execution on the most suitable infrastructures, taking into account data management, costs and performance needs.

Several surveys exist on WMSs, mainly focused on comparing different functionalities concerning high-level definitions and available implementations [11, 43, 28]. Some of them address workflow systems that have been extensively used in scientific communities, providing evidence for their main characteristics and how they evolved in ten years range of time [9]. Others are more oriented to provide characterisation and classification of workflow management systems, in order to depict the main features needed to support extreme-scale applications [15].

We are not providing here a comprehensive survey of existing workflow management solutions. Instead, we are more interested in understanding the most critical needs, the most effective approaches and the most promising evolutions in this continuously changing technological context. We think that two main levels of analysis must be considered: the application level, where the orchestration of the different functional components of the application is managed, and the infrastructure level, where the computational units composing the workflow are executed (workflow engine). At the first level, it is important to evaluate the ability of the system to respond to user needs. Scientific workflows are user-driven systems, specifically developed to satisfy domain requirements. Also, several workflow specifications are now focusing on managing massive amounts of data needed and computed by all the applications. At the infrastructure level, together with established architectures like cluster or grid, clouds are now the most referred infrastructure for application execution. Moreover, HPC facilities are now getting more and more importance outside the research centre, and new paradigms are gaining attention like containers and orchestrators. Starting from the first level, we are interested in evaluating how the main WMSs are responding to user needs and how they can handle new architecture adoption.

## 2.1 Scientific Workflows

Scientific workflows are widely recognised as a useful paradigm to describe, manage, and share complex scientific analyses[1]. Experiments are modelled using high-level declarative language that can be expressed using advanced graphical interfaces, suitable for researchers with little programming experience, or described programmatically. The objective in scientific workflow management is supporting researchers in specifying the experiment, also ensuring reproducibility and scalability.

Many scientific WMSs emerged with the diffusion of the Web Service and Grid technologies, which offered the possibility to access robust services and infrastructures in a more natural way than before[10]. Therefore, they were mainly targeted towards these architectures and not focused on portability. Nevertheless, by evolving in strict contact with the scientific community, they acquired maturity from the functional design point of view and established consensus among researchers. Moreover, some of them currently provide workflows repositories or are evolving to support diverse newer architectures.

Kepler<sup>1</sup>[29], Askalon<sup>2</sup>[21], Taverna<sup>3</sup>[34], Galaxy<sup>4</sup>[2] are widely used in the scientific communities. Workflows are designed through graphical interfaces and executed on a service-based architecture deployed in grids or clouds. Pegasus<sup>5</sup>[17] takes in input an abstract workflow modelled as a DAG (described in an XML format (DAX)) and is able to execute it on a wide range of environments, running either on a single system or across a heterogeneous set of resources like remote clusters (e.g. Open Science Grid), and clouds (Amazon EC2, Google Cloud), relying on HTCondor[38] DAGMan as the workflow execution engine. The PegasusLite[18] remote workflow engine can set up an application container if required for a users task, allowing support for containerised applications both with Docker[31] and Singularity[27]. The Pegasus MPI Cluster (PMC)[35] relies instead on MPI and the master-worker paradigm to execute large, fine-grained workflows. KNIME is a graphical workbench to create workflows, allowing to specify either a local or a distributed execution for each node. The KNIME Cluster Executor is compatible with Grid Engine derivatives.

There are also efforts in defining workflow specification languages or standards. For instance, OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)<sup>6</sup> offers a structured (XML based) language that encodes an application as a Service Template” and defines the different components and relations between them using an application topology [36]. It thus defines service templates that contain a cloud services topology. The Common Workflow Language (CWL)<sup>7</sup> [7] is an open standard for describing analysis workflows following a JSON or YAML syntax or a mixture of the two. CWL is designed to meet the needs of data-intensive science, such as Bioinformatics, Medical Imaging, Astronomy, High Energy Physics and Machine Learning.

## 2.2 Dataflow Approach

Considering how Big Data are spreading in almost every scientific field, interest in dataflow management is growing, and many workflow languages, libraries and system are addressing the problem of efficiently perform massive data computation.

An interesting dataflow language for scientific computing is Swift<sup>8</sup>[41]. It has implicitly parallel dataflow semantics, in which all statements are eligible to run concurrently, limited only by the data flow. It is typically used to express scientific workflows, controlling the execution of relatively large tasks. Swift/K implementation focuses on distributed execution of tasks on diverse computing resources, including clouds and clusters, while Swift/T focuses on high-performance computation on clusters and supercomputers, by translating Swift scripts into MPI programs[20].

A different approach consists in the definition of parallel libraries, which are then included in workflows. For instance, Dask<sup>9</sup>[16], a library for distributed

---

<sup>1</sup><https://kepler-project.org/>

<sup>2</sup><http://www.askalon.org/>

<sup>3</sup><https://taverna.incubator.apache.org/>

<sup>4</sup><https://galaxyproject.org/learn/advanced-workflow/>

<sup>5</sup><https://pegasus.isi.edu/>

<sup>6</sup><https://www.oasis-open.org/>

<sup>7</sup><https://www.commonwl.org>

<sup>8</sup><http://swift-lang.org/main/>

<sup>9</sup><https://docs.dask.org/>

computing in Python, allows defining data collections like parallel arrays, data frames and lists, which extend common interfaces like NumPy, Pandas, or Python iterators to distributed environments. These parallel collections run on top of dynamic task schedulers that can run locally or in a distributed fashion across a cluster.

Luigi<sup>10</sup> is a Python package for building complex pipelines of batch jobs, developed by Spotify. Conceptually, it is similar to GNU Make where tasks, in turn, may have dependencies on other tasks. The dependency graph is specified within Python, and this makes it easy to build up complex dependency graphs of tasks, where the dependencies can involve date algebra or recursive references to other versions of the same task. Pipelines can be executed as Hadoop jobs, but Luigi can also be used to create workflows with any external job written in R, Scala or Spark.

Apache Airflow<sup>11</sup> allows authoring workflows as Directed Acyclic Graphs (DAGs) of tasks and the scheduler executes the tasks on an array of workers while following the specified dependencies. Airflow has a modular architecture and allows defining operators that determine what actually executes when a DAG is running. Operators are already available for different cloud providers and also for Kubernetes. CWL-Airflow is one of the first pipeline managers supporting version 1.0 of the CWL standard [25]

Makeflow<sup>12</sup>[3] is a workflow engine for data-intensive scientific applications that can execute applications on a variety of distributed execution systems including campus clusters, clouds, and grids. The end-user expresses a workflow using a syntax similar to Make in a technology-neutral way. Then the workflow can be deployed to a variety of different systems without modification, including local execution on a single multicore machine, public cloud services such as Amazon EC2 and Amazon Lambda, batch systems like HTCondor, PBS [23], SLURM [42]. Makeflow can be run on Kubernetes cluster and can interoperate with a variety of container technologies, including Docker, Singularity, and Umbrella.

Nextflow<sup>13</sup>[19] is a bioinformatics framework based on the dataflow programming model and based on the UNIX pipe concept. Parallelisation is implicitly defined by the processes input and output declarations. It provides out of the box executors for SGE, LSF, SLURM, PBS and HTCondor batch schedulers and for Kubernetes, Amazon AWS and Google Cloud platforms. It also provides support for dependencies management through built-in support for Conda, Docker, Singularity, and Modules.

### 2.3 Container Orchestration

Containers are gaining popularity also in the scientific domain because they simplify software installation for the end-user and offer isolation between processes.

Pachyderm<sup>14</sup>[33] is a large-scale data processing tool built natively on top of Kubernetes. Users create workflows simply supplying a JSON pipeline specification including a Docker image, an entry point command to execute inside the

---

<sup>10</sup><https://github.com/spotify/luigi/>

<sup>11</sup><https://airflow.apache.org/>

<sup>12</sup><http://ccl.cse.nd.edu/software/makeflow/>

<sup>13</sup><https://www.nextflow.io/>

<sup>14</sup><http://pachyderm.io/>

container and one or more data input(s). Pachyderm ensures that the corresponding pods are created in Kubernetes, distributes the input data across them and collects the corresponding outputs. Argo<sup>15</sup> is an open-source project that provides container-native workflows for Kubernetes, implementing each step in a workflow as a container. Argo enables users to launch multi-step pipelines using a custom DSL, which is similar to the traditional YAML syntax. A Galaxy installation developed in the PhenoMeNal project <sup>16</sup>[32] allows users to access all of the project containerised tools through a workflow environment, on a scalable infrastructure that can be deployed to public and private cloud installations.

