

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

StreamFlow: cross-breeding cloud with HPC

This is a pre print version of the following article:

Original Citation:

Availability:

This version is available <http://hdl.handle.net/2318/1756840> since 2023-12-28T15:38:15Z

Published version:

DOI:10.1109/TETC.2020.3019202

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

StreamFlow: cross-breeding cloud with HPC

Iacopo Colonnelli¹, Barbara Cantalupo¹, Ivan Merelli², and Marco Aldinucci¹

¹*Department of Computer Science, University of Torino, Italy*

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

This paper is the accepted version of IEEE copyrighted material*

I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, Streamflow: cross-breeding cloud with HPC, IEEE Transactions on Emerging Topics in Computing, 2020.

DOI:10.1109/TETC.2020.3019202.

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.

*© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

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 [1]. 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” cross-site platform. 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

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. The Workflow Management System (WMS) 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. Several surveys exist on WMSs, comparing their different functionalities [2, 3], focusing on their evolution [4] or providing classification with respect to the support for extreme-scale applications [5]. In the context of this work, we are particularly interested

in understanding the most critical needs, the most effective approaches and the most promising developments in this continuously changing technological domain.

In particular, two main levels of analysis should 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 by the workflow engine. At the first level, it is essential to evaluate the ability of the system to respond to user needs, by supporting potentially complex multi-node execution environments, and manage massive amounts of data ingested and computed by all the applications. At the infrastructure level, together with established architectures like clusters or grids, the cloud is now the most referred infrastructure for application execution and new paradigms are gaining attention like containers and orchestrators. Finally, HPC facilities are getting more and more importance outside the research centre even if there is no straight road-map for their integration with other platforms till now.

2.1 Scientific workflows

WMSs for scientific workflows are user-driven systems specifically developed to satisfy domain requirements. They provide researchers with a useful paradigm to describe, manage and share complex scientific analyses, also ensuring reproducibility and scalability properties. Experiments can be modelled by using a high-level declarative language or advanced graphical interfaces, suitable for researchers with little programming experience, or described programmatically.

Many scientific WMSs (e.g. Kepler¹ [6], Askalon² [7], Pegasus³ [8], Taverna⁴ [9] and Galaxy⁵ [10]) emerged with the diffusion of the web services and grid technologies, which offered the possibility to access robust services and infrastructures in a more natural way than before [11]. 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 started providing some additional features, as workflow repositories or support for diverse newer architectures, establishing consensus among researchers.

Even if some of these tools like Pegasus and Askalon offer support for automatic data transfers also in the absence of a commonly shared file-system among worker nodes, they rely on specific transfer protocols (e.g. GridFTP, SRM or Amazon S3) or delegate it to an external batch scheduler such as HTCondor, actually constraining the set of supported configurations. Moreover, although both Galaxy and Pegasus offer support for container execution, they only allow mapping a task into a single container. Asterism [12], a hybrid framework where the stream-based workflow execution is managed by dispel4py [13] and the data movement among workers is left to Pegasus, represents an interesting exception. Indeed, it relies on Docker Compose⁶ to set up complex execution

¹<https://kepler-project.org/>

²<http://www.askalon.org/>

³<https://pegasus.isi.edu/>

⁴<https://taverna.incubator.apache.org/>

⁵<https://galaxyproject.org/learn/advanced-workflow/>

⁶<https://docs.docker.com/compose/>

clusters. Nevertheless, at the time of writing, dispel4py only provides MPI and Apache Storm⁷ executors, strongly limiting the potential of the library.

An alternative approach to complex and feature-rich WMSs privileges performances over accessibility, exposing lower-level programming models directly to the user and allowing for the execution of a large number of interconnected tasks on distributed architectures. Among these frameworks are HyperLoom [14] and Dask⁸ [15], where pipelines of tasks can be defined with a Python interface, Spotify’s Luigi⁹, which also comes with a visual interface for monitoring purposes, and COMP Superscalar (COMPSs) [16], that allow users to parallelise existing sequential applications by identifying and annotating functions that can be executed as asynchronous parallel tasks. Despite being very efficient in terms of performances, these libraries are hard to use for domain experts without programming experience.

Another approach tries to lower the level of complexity by implementing a simplified Domain Specific Language (DSL) to describe workflows. For example, in Apache Airflow¹⁰ and Snakemake¹¹ [17] workflows are essentially Python scripts extended by declarative code that can be executed on distributed infrastructures. Other systems adopt Unix-style approaches for defining workflows: in Makeflow¹² [18] the end-user expresses a workflow in a technology-neutral way using a syntax similar to Make, while the Nextflow¹³ [19] bioinformatics framework builds workflows on Unix pipe concept. In these frameworks, a set of pluggable executors allows workflows to be deployed and run on different infrastructures, including public cloud services, batch schedulers (e.g. HTCondor, PBS, SLURM) and Kubernetes clusters. Nevertheless, different steps of the same workflow cannot be managed by different executors, not guaranteeing support for hybrid cloud/HPC configurations. Moreover, even if containers executions are permitted, it is not possible to specify complex multi-container environments to execute a single task.

Since product-specific DSLs tightly couple workflow descriptions to a single software, actually limiting portability and reusability, there are also efforts in defining workflow specification languages or standards. For instance, The Common Workflow Language (CWL)¹⁴ [20] is an open standard for describing analysis workflows following a JSON or YAML syntax or a mixture of the two. One of the first and most used CWL implementations is CWL-Airflow [21], which adds support for CWL to Apache Airflow, but also other products (e.g. Snakemake and Nextflow) offer some compatibility with CWL. Another interesting dataflow language for scientific computing is Swift¹⁵ [22], in which all statements are eligible to run concurrently, limited only by the data flow.