Finally, many other systems adopt their own specific approach. Snake-make<sup>17</sup>[24] workflows are essentially Python scripts extended by declarative code that can be executed on distributed infrastructures, such as clusters, grids and clouds. COMP Superscalar (COMPSs)[30] framework is mainly composed of a programming model, which exploits the inherent parallelism of applications at execution time for distributed infrastructures, such as clusters, grids and clouds, and a runtime system. HyperLoom[14] is an open-source platform for defining and executing pipelines in distributed environments and providing a Python interface for defining tasks. HyperLoom is a self-contained system: it does not rely on any external scheduler for the actual execution of the tasks.

## 3 Methods

### 3.1 Multi-container environments

Portability and reproducibility have always been two fundamental aspects of scientific workflows. Nevertheless, the combination of the two is undoubtedly a non-trivial requirement to satisfy, since it is necessary to guarantee that a piece of code running on top of potentially very diverse execution environments will give identical results. The first obvious issue here comes from the need to provide the same versions of all the libraries directly or indirectly involved in the computation. On top of that, some numerical stability problems can arise when running the same code on different platforms, e.g. on Linux and Mac OS X [19]. Fortunately, with the diffusion of lightweight containerisation technologies like Docker[31] and Singularity[27], a straightforward solution for these issues finally appeared and nowadays container-based tasks are supported by a wide number of WMSs on the market, either as an alternative to native execution or as first-class citizens [26].

Nevertheless, the typical way to support containerisation in WMSs is through a one-to-one mapping between tasks and containers, i.e. a container image is associated with each task in the workflow graph. In this setting, the execution flow of a single task always consists of three sequential steps: the container is launched, the task is executed inside it, and finally the container is stopped. Drawing a parallel with the famous Flynn's taxonomy [22], we could define this execution pattern as *Single-Task Single-Container* (STSC).

When compared with a *Multiple-Tasks Single-Container* (MTSC) alternative, the STSC pattern comes with a decisive advantage: since containers' file-

---

<sup>15</sup><https://argoproj.github.io/argo/>

<sup>16</sup><http://phenomenal-h2020.eu/home/>

<sup>17</sup><https://snakemake.readthedocs.io/en/stable/>

system is commonly ephemeral, every task execution runs inside a clean and consistent environment (with the obvious exception of eventual temporary files saved into persistent folders). For its part, an MTSC execution can provide some performance improvements in those cases when either the task execution is very fast (comparable with the startup and shutdown overheads of a container, generally in the order of milliseconds). Moreover, MTSC can be useful also when a process inside the container must complete a heavy initialisation phase before being ready to perform tasks.

Far more interesting would be the *Single-Task Multiple-Containers* (STMC) setting, because it allows using multiple, possibly heterogeneous environments to solve a single task. For example, with an STMC approach, it would be possible to run an MPI task on top of multiple nodes or a MapReduce-based task with multiple instances of Apache Spark. Finally, the most general *Multiple-Tasks Multiple-Containers* (MTMC) setting would also allow for *concurrent* task execution, i.e. a configuration in which tasks  $T_1$  and  $T_2$  execute at the same time on different resources and  $T_1$  produces data consumed by  $T_2$ . The support for this last configuration becomes fundamental when dealing with stream-based workflows [15]. In principle, also an MTSC configuration enables the concurrent execution of tasks into the same resource, but here the advantage is less valuable. Indeed, it is far easier to obtain the same behaviour in an STSC setting with a single task charged with launching and managing all the required processes.

Unfortunately, a simple many-to-many task-image association is not enough to model a \*MC configuration, because it is also necessary to explicitly specify the connections among different containers. Nevertheless, some ways to define multi-container environments are already present on the market, from simple libraries like Docker Compose<sup>18</sup> and Singularity Compose<sup>19</sup> to complex orchestrators as Kubernetes<sup>20</sup> or Docker Swarm<sup>21</sup>. Therefore, it is a wise choice to rely on them for the environment definition. This can be achieved by substituting the original one-to-one task-container association with a many-to-one task-environment association and by treating an entire multi-container environment as the unit of deployment. It is worth noting that even a many-to-many association would be potentially feasible, allowing a single task to be split among different environments. Nevertheless, this would overcomplicate both the scheduling policies and the communication layer, forcing the need to distinguish between inter-environment and intra-environment interactions among different resources executing the same task.

All these considerations can be summarised by the following two requirements:

*R1* A uniquely identified multi-container environment definition must be treated as an atomic deployment unit. A unit must be deployed before starting to execute the first associated task and undeployed after the execution of the last associated task.

*R2* Each task can be associated with a single deployment unit, but the same deployment unit can be associated with multiple tasks.

---

<sup>18</sup><https://docs.docker.com/compose/>

<sup>19</sup><https://singularityhub.github.io/singularity-compose/>

<sup>20</sup><https://kubernetes.io/>

<sup>21</sup><https://docs.docker.com/engine/swarm/>

## 3.2 Hybrid workflows

When considering data-intensive scientific workflows, all those aspects related to data management (as data locality, data access, data transfers, and so on and so forth) become crucial as well. In this setting, the need for a WMS capable of dealing with *hybrid workflows*, i.e. to coordinate tasks running on different execution environments [15], can be a crucial aspect for performance optimisation when working with massive amounts of input data. Indeed, an *in situ* data processing strategy can prevent all the overheads related to data transfers and even to disk I/O when in-memory processing is allowed. Moreover, hybrid workflow execution becomes an absolutely mandatory requirement when dealing with federated data access or strict privacy policies.

Even if many of the existing WMSs are able to run the same workflow with a diverse set of *executors*, some of them addressing cloud environments and some others more HPC-oriented, a far smaller percentage of them can deal with multi-cloud and hybrid cloud/HPC execution environments for a single workflow. The first step to take in this direction is to waive the requirement for any shared data access abstraction among all the containers, keeping the only constraint for the WMS management node to be able to reach the whole execution environment. Such a scenario provides a significant amount of flexibility. Unfortunately, it implies that, when an inter-container data transfer must be performed, at least two copy operations are needed: a first one from the source to the management node and a second one to the destination. Sometimes this is really the only way to go, but if direct communications between container pairs are possible, then it could be better to rely on them if only to avoid overloads on the central management node. Therefore, the best strategy here would probably be to consider the two-steps copy proposed above as a baseline communication channel between every container pair while allowing users to declare better ways to exchange information when available.

From a practical point of view, the logic related to data transfers can be specified at two different levels:

- At the *host language level*, i.e. directly embedded in the business logic of the producer task. In this scenario, the only thing that the WMS can do is to check for the existence of the expected destination path before starting the data transfer process, in order to avoid useless overheads.
- At the *coordination language level*, i.e. explicitly specified by the user in the workflow description. In this scenario, the management of data transfers is left to the WMS, which can rely on a dedicated channel or fall back to the baseline strategy, as discussed above.

While the former case is quite easy to implement, the latter would require a channel abstraction, flexible enough to manage different data types (from simple values to huge file-system portions) and to deal with the aforementioned multi-container environments, potentially deployed on multi-cloud or hybrid cloud/HPC architectures. For now, to keep things a bit simpler, we decided to always rely on the baseline strategy for the inter-environment case while implementing slightly more optimised solutions for the intra-environment case whenever possible. Nevertheless, a better language specification for communication channels is, for sure, one of the most critical future improvements for the proposed approach.



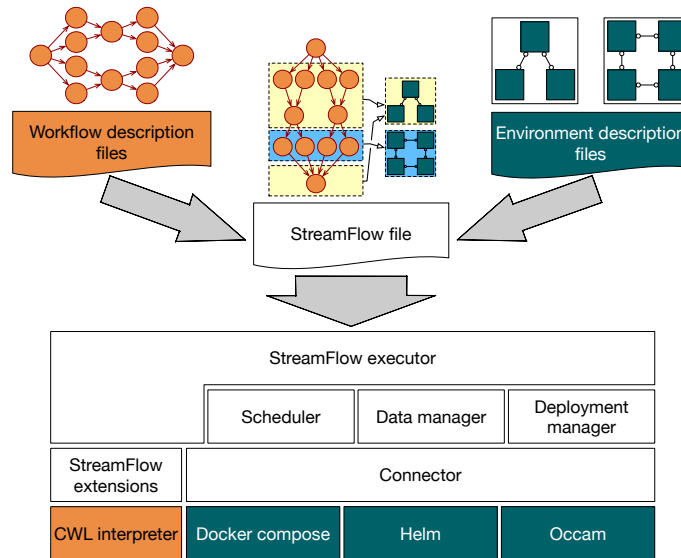


Figure 1: StreamFlow framework’s logical stack. Coloured portions refer to existing technologies, while white ones are directly part of StreamFlow code-base. In particular, the orange area is related to the definition of the workflow’s dependency graph, while the green area refers to the execution environments.