⁷<http://storm.apache.org/>

⁸<https://docs.dask.org/>

⁹<https://github.com/spotify/luigi/>

¹⁰<https://airflow.apache.org/>

¹¹<https://snakemake.readthedocs.io/en/stable/>

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

¹³<https://www.nextflow.io/>

¹⁴<https://www.commonwl.org>

¹⁵<http://swift-lang.org/main/>

2.2 Cloud orchestration

With the rise of cloud computing and the related *-as-a-Service approaches, users from both academia and industry started to move computation from their machines to the cloud. Nevertheless, all the scalability, portability and availability benefits brought by cloud technology necessarily come with increased complexity in deploying, configuring and managing applications, especially in hybrid-cloud scenarios.

Several tools have been developed with the precise aim to seamlessly support hybrid-cloud deployments of complex applications, composed of heterogeneous intercommunicating services, by exposing to the user just an agnostic and straightforward interface, usually in the form of a DSL. This approach is often referred to as *Infrastructure-as-Code* (IaC).

Early solutions [23, 24] focused their attention on the deployment phase, with minimal support for contextualisation (a simple run-once script launched right after the virtual machine creation) and no support at all for automatic orchestration of active applications. In this setting, both the creation and maintenance of ad-hoc base images for each required software application and all the post-deployment operations were left to the IT experts.

More recently, some more advanced tools have been proposed, addressing different aspects of application deployment and orchestration. Caballer et al. [25] focus on reusability, introducing a platform in which every configuration element, from infrastructure descriptions to virtual machine images, can be stored in a dedicated registry. SALSA [26] comes with a fine-grained, multi-layered dependency structure, in order to handle the configuration of both virtual machines and applications deployed upon them. Roboconf [27] proposes a hierarchical DSL, capable of defining both containment and runtime relations, and offers orchestration primitives to automatically manage the reconfiguration of live systems in response to events, e.g. an increased workload or a failure. Occopus [28] privileges compatibility and extendibility, proposing a pluggable architecture in which combinations of replaceable plugins manage interactions with external tools and services (as well as some core features and behaviours). Moreover, it comes with some orchestration features, e.g. health-checking, auto-scaling and garbage collection.

Workflow management and cloud orchestration technologies can benefit from each other. For example, the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)¹⁶ [29] focuses on complex dependency management, using workflow description languages to write the deployment and management plans for a cloud environment. Implementations of the TOSCA standard are currently provided by Cloudify¹⁷, an orchestration platform based on event-driven workflows, and Yorc¹⁸ (Ystia orchestrator), an hybrid cloud/HPC orchestrator developed in the Lexis project [30]. Moreover, a Cloudify plugin to orchestrate batch applications in HPC and cloud environments has been developed in the Croupier¹⁹ project.

With the advent of containerisation as a lightweight alternative to virtualisation, some container orchestrators started to flourish. Among them, Kubernetes

¹⁶<https://www.oasis-open.org/>

¹⁷<https://cloudify.co/>

¹⁸<https://github.com/ystia/yorc>

¹⁹<https://github.com/ari-apc-lab/croupier>

has become the de-facto standard for container orchestration during the last years, and the vast majority of cloud providers include a managed Kubernetes service in their offering. Kubernetes comes with a very flexible YAML-based DSL, able to describe both deployment and runtime orchestration features for multi-container applications. Containers are also gaining popularity in the scientific domain, and several workflow frameworks have been built natively on top of Kubernetes like Pachyderm²⁰ [31], Argo²¹ and a specific Galaxy installation developed in the PhenoMeNa²² project [32]. Moreover, all the leading cloud vendors are currently focusing on offering hybrid solutions that allow combining multi-cloud and on-premises infrastructures. The most integrated framework is GoogleCloudComposer²³, a fully managed workflow orchestration service built on Apache Airflow. Despite offering great flexibility in interacting with Kubernetes resources, these products are tightly coupled with such technology and do not allow for task offloading on different environments, such as HPC sites.

Even if both WMSs and orchestrators must be able to deal with dependencies among different tasks, they differ in their primary goals. Indeed, from one side, WMSs must focus on efficient ephemeral executions of tasks, minimisation of the distributed execution overheads (e.g. through data-locality-based scheduling policies) and should be easy enough to be used by domain experts. From the other one, orchestrators must ensure availability and responsiveness of long-lived systems, portability among different infrastructures and enough flexibility to satisfy the needs of IT experts. With this in mind, StreamFlow aims to offer an easy way to allow the automatic execution of workflows on top of complex and orchestrated environments while keeping the two aspects distinct enough to be easily handled by different kinds of users.

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[33] and Singularity[34], 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 [35].

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

²⁰<http://pachyderm.io/>

²¹<https://argoproj.github.io/argo/>

²²<http://phenomenal-h2020.eu/home/>

²³<https://cloud.google.com/composer>

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 [36], 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-system is commonly ephemeral, every task execution runs inside a clean and consistent environment (with the apparent exception of eventual temporary files saved into persistent folders). For its part, an MTSC execution can provide some performance improvements in those cases when the task execution is high-speed (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 or when some data dependencies are stored in the ephemeral file-system, in order to avoid additional data transfers when recreating containers.

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 setting of *Multiple-Tasks Multiple-Containers* (MTMC) 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 [5]. 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 an *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 and Singularity Compose²⁴ to complex orchestrators as Kubernetes²⁵ or Docker Swarm²⁶. 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 to split a single task 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.

The following two requirements can summarise all these considerations:

²⁴<https://singularityhub.github.io/singularity-compose/>

²⁵<https://kubernetes.io/>

²⁶<https://docs.docker.com/engine/swarm/>

- 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.

3.2 Hybrid workflows

When considering data-intensive scientific workflows, all those aspects related to data management (such as data locality, data access, and data transfers) 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 [5], 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 a mandatory requirement when dealing with federated data access or strict privacy policies.