Again, the following two requirements can be used to summarise the previous discussion:

*R3* If the WMS management node is able to reach the whole execution environment, then an inter-container data transfer must always be possible, with a two-steps copy operation as the baseline strategy. Optimisations are possible for intra-environment data transfers.

*R4* If data are already present in the destination path, the WMS should avoid performing an additional copy.

## 4 StreamFlow Framework

The StreamFlow framework<sup>22</sup> has been created as a proof-of-concept WMS based on the four previously discussed requirements. Written in Python 3, it has been designed to seamlessly integrate with existing WMSs’ coordination languages, in order to allow users to extend their existing workflows without having to change what has been already done. In keeping with this point of view, we also decided not to define a new description language for multi-container environments (*models* in StreamFlow’s jargon). Instead, we built a common **Connector** API to allow for the integration with existing technologies. The StreamFlow file, which constitutes the actual entry point for a StreamFlow execution, contains pointers to workflow and models description files, the way they should relate to each other (i.e. which tasks should be executed on each type

<sup>22</sup><https://github.com/alpha-unito/streamflow>

of container, called *service* in StreamFlow) and some additional configurations. Three additional classes are instead responsible for the effective execution of tasks:

- The **DeploymentManager** class, which is able to create and destroy models when needed.
- The **Scheduler** class, which is in charge to select the best *resource* (i.e. container) on which each task should be executed while guaranteeing that all data dependencies and hardware requirements are satisfied.
- The **DataManager** class, which knows where each task’s input and output data reside and is able to transfer them among different resources whenever required.

The rest of the current section is devoted to analysing with more detail each of the aforementioned components, whose position in the StreamFlow’s logical stack is represented in Fig. 1, and how they coordinate with each other.

## 4.1 The WMS integration layer

As stated before, one of the design choices for the StreamFlow approach is to rely on existing coordination languages, instead of coming with yet another way to describe workflow models. The basic idea here is to directly work with the graphical representation of a workflow obtained by ”compiling” one or more definition files written in a coordination language of choice. In order to realise a first proof-of-concept, we decided to provide an initial integration layer for the Common Workflow Language (CWL) format [7], a YAML-based DSL mainly designed for analysis workflows. Being a fully declarative language, CWL is far simpler to understand than its Make-like or dataflow-oriented alternatives. Moreover, some existing WMSs provide at least a partial compatibility with CWL format, even when it is not their primary coordination language. Another important reason why we opted for CWL is the fact that its reference implementation, called *cwltool*<sup>23</sup>, is written in Python. This not only allowed us to use the official library to obtain the compiled workflow representation, but also to rely on existing classes for the main part of the execution process.

Therefore, what we did in practice was to provide an extension layer to the original *cwltool* codebase, using inheritance to inject additional features or to override the existing ones whenever required. Nevertheless, we have endeavoured to keep a high level of separation between CWL-specific features and the more generic StreamFlow logic, in order to be able to extend support to other coordination languages in the future easily.

## 4.2 The Connector API

Contrary to what happens for the vast majority of WMSs on the market, in StreamFlow the service allocation and the subsequent task execution happen in two strictly distinct phases. Indeed, as will be discussed in Sec. 4.5, while the task scheduling is effectively managed by StreamFlow, the containers’ lifecycle is left to an external orchestration library. A first clear advantage of this

---

<sup>23</sup><https://github.com/common-workflow-language/cwltool>

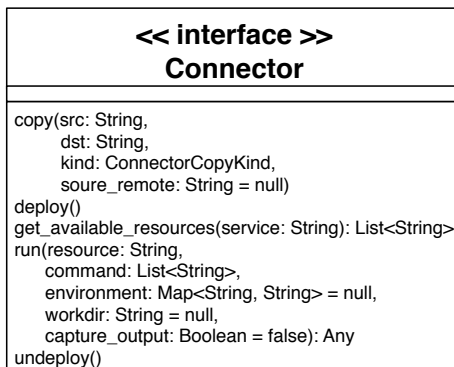


Figure 2: UML class diagram for the `Connector` interface.

approach lies in the possibility to rely on all the orchestration features provided by a mature product: autoscaling, restarting policies, affinity-based scheduling, and so on and so forth. Moreover, as behind the scenes StreamFlow demands the deployment and undeployment phases to the original orchestrator, there are no constraints on the supported features: if it works with the original library, it works with StreamFlow. Another related aspect is the adoption of the original file format to describe the models, sparing users the extra effort needed to learn a new syntax and possibly letting them use something they already know from past experiences.

For now, three different model description formats are supported: Docker Compose, Helm 2<sup>24</sup> and Occam, the supercomputing centre of Università di Torino [4]. It is worth noting that, before the development of StreamFlow, Occam did not support any way to define multi-container deployments. Indeed, even if it relies on Docker for internal nodes allocation, some privilege restrictions introduced for security purposes make it impossible for users to interact with Docker Compose or Docker Swarm. Therefore, a simple declarative format (based on YAML language) has been developed in order to make it possible to run StreamFlow workloads on top of Occam nodes.

StreamFlow interacts with each underlying orchestration technology by means of a common connector API. This adheres to the separation of concerns principle, providing an easy way to add support for additional products if required. For each model, StreamFlow creates one or more instances of a class which must extend the `Connector` interface, whose UML diagram is shown in Fig. 2. From Fig. 1 it is possible to notice how such low-level interface is used by all the other components in StreamFlow’s logical stack. For example, the `deploy` and `undeploy` methods are called by the `DeploymentManager` class to manage the life-cycle of each model, while the `get_available_resources` method is invoked by the `Scheduler` class to obtain all the replicas of a given service in the model. The `copy` method is instead used by the `DataManager` class to perform data transfers among resources, with the `kind` argument specifying the direction of the transfer operation: from the local StreamFlow management node to a remote resource (`localToRemote`), from a remote resource to the management

<sup>24</sup><https://v2.helm.sh/>

node (`remoteToLocal`) or between two remote resources (`remoteToRemote`). In the last case, the `source_remote` argument is used to specify the resource into which the data reside. Finally, the `run` method is used to execute a command on top of a remote resource and potentially to capture the generated output value.

### 4.3 The StreamFlow file

When launching a StreamFlow execution, the only argument it takes is the path of a YAML file, which in this article is referred to as the StreamFlow file. A valid StreamFlow file contains the version number (which currently only accepts the `v1.0` value) and two main sections.

The `models` section contains a dictionary with one or more uniquely named models. Each model is an object with two distinct fields:

- The `type` field identifies which `Connector` implementation should be used for its creation, destruction and management. For now, this field can take three different values (`docker-compose`, `helm` and `occam`) which correspond to the three connector types currently supported.
- The `config` field contains a dictionary with configuration parameters for the corresponding `Connector`, including paths to one or more description files. Such parameters are directly passed to the `Connector`'s constructor at deployment time, making it very easy to extend the supported set of configurations if a new version of the underlying orchestration library comes out.

The `workflows` section contains a dictionary with one or more uniquely named workflows to be executed in the current run. It is important to notice that, for now, different workflows are totally independent of each other, in that an entire StreamFlow logical stack is allocated for each of them. This means that, even if two tasks in two different workflows can refer to the same environment description file, two different models will be allocated for their execution. In the StreamFlow file specification, each workflow is an object containing the `type` and `config` fields, as for the previously described model object, plus an additional `bindings` list. At the moment, the `type` field only accepts the `cwl` value, since it is the only configuration language currently supported, while the `config` field contains the paths to the CWL files describing the workflow model.

The `bindings` list contains the task-model associations represented by the curly braces in Fig. 1. Since for simplicity purposes each task can be associated with a single service (as pointed out in requirement *R2*), there is the need to uniquely identify both a service inside a model and a task inside a workflow, given the fact that workflows and models themselves are uniquely identified by their names. Considering workflow models as dependency graphs, typically each node in such representation can refer to either a simple task or a nested sub-workflow. Therefore, we decided to adopt a file-system based mapping of each task to a Posix-like path, where:

- Each simple task is mapped to a file.
- Each sub-workflow is mapped to a folder, which can contain both files and sub-folders. In particular, the most external workflow description is mapped to the root folder.