Even if many of the existing WMSs can 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 every inter-container data transfer needs at least two copy operations: a first one from the source to the management node and a second one to the destination. Sometimes this is 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

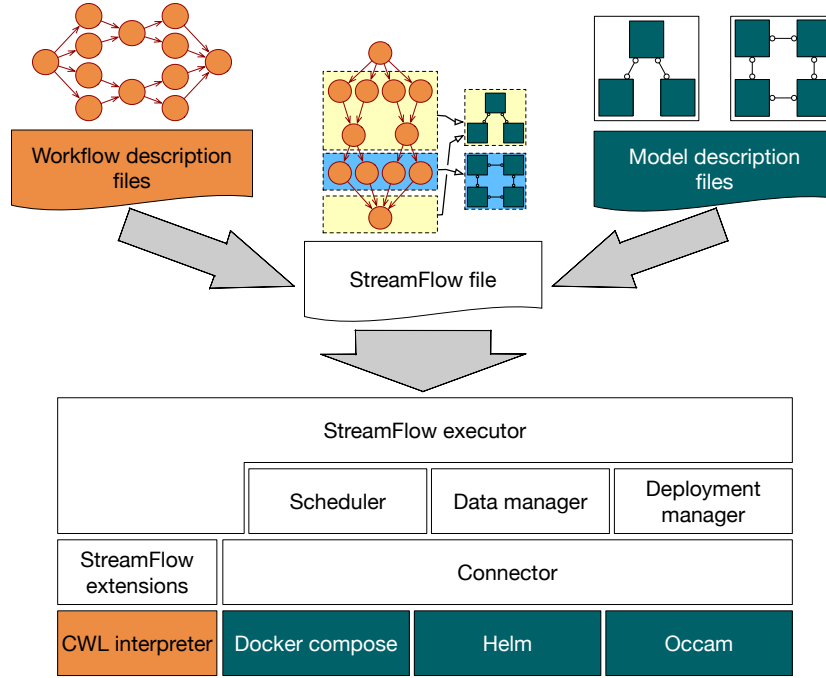


Figure 1: StreamFlow framework’s logical stack. Coloured portions refer to existing technologies, while white ones are directly part of StreamFlow code-base. In particular, the orange area is related to the definition of the workflow’s dependency graph, while the green area refers to the execution environments.

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.

Again, the following two requirements can summarise the previous discussion:

R3 If the WMS management node can 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 framework²⁷ 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, but rather to build a common interface to allow for the integration with existing technologies.

In StreamFlow's glossary, a complex multi-container environment is called *model*. Each model is managed independently of the others by a dedicated **Connector** implementation, which acts as a proxy for the underlying orchestration library. A single model can include multiple types of containers, called *services*. For example, a Docker Compose file describing a database and a Tomcat container linked together constitutes a model with two services. The `streamflow.yml` file, the actual entry point for a StreamFlow execution, contains pointers to workflows and models descriptions and specifies the way they should relate to each other, i.e. the service that should execute each workflow *step*. Since multiple replicas of the same service could coexist in a given model, each service can refer to one or more containers, called *resources*.

Before actually executing a task, it is necessary to deploy the related model successfully. The **DeploymentManager** class has precisely the role of creating models when needed and destroying them as soon as they become useless. Then the **Scheduler** class is in charge to select the best resource on which each task should be executed while guaranteeing that all requirements are satisfied. Finally, the **DataManager** class, which knows where each task's input and output data reside, must ensure that each service can access to all the data dependencies required to complete the assigned task, performing data transfers only when necessary. At this point, a *job* (i.e. the runtime representation of a task) can be successfully executed on the selected resource.

The rest of the current section is devoted to analysing with more detail each of the components mentioned above, 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. In order to realise a first proof-of-concept, we decided to integrate with the CWL format. 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. Last but not least, the CWL's reference implementation, called `cwltool`²⁸, 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.

²⁷<https://streamflow.di.unito.it/>

²⁸<https://github.com/common-workflow-language/cwltool>

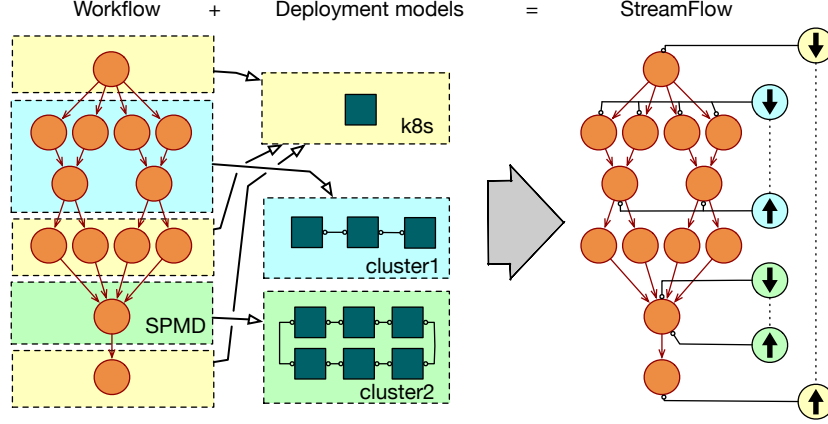


Figure 2: Workflow graph transformation to include model deployment and undeployment tasks. Orange nodes represent original tasks, while the others refer to model deployment (downward pointing arrow) and undeployment (upward pointing arrow) phases.

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. This approach considerably reduced the development time, but the risk is to introduce excessively tight coupling between the CWL-specific features and the more generic StreamFlow logic. Since we plan to support other coordination languages in the future, a more agnostic mid-layer representation of a workflow graph is definitely on the todo list.

4.2 Model life-cycle management

In StreamFlow, the service allocation and the subsequent task execution happen in two strictly distinct phases, leaving the containers' life-cycle management to an external orchestration library. A clear advantage of this approach lies in the possibility to rely on all the orchestration features provided by a mature product (e.g. autoscaling, restarting policies, affinity-based scheduling) and to adopt the original deployment description language, sparing users the extra effort needed to learn a new syntax. 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.