Such method allows for easy identification of tasks, given that there exists an intuitive way to assign a name to each task in the workflow’s graphical structure and that such name has the uniqueness constraints required by a typical file-system representation. Since both these requirements are satisfied by the CWL coordination language (and also by the vast majority of coordination languages proposed by WMSs on the market), we decided to adopt this strategy.

For its part, a service can be uniquely identified if there is a way to assign a name to it and if such name is unique inside its model. In this case, the best mapping strategy for services strictly depends on the model specification itself. For Docker Compose, where the unit of deployment is a single container, it is enough to take a key in the `services` dictionary to uniquely identify the related service. Moreover, since an Occam description file is basically equivalent to the `services` section of a Docker Compose file, the same strategy can be applied to it, too. Unfortunately, in Kubernetes (and consequently in Helm) the unit of deployment is a Pod, which can contain multiple containers inside it. In this case, the user is required to explicitly add the `name` attribute for each container in the Pod template and to ensure the uniqueness of such name in the context of the whole Helm release.

Starting from all these considerations, we derived a format for the `bindings` list. In particular, each element of such list is an object with two attributes:

- A `target` attribute, which in its turn is an object with a `model` and a `service` attributes that uniquely identify a service according to the aforementioned mapping techniques.
- A `step` attribute, referring to a file or a folder in the aforementioned file-system abstraction. If the path resolves to a folder, the same target service is applied recursively in the file-system hierarchy, unless a more specific configuration (i.e. another entry in the `bindings` list with a deeper path in its `step` field) overrides it.

The whole specification for the current version of the StreamFlow file is contained in a JSON Schema file named `config.schema.json`. Since such file is also used in the validation phase during a StreamFlow execution, it represents the authoritative source of truth for the StreamFlow file format. Therefore, we invite the interested reader to search for it in the StreamFlow code repository for further details.

## 4.4 Task scheduling

The task scheduling strategy is a fundamental component of a WMS, mainly for the huge impact it has on the overall execution performances. It is a common practice for WMSs to allow users to specify some minimum hardware requirements for a task, e.g. in terms of the number of cores or the amount of memory. Such requirements are generally configurable by means of optional parameters in the coordination language, while the effective mapping on top of adequate worker nodes is left to the implementation of the specific executor.

Obviously, it is much easier for a scheduling algorithm to work with *homogeneous* resource pools, in which all the nodes have the same characteristics in terms of cores, memory, persistence and so on and so forth. Nevertheless, in a real scenario, it is very likely that different tasks require very diverse amounts of

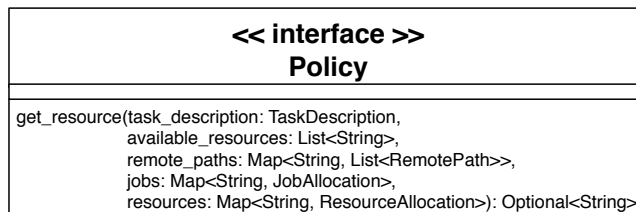


Figure 3: UML class diagram for the Policy interface.

resources, resulting in sub-optimal workloads for homogenous pools. The case of hybrid workflows is even more complicated, since the non-uniform data access makes it particularly important to rely on data locality whenever suitable, trying to minimise the need for data transfers among different models. In general, however, all container-based WMSs tend to tightly-couple the allocation of a container with the subsequent execution of the task inside it. In this setting, all the available worker nodes are ultimately identified by the amount of computing power they can provide.

The StreamFlow approach introduces an additional level of complexity here. Indeed, since requirement *R1* explicitly states that in StreamFlow the unit of deployment should be a complex environment with different containers, it is no longer true that a task can be executed on any worker node equipped with enough hardware. In a way, in this setting, the services exposed by each container can be identified as *capabilities*, and a task can be executed on top of it only if all its *requirements* are satisfied. StreamFlow manages this requirement-capability association in a straightforward way, by identifying each container type with a single service, according to requirement *R2*, and specifying which service is required by each task (as described in Sec. 4.3).

Since in StreamFlow the resource allocation for services in a model is managed by an external orchestration library, eventually related constraints should be specified in the environment description file (as will be better discussed in Sec. 4.5). Task-related resource constraints, specified in the workflow description, and requirement-capability associations, specified in the StreamFlow file, are instead directly managed by the **Scheduler** class when selecting the target resource inside a specific model. Even if only a single target service can be specified for each task, multiple replicas of the same service could exist at the same time and, if the underlying orchestrator provides auto-scaling features, their number could also change in time. It is the responsibility of the **Scheduler** class to both extract the list of compatible resources for a given task, by calling the previously introduced `get_available_resources` method of the appropriate **Connector** instance, and then to apply some scheduling policy to find the best target. Finally, a new *job* can be allocated to that resource in order to execute the task.

Given the very complex nature of the execution environments managed by StreamFlow, it is improbable that a universally best scheduling strategy actually exists. Indeed, many different factors as data locality, load balancing, and so on and so forth can affect the overall workflow execution time. For this reason, we decided to implement a **Policy** interface to allow users to implement their

custom strategies. As can be seen from the UML class diagram shown in Fig. 3, the `Policy` interface only contains a single method, called `get_resource`, with five input arguments:

- The `job_description` argument contains a characterisation of the current task in terms of resource requirements and data dependencies. This allows a scheduling strategy to take into account a combination of available hardware resources and data locality when evaluating different allocation opportunities.
- The `available_resources` argument is the list of all the resources which satisfy the requirement-capability association for the current task.
- The `remote_paths` argument contains, for each file explicitly managed by the WMS, the list of its remote copies. Each remote path is uniquely described by the combination of the resource name on which it resides and its file path on that resource. This information can be used by a scheduling policy to take into account data locality in its algorithm, perhaps by scheduling a task on the same resource on which resides its maximal-sized data dependency.
- The last two arguments describe the previously allocated tasks. The `JobAllocation` class contains the description of a previously allocated task, the resource to which it has been assigned and its status (`running`, `completed` or `failed`). For its part, the `ResourceAllocation` class contains the related model and service of an existing resource and the list of jobs assigned to it. These data allow scheduling strategies implementing some load-balancing features, perhaps by using the number and type of the tasks currently `running` on a resource as a proxy for its effective load.

The `StreamFlow Scheduler` class processes fireable tasks according to a simple First Come First Served (FCFS) order, without allowing for explicit preemption of tasks. Moreover, since each scheduling policy can only process one task at a time, all those strategies that require a global knowledge of the jobs queue, as the various flavours of backfilling or a Shortest Job First approach, cannot currently be implemented. Even if this can result in suboptimal scheduling solutions in some cases, the proposed approach drastically reduces the implementation complexity, which is an important aspect for proof-of-concept works.

Concerning the default scheduling policy, `StreamFlow` relies on a simple and very generic strategy based on data locality. When a task becomes fireable, the algorithm iterates over all its data dependencies and tries to reserve the first resource without jobs in `running` status and which satisfies the hardware constraints. If such a resource does not exist, then the algorithm iterates over all the remaining available resources, trying to find a suitable one. If the search fails again, then a `null` value is returned. In this case, the task is inserted into a waiting queue, and a new scheduling attempt will be performed as soon a `running` job notifies its termination in `completed` status.

## 4.5 Model life-cycle management

As pointed out before, `StreamFlow` strictly decouples the model life-cycle management and the task scheduling phases. As shown in Fig. 4, from a theoretical

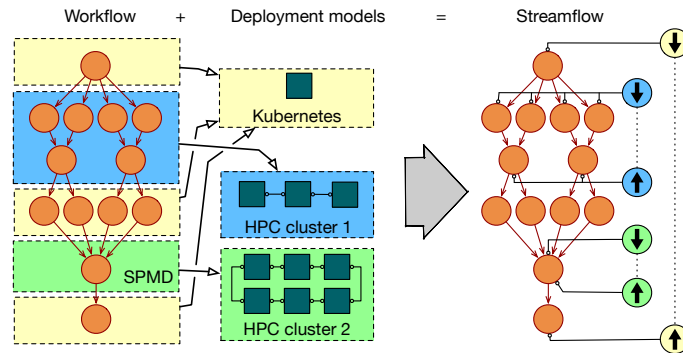


Figure 4: Workflow graph transformation to include model deployment and un-deployment tasks. Orange nodes represent original tasks, while the others refer to model deployment (downward pointing arrow) and un-deployment (upward pointing arrow) phases.