As shown in Fig. 2, from a theoretical point of view, this approach 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 *deployment* 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 a model must depend on it.
- The *undeployment* task, which destroys an existing model. No other task

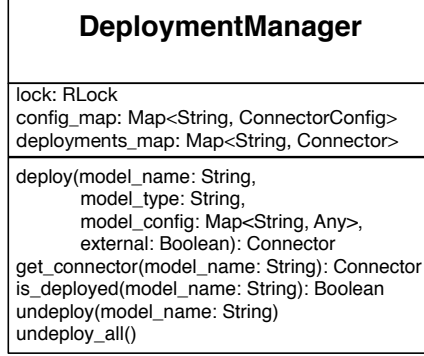


Figure 3: UML class diagram for the `DeploymentManager` class.

depends on it, but it should depend on all the tasks that must be executed on such a model, in order to wait for their termination before starting the undeployment 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. Nevertheless, since deployment tasks have no dependency, a standard scheduler will try to execute them as soon as possible, according to an *eager* resource allocation strategy. In this setting, some models can be up and running long before they are needed, leading to a potential waste in terms of energy consumption and money. In such case, a far more practical approach would be to let a model be deployed by the first fireable task which requires it, according to a *lazy* resource allocation strategy.

The `DeploymentManager` class, whose UML diagram is represented in Fig. 3, has precisely the role of implementing these allocation strategies, relying on the underlying orchestration library through a pluggable implementation of the `Connector` interface. In particular, the `deploy` method atomically checks if a model has been already deployed and, if not, it puts a new `Connector` instance into the `deployments_map` and invokes its `deploy` method. Since the requirement *R2* states that a single instance of a model can be used to execute multiple tasks, the `lock` is necessary to avoid race conditions when concurrent tasks require the same model. Finally, the `external` attribute allows `StreamFlow` to interact with an externally managed model, relieving the `DeploymentManager` of deployment and undeployment duties.

Ideally, a model should be undeployed as soon as the last task needing it has been completed. This logic is quite easy to implement when dealing with static DAGs, but things get more complicated in the dynamic setting. Probably the best strategy for the second case would be to set a grace period, after which the model is undeployed if no new task required it. For now, the `DeploymentManager` confines itself to undeploy all the models at the end of the entire workflow execution, calling the `undeploy_all` method. Moreover, the same method is also invoked by `StreamFlow`'s main process in case of unrecoverable failures. This approach is very straightforward, but it can lead to resource wastes if some models remain unused for a long time.

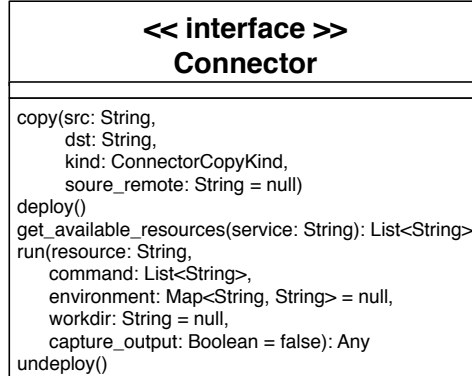


Figure 4: UML class diagram for the **Connector** interface.

As discussed before, StreamFlow interacts with each underlying orchestration technology by means of a common **Connector** interface, whose UML diagram is shown in Fig. 4. This adheres to the separation of concerns principle, providing an easy way to add support for additional products if required. The **Connector** interface is a low-level block in the StreamFlow’s logical stack, which is used by all the higher-level components. Besides the **deploy** and **undeploy** methods, which are called by the **DeploymentManager** class, the **get_available_resources** method is invoked by the **Scheduler** class to obtain all the replicas of a given service in the model, while 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. Finally, the **run** method is used to execute a command on top of a remote resource and potentially to capture the generated output value. For now, three different **Connector** implementations come out-of-the-box with StreamFlow, supporting Docker Compose, Helm²⁹ and Occam, the supercomputing centre of Università di Torino [37].

4.3 The StreamFlow file

When launching a StreamFlow execution, the only argument it takes is the path of a YAML file, conventionally called **streamflow.yml**. The crucial role of such file is to link each task in a workflow with the service that should execute it. Moreover, in order to ensure this binding is unambiguous, each service in a model and each task in a workflow should be uniquely identifiable. This section describes the StreamFlow file syntax and the strategies adopted to guarantee such unambiguity.

A valid StreamFlow file contains the version number (which currently only accepts the **v1.0** value) and two main sections: **workflows** and **models**. The **workflows** section consists of a dictionary with uniquely named workflows to be executed in the current run. Each workflow specification is an object containing three fields. The **type** field identifies which language has been used to describe the dependency graph (at the moment **cwl** is the only accepted value), while the

²⁹<https://helm.sh/>

`config` field includes the paths to the files containing such description. Finally, the `bindings` list contains the task-model associations. Different workflows are independent of each other, in that an entire StreamFlow logical stack is allocated for each of them. It means that, even if two tasks in two different workflows can refer to the same model specification, two different environments will be deployed for their execution.

Considering workflows as dependency graphs, each node 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, and 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 and unambiguous 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. Fortunately, CWL standard (and also the vast majority of coordination languages on the market) satisfies both these requirements.

The `models` section contains a dictionary of uniquely named model specifications, each of which is an object with two distinct fields. The `type` field identifies which `Connector` implementation should be used for its creation, destruction and management, while the `config` field contains a dictionary with configuration parameters for the corresponding `Connector`. Usually, the `config` parameters are directly extracted from the tools commonly used to interact with the underlying orchestration library (e.g. the `docker-compose` CLI for Docker Compose or the `helm` CLI for Helm charts), so that a user who is familiar with these libraries can easily understand the StreamFlow format.

The best way to unambiguously identify services in a model 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 identify the related service uniquely. Moreover, since an Occam description file is practically 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 explicitly required to fill in the `name` attribute of each container in the Pod template with a unique identifier.

The format adopted for the `bindings` list takes into account all the previously discussed considerations on unambiguous identification of tasks and services. In particular, each element of such list contains a `target` object, with a `model` and a `service` attributes that uniquely identify a service, and a `step` attribute containing a path in the aforementioned file-system abstraction of a workflow graph. If the path resolves to a folder (i.e. to a nested sub-workflow), 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. For the interested reader, 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.

4.4 Task scheduling

The task scheduling strategy is a fundamental component of a WMS, mainly for the large 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 using optional parameters in the coordination language, while the actual mapping on top of adequate worker nodes is left to the implementation of the specific executor.