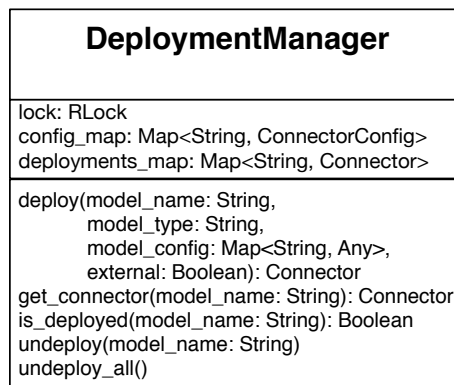


Figure 5: UML class diagram for the DeploymentManager class.

point of view, this can still be represented with a traditional workflow model by transforming the original dependency graph in order to include two new special kinds of tasks:

- The *deploy* task, which synchronously creates a new model. This task does not depend on anything else, but all the tasks that should be executed in such model must depend on it.
- The *undeploy* task, which destroys an existing model. No other task depends on it, but it should depend on all the tasks that must be executed on such model, in order to wait for their termination before starting the un-deployment process.

The result of this transformation is a perfectly fine dependency DAG, which satisfies requirement *R1* and can be correctly described by the vast majority of coordination languages made available by other WMSs on the market.



Nevertheless, a far more practical strategy would be to let a model be deployed by the first fireable task which requires it. Indeed, a *lazy* allocation strategy can save resource allocation time, which in cloud infrastructures is commonly proportional to money spent. Then, when there are no more tasks needing it, a model can be undeployed. The `DeploymentManager` class, whose UML diagram is represented in Fig. 5, has precisely the role of executing these actions, relying on the underlying orchestration library by means of the `Connector` API. In particular, the `deploy` method atomically checks if a model has been already deployed and, if not, it builds a new `Connector` instance from the configuration provided in the `StreamFlow` file and stores it into the `deployments_map` data structure. The `external` attribute then determines if the model should effectively be deployed or not: if such attribute is equal to `true`, the `DeploymentManager` confines itself to the creation of a new `Connector` instance, thus allowing the user to manage the related model’s life-cycle externally.

Here a `lock` is necessary to avoid race conditions when multiple threads or processes request the deployment of the same model in a concurrent fashion. Indeed, in order to satisfy requirement *R2*, the same model must only be deployed once, while subsequent calls to the `deploy` method should only return a new `Connector` instance referring to it. The choice to create a new `Connector` instance at each method invocation comes from the need to avoid potential resource conflicts (e.g. sockets of buffers) in a concurrent execution without introducing the unnecessary overhead of fully-atomic access to the `Connector` methods. For the same reason, the `get_connector` method creates a new instance every time it is invoked, too.

As discussed before, when the last task needing a particular model is completed, the corresponding model should be undeployed. This logic is quite easy to implement when the entire workflow DAG is known a priori, but it can be a bit more complicated when new tasks can be added at runtime. Probably the best strategy for the second case would be to set a grace period, after which the model is undeployed if none have asked for it. For now, `StreamFlow` confines itself to undeploy all the models at the end of the entire workflow execution by using the `undeploy_all` method. This is a conservative strategy, but it can lead to resource wastes if some models remain unused for a long time. The same method is also invoked by `StreamFlow`’s main process if an unhandled exception is raised, in order to prevent a potential waste of resources in case of failure.

## 4.6 Data transfers

As pointed out in Sec. 3.2, data transfers play a fundamental role in hybrid workflow executions. Unfortunately, in order to support an inter-model allocation of the tasks, it is necessary to waive the comfort brought by a globally shared data space, making it necessary for the WMS to explicitly move the data whenever required. Since large data transfers are very time consuming operations, especially for long distances and in the absence of dedicated high-throughput communication networks, a good scheduling policy for hybrid workflows should try to rely on data locality as much as possible (as described in Sec. 4.4). At the same time, it is also essential to ensure that the best communication channel between two endpoints is always selected to perform a data transfer and that all unnecessary data movements are avoided, in order to further reduce overheads.

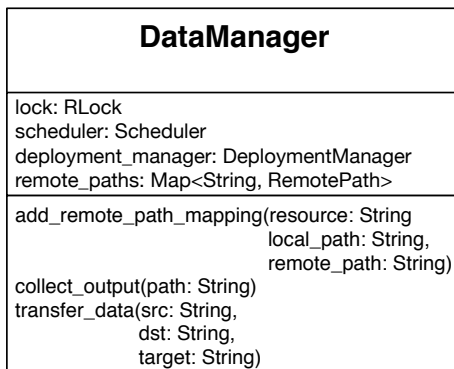


Figure 6: UML class diagram for the `DataManager` class.

The StreamFlow framework has been designed in order to meet requirements *R3* and *R4*, which represent two fundamental steps in this direction. In particular, with a view to the separation of concerns principle, a dedicated `DataManager` class has been developed with the precise goals of keeping track of the remote locations of each data dependency and performing data transfers between successive steps. As can be seen from its UML class diagram in Fig. 6, the `DataManager` class contains pointers to both the `Scheduler` and the `DeploymentManager` classes discussed above. Moreover, it contains a data structure, called `remote_paths`, which is equivalent to the namesake argument of the `Policy` interface introduced in section 4.4. Whenever a task terminates in `completed` status, it is in charge of populating such structure with the remote position of all its output files and folders by calling the `add_remote_path_mapping` method. The `lock` is necessary to protect such structure from concurrent accesses.

The same `remote_paths` structure is also used by the `data_transfer` method to check where data dependencies of a task reside in the execution environment. Indeed, every time a task needs a file or a folder from one of its predecessors, such method is called to verify if a data transfer is actually needed or not. Firstly, it is necessary to satisfy requirement *R4* by checking if data are already present on the target resource. This is always true when both tasks run on the same resource, but can also be verified if the two resources share a data space (e.g. a persistent volume) or if a task explicitly performs a data transfer before completing. It is worth noting that some WMSs always copy input data to a staging folder, in order not to compromise the original data when in-place modifications are performed by a task. Since also StreamFlow adopts such behaviour, a local copy is also performed when data are already present on the target resource. However, such operation obviously adds a negligible amount of overhead when compared with a remote data transfer.

If the destination path does not exist, then a data movement is unavoidable. If the source and the target resources belong to distinct models, then StreamFlow adopts the baseline strategy mentioned in requirement *R3*, performing the first transfer from the source resource to the management node and a second copy to the target resource. Instead, if the two resources belong to

the same model, the transfer is directly performed by the `copy` method of the corresponding `Connector` implementation. In the latter case, some optimisations are possible. For example, since all Occam nodes share the `/archive` and `/scratch` portions of the file-system, only a local copy on the target resource is required to transfer a data dependency which resides in one of such folders.

Finally, the `collect_output` method performs a data transfer from a remote resource to the local management node. This method is always called before a remote resource is undeployed, in order to retrieve the final output of the workflow model. Moreover, when a task must be performed locally but requires some remote input data, this method is called before starting its execution.

## 5 Single-Cell Application Use-case

Bulk sequencing of biomedical samples provides an average response across the entire cell population, and it is not fully representative of any one cell. The population average may mask the reaction of a single cell, although such heterogeneity can be of critical importance when attempting to develop accurate disease models, or elucidating patient responses to specific therapies. Fundamentally, the analysis of individual single cells from a heterogeneous population enables the reduction of biological noise and offers the ability to investigate and characterise rare cells.

The power of single-cell sequencing is crucial in transcriptomics in order to study the activity of every single cell in a sample. It relies on the reverse transcription of RNA to complementary DNA and subsequent amplification by reverse transcription before deep sequencing. One of the most popular platforms for single-cell analysis is marketed by 10X Genomics, which can encapsulate 500 to 20,000 cells per library into nano-droplets together with micro-beads that can tag each different transcript before amplification. This is possible thanks to the 750000 different UMIs that are anchored to each bead together with sequencing adapters.

The problem with this technique is the noise that is exaggerated by the need for very high amplification from the small amounts of RNA found in an individual cell. Although technical noise confounds precise measurements of low-abundance transcripts, modern protocols have progressed to the point that single-cell measurements are rich in biological information. Indeed, many exploratory studies have already led to insights into the dynamics of differentiation, cellular responses to stimulation and the stochastic nature of transcription. Moreover, the single-cell analysis allowed the identification of tumour polyclonality and explained some of the mechanism of relapse after chemotherapy.

A critical limitation of the single-cell analysis is the use of a complex algorithm to control noise and cluster cells with similar expression profile. This requires many statistics and repeating the procedures many times to identify the right thresholds for the sample in analysis. In this context, the processing power and the automatic management of the analysis is of critical importance, since analysing each cell in a population requires hundreds-of-thousands to millions of comparisons to be processed in a high throughput manner.

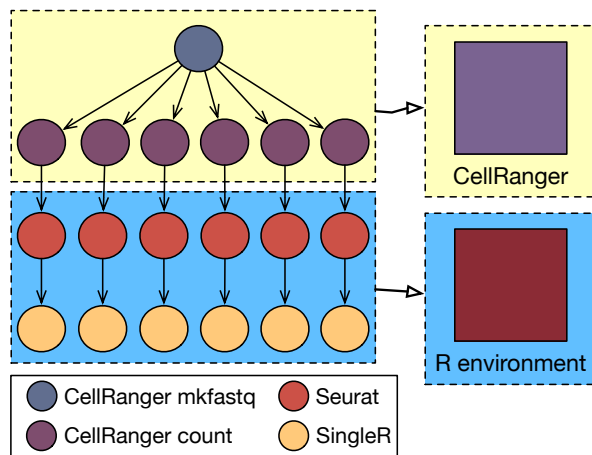


Figure 7: Dependency graph and model bindings for the single-cell workflow. In this case, the first step creates six different sequences, which can then be processed independently of each other for the remaining three steps.

## 5.1 Application Pipeline

Novel single-cell transcriptome sequencing assays allow researchers to measure gene expression levels at the resolution of single cells and offer the unprecedented opportunity to investigate fundamental biological questions at the cellular level, such as stem cell differentiation or the discovery and characterisation of rare cell types.

A typical pipeline for single-cell transcriptomic data analysis, like the one represented in Fig. 7, relies on a data structure that represents the read counts for each gene in each cell. Accordingly, the analysis pipelines can be broadly divided into two main parts: the creation of the count matrix and its statistical analysis.

The first step for the creation of the count matrix is performed according to the single-cell isolation technology and the sequencing approach. For example, considering a typical 10x genomics experiment followed by an Illumina Novaseq sequencing, the first part of the pipeline will be performed using a tool called CellRanger [44]. In particular, this part of the analysis will consist in two steps: the creation of the fastq files from the flowcell provided in output by the sequencer and the alignment of the reads against the reference genome.

The fastq creation is performed by looking at the images generated by the sequencer cycle after cycle into the flowcell on which the sequences have been hybridised. From the computational point of view, the algorithm looks at the images and calls the bases for each position. It also provides, for each base in each read, a quality score according to the accuracy by which the base has been called. Moreover, the algorithm is able to divide the sequences in chunks, according to what indicated in the configuration file, in order to identify the different parts of each sequence:

- The cellular barcode, that identifies the cell by which the transcript has been captured.

- The UMI (Unique Molecular Identifier), which uniquely identifies the transcript allowing the removal of PCR duplicates.
- The transcript sequence, which corresponds to the actual part of the expressed gene that has been captured.

The second step performed by CellRanger is the creation of the count matrix itself, a process that requires two distinct procedures. First, sequences that have been generated in the previous step are aligned against the reference genome using STAR, which is the most popular aligner currently available for transcriptomic analysis. For each read, the corresponding barcode and UMI are tracked along with the genome position to which the read aligns. Using the UMI, reads coming from the same transcript are collapsed in a unique hit. These alignments are then processed according to the genome annotation, in order to recapitulate for each gene how many reads have been captured. According to the specified parameters, usually only reads with a single hit are retained in the final matrix count, which represent, for each cell (columns) and for each gene (row) how many transcripts have been captured.

Once the count matrix has been computed, a quantitative analysis of the results is usually performed. The aim is clustering cells having similar transcriptomic profiles and characterise them according to some reference databases. This can be performed using ad-hoc developed software in Python or R, the latter being probably the most popular at the moment. In the context of this pipeline, we used two main R packages for the analysis of the count matrix: Seurat [12, 37] for the normalisation, dimensionality reduction and clustering of cells, and SingleR [8] for labelling the clusters according to public databases of single-cell data annotation.

In particular, Seurat is used to loading data in the R environment and filtering cells that are outliers for specific statistics, such as the number of unique transcripts or the presence of mitochondrial transcripts. Data are then normalised, to take into account the different coverage in the different cells, and the most variable genes are identified. These genes are used to perform a dimensionality reduction through the computation of principal component analysis. Cells are then clustered using the Louvain algorithm, which has been specifically designed for detecting communities in networks. In particular, it maximises a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities by evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network. At last, the marker genes for each cluster are identified, by comparing the expression profile of the cells inside the cluster with all the other cells. Since the count matrix is usually quite sparse, a specific algorithm, called MAST, has been used to identify these markers.

Once clusters have been identified, the pipeline uses an R package called SingleR in order to label each cluster and characterise its cells in an unbiased way. SingleR leverages reference transcriptomic datasets of pure cell types to infer the cell identity for each single cells independently. In particular, SingleR starts by calculating a Spearman coefficient for each cell in the data set, using only variable genes, thus increasing the ability to distinguish closely related cell types. This process is performed iteratively using only the top cell types from the previous step and the variable genes among them until only one cell type remains.

## 5.2 StreamFlow Implementation

From a practical perspective, the design of a StreamFlow application can be split into three high-level steps:

- The design of the workflow dependency graph, using a coordination language of choice among those supported by the framework. For now, as discussed in Sec. 4.1, CWL is the only possible choice, so we obviously opted for it.
- The design of the execution environment, in terms of one or more models containing one or more containers each. Here we decided to experiment two different combinations of Occam and Helm environments, as better detailed below in this section.
- The creation of a StreamFlow file, as described in Sec. 4.3, in order to wrap things together and to provide a unique entry point for the StreamFlow execution.

Fig. 7 provides a graphical representation of the whole StreamFlow model for a single-cell pipeline of the kind described in Sec. 5.1. In this case, the workflow dependency graph is a simple DAG with four different kinds of tasks. In terms of workflow patterns [39], it can be represented as an initial parallel split, with a fan-out equal to the number of sequences produced by the first task (six in this case), followed by as many independent sequence blocks of three tasks each. It is also worth noting that, since none of the tasks can be executed in a distributed fashion, the maximum number of nodes from which the workflow execution can take some benefit is equal to the fan-out of the initial parallel split.

Concerning the required services, the first two types of tasks are executed by CellRanger, a Linux-compatible tool downloadable as a standalone tar package, while the last two tasks require two main R packages, called Seurat and SingleR respectively, plus all the related dependencies. We decided to split these requirements into two distinct images, the first one providing the execution environment for the first two tasks and the second one with all the R packages needed to execute the others.

Partitioning the tasks with respect to their target container, we obtain two disjoint subsets, each of which can execute concurrently on a maximum of six nodes. Therefore, if enough hardware resources are available, the best strategy would be to allocate six replicas of each image. From the perspective of task-container associations, this configuration is perfectly equivalent to an STSC approach, where each container is managed separately by the WMS (as described in Sec. 3.1). Moreover, since the time needed by a container to be ready to execute tasks is totally negligible with respect to the time required for the completion of tasks themselves, even a configuration with less than six container replicas ends up being practically equivalent to an STSC execution. Given that, it should be clear that requirements *R1* and *R2* of Sec. 3.1 do not bring an additional concrete value to this workflow.

Conversely, the hybrid workflow execution enabled by requirements *R3* and *R4* of Sec. 3.2 can be very useful, for example, to perform a data preprocessing phase on a dedicated HPC structure before moving data to the cloud to complete the remaining steps. Indeed, in the examined case the total size of the

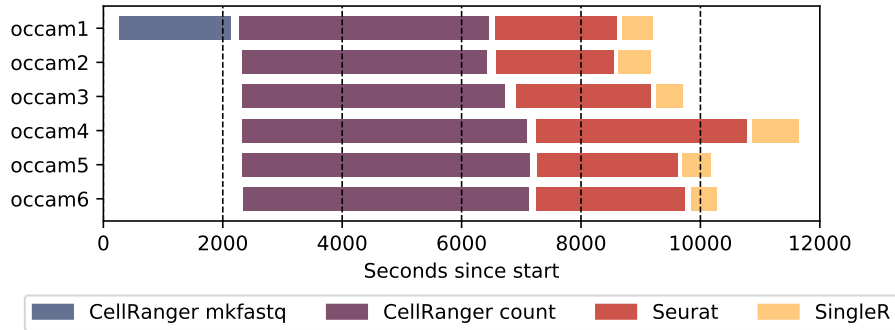


Figure 8: Execution timeline for the StreamFlow single-cell application on six Occam nodes, each allocated to both a CellRanger and an R environment containers.

initial data is almost 60GB, but modern sequencing machines can achieve 10 billion of sequences per flowcell, corresponding to about 3TB of data. Moreover, the `cellranger count` command, executed by the second kind of task in Fig. 7, requires a quite high amount of resources to be performed: the official documentation reports 8 cores and 32GB of memory as minimum requirements, but a significant speedup can be appreciated until up to 32 cores and 128GB of memory.