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, network and persistence. Nevertheless, in a real scenario, different tasks likely require very diverse amounts of 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, 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. Requirement *R1*, which states that in StreamFlow, the unit of deployment should be a complex environment with different containers, introduces an additional level of complexity here. Indeed, it is no longer true that a task can be executed on any worker node equipped with enough hardware, but rather 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 straightforwardly manages this requirement-capability association, by identifying each container type with a single service, according to requirement *R2*, and specifying which service is required by each task (through the `bindings` list described in Sec. 4.3). Since in StreamFlow the model life-cycle is managed by an external orchestration library, container-related resources constraints should be specified in the environment description file. 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. 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 `get_available_resources` method of the appropriate `Connector` instance) and to apply a scheduling policy to find the best target.

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 (e.g. computing power, data locality, load balancing) 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. 5, the `Policy` interface only contains a single method, called `get_resource`, with five input arguments:

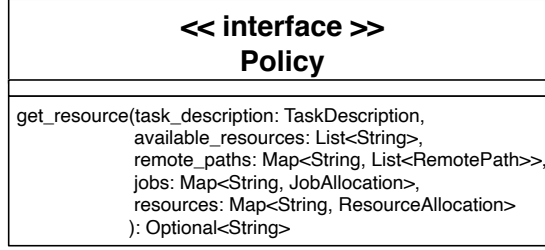


Figure 5: UML class diagram for the Policy interface.

- The **task_description** argument contains a characterisation of the current task in terms of resource requirements and data dependencies.
- 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. This information can be used by a scheduling policy to take into account data locality in its algorithm.
- The last two arguments describe the previously allocated jobs, allowing the implementation of load-balancing features in the scheduling strategy. In particular, the **JobAllocation** class contains the task description, the resource to which it has been assigned and its status, while the **ResourceAllocation** class contains the related model and service of an existing resource and the list of jobs assigned to it.

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

A very general scheduling policy, serving as a default strategy, comes out-of-the-box with **StreamFlow**. When a task becomes fireable, the algorithm iterates over all available resources, starting from those containing at least one of its data dependencies to privilege data locality and trying to reserve the first one which is free (i.e. does not contain jobs in the **running** status) and satisfies all the constraints. If the search fails, then a **null** value is returned, and the task is inserted into a waiting queue: a new scheduling attempt will be performed as soon as a **running** job notifies its termination.

4.5 Data transfers

As pointed out in Sec. 3.2, hybrid workflow executions make it necessary to waive the comfort brought by a globally shared data space, leaving to the WMS the task of explicitly moving the data whenever required. Since large data

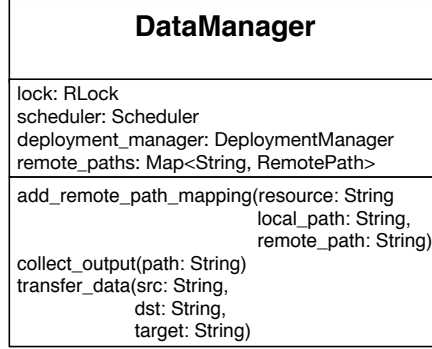


Figure 6: UML class diagram for the **DataManager** class.

transfers are very time-consuming operations, especially for long distances and in the absence of dedicated high-throughput communication networks, the WMS should always select the best communication channel between two endpoints and avoid all unnecessary data movements. The StreamFlow framework has been designed in order to meet requirements *R3* and *R4*, which represent two fundamental steps in this direction. In particular, a dedicated **DataManager** class, whose UML diagram is in Fig. 6, 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.

Whenever a task terminates in **completed** status, it is in charge of atomically updating the **remote_paths** structure with the remote position of all its output files and folders by calling the **add_remote_path_mapping** method. The same structure is also used by the **transfer_data** method, which is called every time a task needs a file or a folder from one of its predecessors, to verify if a data transfer is needed or not. In particular, transfers can always be avoided when both tasks run on the same resource, but this can also happen when two resources share a data space (e.g. a persistent volume) or if a task explicitly performs a data transfer before completing.

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.

4.6 Task-container mapping patterns

Since models are not redeployed after each task execution, when multiple tasks are bound to the same service StreamFlow implements by default an MTSC pattern. This design choice is the standard one adopted by CWL when tasks are executed on the local machine and is consistent with *R4*, which tries to minimise data transfers to achieve better performances. Nevertheless, it gives up for the clean and consistent execution environment commonly provided by containers, which can be problematic if previous task executions can have unexpected effects on the next ones. In order to force an STSC pattern, a `recycle` directive can be added to a binding entry in the StreamFlow file. This induces a redeployment of the involved service before the task execution. In such a case, StreamFlow will automatically handle all required data transfers, ensuring that at least one copy of each task output is stored in a persistent location before deleting the container.

The unique parallel execution pattern natively included in CWL standard is the `scatter`, in which a list of input data is partitioned among multiple, identical tasks that can be executed in parallel by multiple nodes. More complex interactions among tasks (e.g. an MPI application) must be directly handled in the code, and it is up to the user to ensure that the correct amount of worker nodes is up and running before the task execution. Conversely, StreamFlow offers explicit support for STMC mapping: in the StreamFlow file, a single step can be bound to multiple resources through the `replicas` directive (which defaults to 1). In case of multiple replicas, StreamFlow initialises two additional environment variables: a `STREAMFLOW_RANK` variable, containing a unique rank for each job, and a `STREAMFLOW_HOSTS` variable, containing the comma-separated list of nodes (i.e. hostnames) allocated for the task. These variables can be used inside the task script to guide the execution on each node, e.g. to implement a master-worker pattern or to execute an `mpirun` command on the node with rank 0.

The case of MTMC pattern is a bit trickier. Indeed, while StreamFlow seamlessly supports co-allocation of different tasks on different services, the CWL standard is not able to explicitly describe such property. Although the simultaneous allocation of apparently independent steps just happens in many situations, it is evident that a formal and generic way to express tasks co-allocation directly in the workflow model is a mandatory requirement to entirely support this feature. Find the best way to improve MTMC pattern support is an essential milestone in StreamFlow’s future development.