If hybrid workflows were not supported, the best strategy would be to execute the entire set of tasks on top of six HPC nodes, in order to take full advantage of the available grade of parallelism while avoiding data transfers. Moreover, when using total wall clock time as the only evaluation metric, this one keeps being the best solution also when compared with hybrid alternatives. Therefore, it is worth to use this setting as a baseline, in order to evaluate the significance of performance loss when switching to a mixed HPC/cloud configuration.

We reserved six Light nodes on the Occam facility, each of which having 2x Intel Xeon E5-2680 v3 (12 core each, 2.5GHz) CPUs and 128GB (8x16, 2133MHz) of memory, and prepared a model which allocates each node to both a CellRanger and an R environment containers. As mentioned in Sec. 4.6, all Occam nodes share the `/archive` folder, mounted as an NFS export, and the `/scratch` folder, with a LUSTRE parallel file-system. We copied initial data on the `/archive` file-system and configured StreamFlow to use a folder on the `/scratch` hierarchy as its output folder. In this way, data could be accessed by dependent tasks without the need for explicit transfers. Then we ran the StreamFlow application inside a container launched on an additional Occam node.

The timeline for this execution is reported in Fig. 8. The whole duration is about three hours and a quarter, dominated by the CellRanger count and the Seurat commands. White space between subsequent bars represents the time needed by StreamFlow itself to perform some internal tasks before launching a new command, including copying the input data on a staging folder (as mentioned in Sec. 4.6). Nevertheless, the time taken to perform each of

these operations is negligible with respect to the time needed to complete tasks themselves.

In a real scenario, it would be probably better to dedicate the HPC structure to the completion of the first tasks, while executing the rest of the workflow directly on a cloud environment. Indeed, the output data of the last task must often be stored into a database or visualised in a web application, and the cloud is undoubtedly the most natural place to host such kind of applications. By observing intermediate data in the workflow model, it is possible to notice that output data of the second task have a total size of about 15-30MB, while the third task produces output data for more than 200MB. Given that, in order to minimise the overhead introduced by a data transfer, the best strategy would be to execute the first two tasks on an HPC facility and the remaining two in a cloud infrastructure.

We configured a virtualised Kubernetes cluster on top of the GARR<sup>25</sup> cloud, based on OpenStack<sup>26</sup>, containing six worker nodes with 4 virtual CPUs and 8GB of memory each. Then we prepared two different models:

- A first model with six Occam nodes, with an instance of the CellRanger container allocated on each of them
- A second model with six Kubernetes Pods, each with an instance of the R environment container and a `podAntiAffinity` parameter to ensure that each Pod is allocated on a different worker node whenever possible.

On Kubernetes, the StreamFlow output folder of each container has been mapped to a persistent volume managed by Cinder, the OpenStack's block storage service, configured with a `readOnlyOnce` access mode. This means that no shared data space exists between different worker nodes. Nevertheless, the scheduling policy described in Sec. 4.4 makes it so that each SingleR task is executed by the node where its required input data already reside, removing the need for additional data transfers. Given that, since we kept running StreamFlow application inside an Occam node, the only unavoidable data movement is from Occam to Kubernetes, between the second and the third tasks.

The timeline for this second run is reported in Fig. 9. The first important thing that can be observed is how the whole duration of this hybrid execution is comparable with the previous full-HPC configuration. This is mainly due to the combination of two factors. Firstly, the time needed to transfer data from the Occam facility to the GARR cloud is negligible when compared with the time needed to complete the tasks themselves. Moreover, the Seurat task seems not to benefit so much from additional computing power, making it quite useless to commit HPC machines for its execution. In a situation like this, it is pretty clear that the StreamFlow approach can be beneficial to obtain a more efficient resource usage without significant performance drops.

## 6 Conclusion and Further Development

The recent explosion in popularity faced by lightweight containerisation technologies also invested the scientific workflows' ecosystem, with undoubted gains

---

<sup>25</sup><https://garr.it/it/>

<sup>26</sup><https://www.openstack.org/>



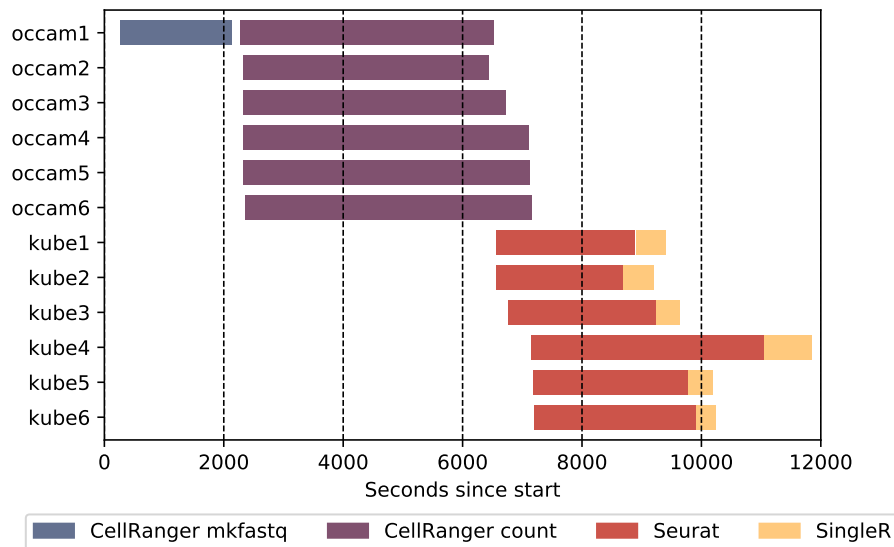


Figure 9: Execution timeline for the StreamFlow single-cell application in a hybrid configuration, with six Occam nodes allocated to CellRanger as many replicas and six Kubernetes worker nodes allocated to as many R environment containers.

in portability and reproducibility. During the very last years, a significant number of WMSs started to include container-based workflow executions among their features, while new container-native alternatives began to appear. Nevertheless, some common simplifications in the design process can prevent a WMS to fully exploit the potential of containerisation technologies.

This work aims at exploring the potential benefits deriving from waiving two common properties of existing WMSs. Firstly, a one-to-one task-container mapping prevents the execution of tasks in multi-container environments and makes it unnecessarily difficult to support concurrent executions of communicating tasks. Moreover, the requirement for a single shared represents an obvious obstacle for hybrid workflow executions, which could instead highly benefit from containers' portability properties.

The StreamFlow framework has been developed as a proof-of-concept WMS which explicitly drops these constraints by design. In StreamFlow, the unit of deployment is a complex multi-container environment, directly managed by an underlying orchestration technology. Moreover, each container can exchange files with every other, with the only constraint for the WMS management node to be able to reach the whole execution environment. This second feature has been used to run a bioinformatics workflow on top of a hybrid HPC/cloud environment without significant performance losses, therefore showing the potential benefits introduced by the proposed approach in terms of more efficient resource usage.

The next crucial step now is to investigate benefits brought by multi-container deployment units in scientific applications. Potential forthcoming candidates

for experimentation are all those applications which require distributed execution, as MPI-based simulations or distributed deep learning frameworks. In case of positive feedback, some further developments will be necessary to evolve StreamFlow in a mature product, as the support for more coordination languages and orchestration libraries. Moreover, as previously mentioned, a robust abstraction for inter-container communication channels would significantly reduce performance losses introduced by large data transfers.

## Acknowledgment



This article describes work undertaken in the context of the Deep-Health project, “*Deep-Learning and HPC to Boost Biomedical Applications for Health*” (<https://deephealth-project.eu/>) which has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No. 825111 [13]. This work has been partially supported by the HPC4AI project <http://www.hpc4ai.it> [6].