5 Single-Cell application use-case

As extensively described in Sec. 2.1, scientific applications are an ideal target for workflow modelling. To demonstrate StreamFlow’s ability to satisfy requirements at both user-level, in terms of supporting easy application modelling, and infrastructure-level, offering flexibility in the choice of deployment targets, we selected a novel pipeline in the field of Bioinformatics, which is the discipline supporting molecular biology and biomedicine in the analysis of data. Bioinformatics, together with astronomy, is one of the first scientific fields to deal with Big Data. Indeed, biological datasets are massive, heterogeneous, and grow

very fast, making this discipline hungry for computational power and storage capabilities.

Moreover, these data should be analysed by Bioinformatics researchers, with a mixed background in biology, medicine and computer science. This scenario requires the development of hybrid computational systems by HPC experts, implying a careful selection of the target infrastructure according to the different applications, desired performance, but also cost requirements.

From this perspective, in this section, we want to show how StreamFlow features can be used to implement complex analysis workflows in an extremely portable way, which allows users to find the best deployment option for each step in heterogeneous, hybrid HPC/cloud infrastructures without modifying neither the code nor the workflow description itself.

5.1 Single-cell sequencing

More specifically, the Bioinformatic application we selected for our tests is a pipeline for single-cell sequencing data analysis. Generally stated, sequencing is the process of determining the order of the four bases (adenine, guanine, cytosine, and thymine/uracil) of a nucleic acid molecule, which can be DNA or RNA. The first nucleic sequences were obtained in the early 1970s by academic researchers using laborious methods based on two-dimensional chromatography. Following the development of fluorescence-based sequencing methods, sequencing has become more accessible and orders of magnitude faster, allowing the first draft of the human genome. During the last decade, massive high-throughput sequencing methods have revolutionised the entire field of molecular biology, both considering DNA sequencing and RNA sequencing, accelerating medical research and discovery, since samples from patients can now be sequenced routinely.

DNA sequencing is usually performed to describe the genomic differences between two samples, which impact on their phenotypes, such as having different eyes' colour or susceptibility to a specific disease. On the other hand, RNA sequencing (RNA-seq) is performed to understand what is going on inside the cell, which genes are actually transcribed and active, because, for example, a liver must implement different biological processes in comparison to a spleen (although they share precisely the same genome). The idea is that we can compare DNA to a program stored on a disk and RNA as the same program loaded into RAM.

The opportunity to study the transcriptome of cells (cultured cells, cells from mouse models or cells from human samples, such as blood) using RNA-seq has fuelled many crucial discoveries in biology and biomedicine, being now a routine method in clinical research. However, RNA-seq is typically performed in "bulk", which means to sequence the RNA of all the cells in a sample (thousands or millions of cells), and the data represent an average gene expression pattern across a population of cells. This might obscure biologically relevant differences between cells, such as tumour clones that are resilient to chemotherapy.

Single-cell RNA-seq (scRNA-seq) represents an approach to overcome this problem. The idea is to isolate single cells through microfluidic approaches, capturing their transcripts through emulsion droplets loaded with chemical reagents, and generating sequencing libraries in which the transcripts are tagged (through a nucleotidic barcode) to track their cell of origin. One of the most

popular platforms for single-cell analysis is marketed by 10X Genomics, which is capable of analysing from 500 to 20,000 cells in each run. Then, combined with massive high-throughput sequencing producing billions of reads, scRNA-seq allows the assessment of fundamental biological properties of cells populations and biological systems at unprecedented resolution.

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 each cell. Denoising these data and estimating the adequate amount of sequencing reads covering each gene in the cell is of critical importance to define a reliable RNA *count matrix*, the fundamental data structure for this kind of analysis which represents for each cell and each gene how many transcripts have been captured. This is a quite complex bioinformatic pipeline that requires many different 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 pipeline are of critical importance, since analysing each cell in a population requires from hundreds-of-thousands to millions of comparisons to be processed in a high throughput manner.

5.2 Application pipeline

Once the noise caused by the experimental amplification of the RNA has been controlled, and the count matrix has been built, the key idea is to implement techniques aimed at reducing the data dimensionality in order to cluster cells with a similar expression profile. Therefore, a typical pipeline for single-cell transcriptomic data analysis can be broadly divided into two main parts: the creation of the count matrix and its statistical analysis.

The first step is the creation of the RNA count matrix, and it must be performed according to the adopted single-cell experimental technology and the used 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 [38]. In particular, this part of the analysis will consist in two steps: the creation of the fastq files (the raw sequences of the four bases, called reads) 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.

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. These alignments are then processed according to the genome annotation, in order to recapitulate for each gene how many reads 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 transcrip-

tomic profiles and characterising 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 [39, 40] for normalisation, dimensionality reduction and clustering of cells, and SingleR [41] for labelling the clusters, that is identifying the cell type, according to public databases of single-cell data annotation.

In particular, Seurat is used to loading data into the R environment and to filter outliers for specific statistics, such as the number of unique transcripts or the presence of mitochondrial transcripts, which correlate with the vitality of the cells. Data are then normalised, taking into account the different coverage of 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. At last, marker genes are identified for each cluster by comparing the expression profile of the cells inside the cluster with all the other cells.

Once clusters have been identified, the pipeline uses SingleR to characterise each cell trying to identify its type (such as Blood Cell, Bone Cell, and Stem Cell) in an unbiased way. SingleR leverages reference transcriptomic datasets of pure cell types to infer the identity of every single cell independently. In particular, SingleR starts by calculating a Spearman coefficient for each cell in the single-cell experiment with the reference 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 only the variable genes among these remaining cell types, until a precise cell type can be assigned to the analysed cell.

As a test case, in this work, we used a published dataset [42] concerning Gene Editing in Hematopoietic Stem Cell. In particular, this dataset was produced to compare the efficiency of different gene-editing approaches and, for this reason, the whole experiment is composed of 6 different single-cell samples sequenced independently. This complex experimental design resulted in a particularly challenging and time-consuming dataset, making a flexible, automated and scalable workflow management systems particularly desirable.

5.3 StreamFlow implementation

In general, 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 services each. Here we decided to experiment two different combinations of Occam and Helm environments, as better detailed below in this section.

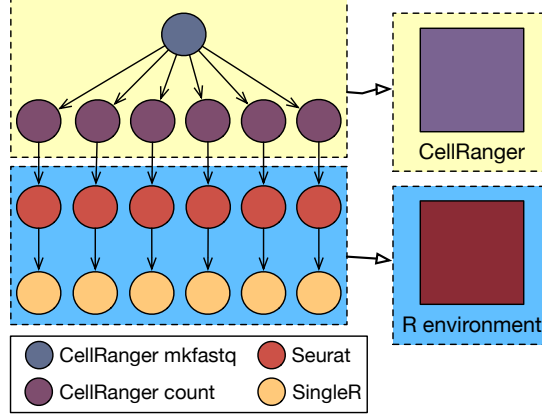


Figure 7: Dependency graph and model bindings for the single-cell workflow. In this case, the first step creates six different sequences, which can then be processed independently of each other for the remaining three steps.

- The creation of a StreamFlow file, as described in Sec. 4.3, in order to wrap things together.

Fig. 7 provides a graphical representation of the whole StreamFlow model for a single-cell pipeline of the kind described in Sec. 5.2. In this case, the workflow dependency graph is a simple DAG with four different kinds of tasks. In terms of workflow patterns [43], 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. In CWL, this can be easily implemented using the `scatter` directive. 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.

Since CellRanger executes the first two types of tasks and the last two tasks require two main R packages (i.e. Seurat and SingleR) plus all the related dependencies, we decided to implement two distinct container images. 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, implementing an MTSC mapping as described in Sec. 4.6. Nevertheless, since the containers initialisation time is negligible with respect to the time required for the completion of tasks themselves and outputs are always stored in a persistent location, this ends up being practically equivalent to an STSC pattern. Given that, it should be clear that requirements *R1* and *R2* of Sec. 3.1 do not bring additional concrete value to this workflow.

Conversely, the hybrid workflow execution enabled by requirements *R3* and *R4* of Sec. 3.2 can be beneficial, 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 initial

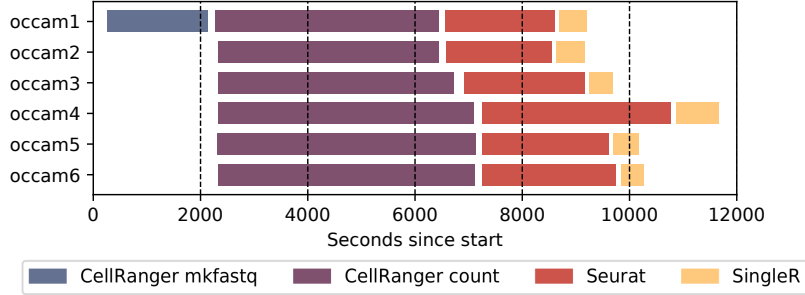


Figure 8: Execution timeline for the StreamFlow single-cell application on six Occam nodes, each allocated to both a CellRanger and an R environment containers.

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 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. 4.5, 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. 4.5). 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 services. 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 GARR³⁰ cloud, based on OpenStack³¹, 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.

It is worth noting that there is no need to modify the CWL description of the workflow to run it on the new environment: changes only involve model descriptions and the `streamflow.yml` file.

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. It 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 allocation 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

³⁰<https://garr.it/it/>

³¹<https://www.openstack.org/>

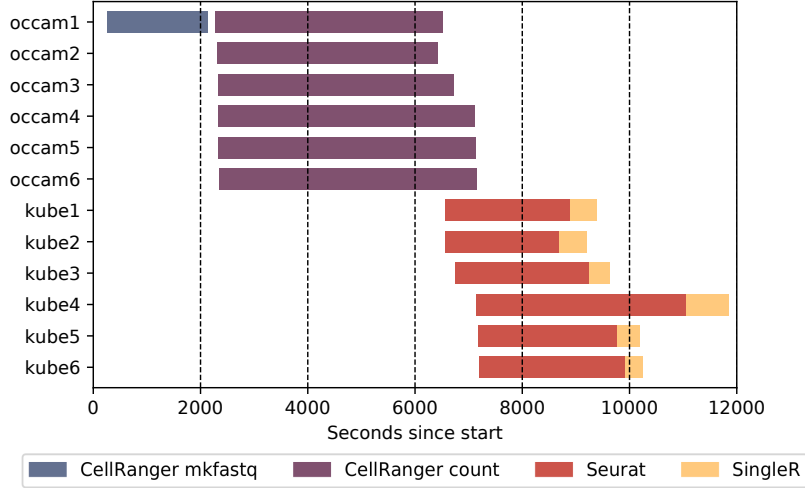


Figure 9: Execution timeline for the StreamFlow single-cell application in a hybrid configuration, with six Occam nodes allocated to CellRanger as many replicas and six Kubernetes worker nodes allocated to as many R environment containers.

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 exploit the potential of containerisation technologies fully.

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 data space 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.

Acknowledgement



This article describes work undertaken in the context of the Deep-Health project³², “*Deep-Learning and HPC to Boost Biomedical Applications for Health*” which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 825111 [44]. This work has been partially supported by the HPC4AI project³³ [45].

References

- [1] 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*, ser. LNCS, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 678–690.
- [2] 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.*, vol. 75, pp. 284–298, 2017.
- [3] J. Liu, E. Pacitti, and V. P. et al, “A survey of data-intensive scientific workflow management,” *Journal of Grid Computing*, vol. 13, no. 4, pp. pp 457–493, Dec. 2015.
- [4] M. P. Atkinson, S. Gising, J. Montagnat, and I. J. Taylor, “Scientific workflows: Past, present and future,” *Future Generation Comp. Syst.*, vol. 75, pp. 216–227, 2017.
- [5] 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.*, vol. 75, pp. 228–238, 2017.
- [6] 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*, vol. 18, no. 10, pp. 1039–1065, 2006.