## References

- [1] Why use workflows? <https://taverna.incubator.apache.org/introduction/why-use-workflows>. Accessed: 2020-02-16.
- [2] E. Afgan, D. Baker, M. van den Beek, D. J. Blankenberg, D. Bouvier, M. Cech, J. Chilton, D. Clements, N. Coraor, C. Eberhard, B. A. Grüning, A. Guerler, J. Hillman-Jackson, G. V. Kuster, E. Rasche, N. Soranzo, N. Turaga, J. Taylor, A. Nekrutenko, and J. Goecks. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic Acids Research*, 44(Webserver-Issue):W3–W10, 2016.
- [3] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, SWEET@SIGMOD 2012, Scottsdale, AZ, USA, May 20, 2012*, page 1, 2012.
- [4] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino. Occam: a flexible, multi-purpose and extendable HPC cluster. In *Journal of Physics: Conf. Series (CHEP 2016)*, volume 898, page 082039, San Francisco, USA, 2017.
- [5] M. Aldinucci, H. L. Bouziane, M. Danelutto, and C. Pérez. STKM on SCA: a unified framework with components, workflows and algorithmic skeletons. In *Proc. of 15th Intl. Euro-Par 2009 Parallel Processing*, volume 5704 of *LNCS*, pages 678–690, Delft, The Netherlands, Aug. 2009. Springer.
- [6] M. Aldinucci, S. Rabellino, M. Pironti, F. Spiga, P. Viviani, M. Drocco, M. Guerzoni, G. Boella, M. Mellia, P. Margara, I. Drago, R. Marturano, G. Marchetto, E. Piccolo, S. Bagnasco, S. Lusso, S. Vallero, G. Attardi,

- A. Barchiesi, A. Colla, and F. Galeazzi. HPC4AI, an AI-on-demand federated platform endeavour. In *ACM Computing Frontiers*, Ischia, Italy, May 2018.
- [7] P. Amstutz, M. R. Crusoe, N. Tijani, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, J. Kern, D. Leehr, H. Mnager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, and L. Stojanovic. Common workflow language, v1.0, 2016.
- [8] D. Aran, A. P. Looney, L. Liu, E. Wu, V. Fong, A. Hsu, S. Chak, R. P. Naikawadi, P. J. Wolters, A. R. Abate, A. J. Butte, and M. Bhattacharya. Reference-based analysis of lung single-cell sequencing reveals a transitional profibrotic macrophage. *Nature Immunology*, 20(2):163–172, 2019.
- [9] M. P. Atkinson, S. Gesing, J. Montagnat, and I. J. Taylor. Scientific workflows: Past, present and future. *Future Generation Comp. Syst.*, 75:216–227, 2017.
- [10] R. Badia, E. Ayguade, and J. Labarta. Workflows for science: A challenge when facing the convergence of HPC and big data. *Supercomput. Front. Innov.: Int. J.*, 4(1):2747, Mar. 2017.
- [11] S. C. Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsén, P. Larmande, Y. L. Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, and C. Blanchet. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Comp. Syst.*, 75:284–298, 2017.
- [12] A. Butler, P. Hoffman, P. Smibert, E. Papalexi, and R. Satija. Integrating single-cell transcriptomic data across different conditions, technologies, and species. *Nature Biotechnology*, 36(5):411–420, 2018.
- [13] M. Caballero, J. Gomez, and A. Bantouna. Deep-learning and hpc to boost biomedical applications for health (deephealth). In *2019 IEEE 32nd International Symposium on Computer-Based Medical Systems (CBMS)*, pages 150–155, Los Alamitos, CA, USA, June 2019. IEEE Computer Society.
- [14] V. Cima, S. Böhm, J. Martinovic, J. Dvorský, K. Janurová, T. V. Aa, T. J. Ashby, and V. I. Chupakhin. Hyperloom: A platform for defining and executing scientific pipelines in distributed environments. In *Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM@HiPEAC 2018, Manchester, United Kingdom, January 23-23, 2018*, pages 1–6, 2018.
- [15] R. F. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Comp. Syst.*, 75:228–238, 2017.
- [16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

- [17] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and R. K. Wenger. Pegasus, a workflow management system for science automation. *Future Generation Comp. Syst.*, 46:17–35, 2015.
- [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. *Computing in Science and Engineering*, 21(4):22–36, 2019.
- [19] P. Di Tommaso, M. Chatzou, E. W. Floden, et al. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, Apr. 2017.
- [20] M. Dorier, J. M. Wozniak, and R. Ross. Supporting task-level fault-tolerance in HPC workflows by launching MPI jobs inside MPI jobs. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, WORKS '17, pages 5:1–5:11, New York, NY, USA, 2017. ACM.
- [21] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. L. Truong, A. Villazón, and M. Wiczorek. ASKALON: A development and grid computing environment for scientific workflows. In *Workflows for e-Science, Scientific Workflows for Grids*, pages 450–471. 2007.
- [22] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972.
- [23] R. L. Henderson. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing, IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995, Proceedings*, pages 279–294, 1995.
- [24] J. Köster and S. Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.
- [25] M. Kotliar, A. V. Kartashov, and A. Barski. CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language. *GigaScience*, 8(7), 07 2019.
- [26] N. Kulkarni, L. Alessandrì, R. Panero, M. Arigoni, M. Olivero, G. Ferrero, F. Cordero, M. Beccuti, and R. A. Calogero. Reproducible bioinformatics project: a community for reproducible bioinformatics analysis pipelines. *BMC Bioinformatics*, 19(10):349, 2018.
- [27] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.
- [28] J. Liu, E. Pacitti, and V. P. et al. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):pp 457–493, Dec. 2015.
- [29] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

- [30] F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia. Enabling cloud interoperability with compss. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 16–27, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [31] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.
- [32] P. Moreno, L. Pireddu, P. Roger, N. Goonasekera, E. Afgan, M. van den Beek, S. He, A. Larsson, D. Schober, C. Ruttikes, D. Johnson, P. Rocca-Serra, R. J. Weber, B. Gruening, R. M. Salek, N. Kale, Y. Perez-Riverol, I. Papatheodorou, O. Spjuth, and S. Neumann. Galaxy-kubernetes integration: scaling bioinformatics workflows in the cloud. *bioRxiv*, 2019.
- [33] J. A. Novella, P. E. Khoonsari, S. Herman, D. Whitenack, M. Capuccini, J. Burman, K. Kultima, and O. Spjuth. Container-based bioinformatics with Pachyderm. *Bioinformatics*, 35(5):839–846, 2019.
- [34] T. M. Oinn, R. M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. A. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. W. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.
- [35] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, and P. J. Maechling. Enabling large-scale scientific workflows on petascale resources using MPI master/worker. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the EXtreme to the Campus and Beyond*, XSEDE 12, New York, NY, USA, 2012. Association for Computing Machinery.
- [36] Topology and orchestration specification for cloud applications version 1.0. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>, 2013.
- [37] T. Stuart, A. Butler, P. Hoffman, C. Hafemeister, E. Papalexi, W. M. I. Mauck, Y. Hao, M. Stoeckius, P. Smibert, and R. Satija. Comprehensive integration of single-cell data. *Cell*, 177(7):1888–1902.e21, Jun 2019.
- [38] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
- [39] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.
- [40] WFMC. *Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011, Document Status – Issue 3.0*. Workflow Management Coalition, Brussels, Feb. 1999.
- [41] J. M. Wozniak, M. Wilde, and I. T. Foster. Language features for scalable distributed-memory dataflow computing. In *Proceedings of the 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM 14, page 5053, USA, 2014. IEEE Computer Society.

- [42] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, pages 44–60, 2003.
- [43] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.
- [44] G. X. Y. Zheng, J. M. Terry, P. Belgrader, P. Ryzkin, Z. W. Bent, R. Wilson, S. B. Zivaldo, T. D. Wheeler, G. P. McDermott, J. Zhu, M. T. Gregory, J. Shuga, L. Montesclaros, J. G. Underwood, D. A. Masquelier, S. Y. Nishimura, M. Schnall-Levin, P. W. Wyatt, C. M. Hindson, R. Bharadwaj, A. Wong, K. D. Ness, L. W. Beppu, H. J. Deeg, C. McFarland, K. R. Loeb, W. J. Valente, N. G. Ericson, E. A. Stevens, J. P. Radich, T. S. Mikkelsen, B. J. Hindson, and J. H. Bielas. Massively parallel digital transcriptional profiling of single cells. *Nature communications*, 8:14049–14049, Jan. 2017.