³²<https://deephealth-project.eu/>

³³<http://www.hpc4ai.it>

- [7] 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*, 2007, pp. 450–471.
- [8] 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.*, vol. 46, pp. 17–35, 2015.
- [9] 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*, vol. 18, no. 10, pp. 1067–1100, 2006.
- [10] 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*, vol. 44, no. Webserver-Issue, pp. W3–W10, 2016.
- [11] 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.*, vol. 4, no. 1, p. 2747, Mar. 2017.
- [12] R. Filgueira, R. F. d. Silva, A. Krause, E. Deelman, and M. Atkinson, “Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science,” in *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud)*, 2016, pp. 1–8.
- [13] R. Filguiera, A. Krause, M. Atkinson, I. Klampanos, and A. Moreno, “dispel4py: A python framework for data-intensive scientific computing,” *The International Journal of High Performance Computing Applications*, vol. 31, no. 4, pp. 316–334, 2017.
- [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*, 2018, pp. 1–6.
- [15] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: <https://dask.org>

- [16] F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia, “Enabling cloud interoperability with compss,” in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 16–27.
- [17] J. Köster and S. Rahmann, “Snakemake - a scalable bioinformatics workflow engine,” *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.
- [18] 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*, 2012, p. 1.
- [19] P. Di Tommaso, M. Chatzou, E. W. Floden *et al.*, “Nextflow enables reproducible computational workflows,” *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, Apr. 2017.
- [20] 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. [Online]. Available: <https://doi.org/10.6084/m9.figshare.3115156.v2>
- [21] M. Kotliar, A. V. Kartashov, and A. Barski, “CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language,” *GigaScience*, vol. 8, no. 7, 07 2019.
- [22] 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*, ser. DFM 14. USA: IEEE Computer Society, 2014, p. 5053.
- [23] T. C. Chieu, A. A. Karve, A. Mohindra, and A. Segal, “Simplifying solution deployment on a cloud through composite appliances,” in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, 2010, pp. 1–5.
- [24] A. Lenk, C. Dänschel, M. Klems, D. Bermbach, and T. Kurze, “Requirements for an iaas deployment language in federated clouds,” in *2011 IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2011, Irvine, CA, USA, December 12-14, 2011*, 2011, pp. 1–4.
- [25] M. Caballer, J. D. S. Quilis, G. Moltó, and I. Blanquer, “A platform to deploy customized scientific virtual infrastructures on the cloud,” *Concurr. Comput. Pract. Exp.*, vol. 27, no. 16, pp. 4318–4329, 2015.
- [26] D. Le, H. L. Truong, G. Copil, S. Nastic, and S. Dustdar, “SALSA: A framework for dynamic configuration of cloud services,” in *IEEE 6th International Conference on Cloud Computing Technology and Science, Cloud-Com 2014, Singapore, December 15-18, 2014*, 2014, pp. 146–153.

- [27] L. M. Pham, A. Tchana, D. Donsez, N. D. Palma, V. Zurczak, and P. Gibello, “Roboconf: A hybrid cloud orchestrator to deploy complex applications,” in *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*, 2015, pp. 365–372.
- [28] J. Kovács, P. Kacsuk, and M. Emodi, “Deploying docker swarm cluster on hybrid clouds using occopus,” *Adv. Eng. Softw.*, vol. 125, pp. 136–145, 2018.
- [29] “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.
- [30] A. Scionti, J. Martinovic, O. Terzo, E. Walter, M. Levrier, S. Hachinger, D. Magarielli, T. Goubier, S. Louise, A. Parodi, S. Murphy, C. D’Amico, S. Ciccica, E. Danovaro, M. Lagasio, F. Donnat, M. Golasowski, T. Quintino, J. Hawkes, T. Martinovic, L. Riha, K. Slaninova, S. Serra, and R. Peveri, “Hpc, cloud and big-data convergent architectures: The lexis approach,” in *Complex, Intelligent, and Software Intensive Systems*, L. Barolli, F. K. Hussain, and M. Ikeda, Eds. Cham: Springer International Publishing, 2020, pp. 200–212.
- [31] 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*, vol. 35, no. 5, pp. 839–846, 2019.
- [32] P. Moreno, L. Pireddu, P. Roger, N. Goonasekera, E. Afgan, M. van den Beek, S. He, A. Larsson, D. Schober, C. Ruttkies, 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. [Online]. Available: <https://www.biorxiv.org/content/early/2019/02/12/488643>
- [33] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014.
- [34] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017.
- [35] 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*, vol. 19, no. 10, p. 349, 2018.
- [36] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.
- [37] 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)*, vol. 898, no. 8, San Francisco, USA, 2017, p. 082039.

- [38] G. X. Y. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, 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*, vol. 8, pp. 14 049–14 049, Jan. 2017.
- [39] A. Butler, P. Hoffman, P. Smibert, E. Papalexi, and R. Satija, “Integrating single-cell transcriptomic data across different conditions, technologies, and species,” *Nature Biotechnology*, vol. 36, no. 5, pp. 411–420, 2018.
- [40] 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*, vol. 177, no. 7, pp. 1888–1902.e21, Jun 2019.
- [41] 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*, vol. 20, no. 2, pp. 163–172, 2019.
- [42] G. Schirotti, A. Conti, S. Ferrari, L. della Volpe, A. Jacob, L. Albano, S. Beretta, A. Calabria, V. Vavassori, P. Gasparini, E. Salataj, D. Ndiaye-Lobry, C. Brombin, J. Chaumeil, E. Montini, I. Merelli, P. Genovese, L. Naldini, and R. Di Micco, “Precise gene editing preserves hematopoietic stem cell function following transient p53-mediated dna damage response,” *Cell Stem Cell*, vol. 24, no. 4, pp. 551–565.e8, Apr 2019.
- [43] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, “Workflow patterns,” *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003.
- [44] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2019, pp. 150–155.
- [45] 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.