Geographically distributed Batch System as a Service: the INDIGO-DataCloud approach exploiting HTCondor

Geographically distributed Batch System as a Service: the INDIGO-DataCloud approach exploiting HTCondor

D C Aiftimiei[1], M Antonacci[2], S Bagnasco[3], T Boccali[4], R Bucchi[1], M Caballer[5], A Costantini[1], G Donvito[2], L Gaido[3], A Italiano[2], D Michelotto[1], M Panella[1], D Salomoni[1] and S Vallero[3]

[1] INFN-CNAF, Viale Berti Pichat 6/2, 40127 Bologna, Italy
[2] INFN Bari, Via E. Orabona 4, 70125 Bari, Italy
[3] INFN Torino, Via Pietro Giuria 1, 10125 Torino, Italy
[4] INFN Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[5] UPV, Camino de Vera s/n, 46022 Valencia, Spain

E-mail: `sara.vallero@to.infn.it`

Abstract. One of the challenges a scientific computing center has to face is to keep delivering well consolidated computational frameworks (i.e. the batch computing farm), while conforming to modern computing paradigms. The aim is to ease system administration at all levels (from hardware to applications) and to provide a smooth end-user experience. Within the INDIGO-DataCloud project, we adopt two different approaches to implement a PaaS-level, on-demand Batch Farm Service based on HTCondor and Mesos. In the first approach, described in this paper, the various HTCondor daemons are packaged inside pre-configured Docker images and deployed as Long Running Services through Marathon, profiting from its health checks and failover capabilities. In the second approach, we are going to implement an ad-hoc HTCondor framework for Mesos. Container-to-container communication and isolation have been addressed exploring a solution based on overlay networks (based on the Calico Project). Finally, we have studied the possibility to deploy an HTCondor cluster that spans over different sites, exploiting the Condor Connection Broker component, that allows communication across a private network boundary or firewall as in case of multi-site deployments. In this paper, we are going to describe and motivate our implementation choices and to show the results of the first tests performed.

1. Introduction

One of the challenges a scientific computing center has to face is to keep delivering a computational framework which is well consolidated within the community, such as the batch computing farm for non-interactive processing, while adhering to modern computing paradigms. The goal is to ease system administration at all levels, from hardware to applications, and to provide a smooth end-user experience.

1.1. The underlying computing paradigm

In the last years, it has been shown by various large companies that encapsulating tasks into higher and higher layers of abstraction can provide a remarkable competitive advantage [1]. Standard hardware infrastructures are being replaced by a multitude of heterogeneous devices (the Internet of Things). The application's architecture is no longer monolithic, instead it is being factorized into multiple *microservices*, with great benefits in terms of extensibility and re-usability. Process delivery is now based on a dynamic DevOps approach [2]. These novel paradigms naturally fit in the Cloud Computing model: *a style of computing in which scalable and elastic IT-enabled capabilities are delivered as a service using Internet technologies* [3]. We usually think about Cloud Computing in terms of virtual machines. This still remains a good model for Infrastructure-as-a-Service (IaaS) provisioning, while it could be a too heavy and static approach to virtualization in the case of an application-focused deployment. Therefore, Linux containers have gained great popularity in recent times, since they provide a light-weight virtualization (the kernel is shared with the host) and allow to package and run distributed application components within the same platform across different environments. Based on this novel computing paradigm, the INDIGO-DataCloud (INDIGO-DC) project [4] is aiming at developing an open source data and computing platform targeted at scientific communities, deployable on heterogeneous resources and provisioned over multiple e-infrastructures. The work we are going to describe in this paper is part of the INDIGO-DC Platform-as-a-Service (PaaS) component [5].

1.2. Batch System as a Service
We call *Batch System as a Service (BSaaS)* the strategy to automatically and dynamically deploy a complete batch system cluster, with appropriate user interfaces, in highly-available and scalable configurations. From the end-user perspective, an optimal solution would be a cluster tailored to her needs, in terms of available software and configurations, easy to use and modify. This would release lots of efforts on the system administrators side. On the other hand, while having (almost) full control on their own cluster, users should be isolated from the rest of the system as not to disrupt other users work or the underlying infrastructure itself. The most critical aspects of the BSaaS implementation concern networking and storage. Container-to-container communication and isolation have been addressed with overlay networks (L3). Concerning the storage aspects, which will not be discussed in this paper, factors such as scalability, performance and reliability have to be taken into account. We have explored the usage of CVMFS [6](using Parrot [7]) and the integration with the INDIGO-DC Data Services (OneData, Dynafed, FTS) [8]. In the following sections, we are going to describe the tools chosen to deploy the INDIGO-DC BSaaS as well as its architecture, also extending to multi-site deployments. Finally, we introduce the monitoring system and show the first results obtained.

2. Software stack
Mesos - Physical resources are administered with the Apache Mesos [9] cluster manager. Mesos is built using the same principles as the Linux kernel, but at a different level of abstraction: it acts at a data-center level rather than at the operating system level. The Mesos kernel abstracts CPU, memory, storage and other resources away from physical or virtual machines. It schedules processes across the entire distributed infrastructure and provides APIs for resource management. It has a master-slave architecture, where the slave agent runs on each managed node. The Mesos master can run in high-availability mode, distributed over several instances and where one instance at the time is elected as leader.
Marathon - In the context of a data-center level operating system, the Mesosphere Marathon [10] application acts as an *init* system: it is designed to launch long-running applications. Marathon provides applications with high-availability, scaling and self-healing capabilities. It implements an API for scriptability and service discovery, and an easy to use web user interface. Formally, Marathon is a Mesos *framework*, that is a set of master, scheduler and executor components designed to work in synergy with Mesos.
Zookeeper - Both Mesos and Marathon rely on the Apache Zookeeper [11] centralized service

to maintain configuration synchronization across the whole cluster.
Calico - Network isolation is achieved with Calico [12]. Calico is a layer 3 approach to data-center networking. It implements a virtual router on each compute node, which leverages the existing Linux kernel forwarding engine. Each router propagates the workload reachability information to the rest of the cluster using Border Gateway Protocol (BGP), either directly or via route reflectors in large deployments. Network policies can be enforced using ACLs on each compute node, to provide project isolation, security groups and external reachability constraints. Calico ACLs leverage the kernel firewall (iptables).
Docker - Application isolation (in terms of code, run-time, system tools and system libraries) is achieved with the Docker [14] containerization platform. Docker containers are deployed using the *Docker Containerizer* module of Mesos.
Etcd - The CoreOS Etcd [13] distributed key-value store is used to synchronize network configuration across the entire infrastructure. Both Docker and Calico refer to the Etcd store to retrieve network configuration information.
HTCondor - HTCondor [17] is the LRMS chosen to implement the BSaaS. Besides being open source and widely used in the scientific community, we chose HTCondor also because it's *cloud-aware*. This means that this batch system is suited to work in a dynamic environment, where the list of worker nodes belonging to a cluster is not fixed. Each node registers with the central *Master* service when it is instantiated and it is removed from the list by the central service called *Collector* when no longer available.

2.1. Provisioning
The reference infrastructure is composed of one or more front-end machines running the core services (Mesos, Marathon, Zookeeper, Calico and Etcd) in a fault-tolerant way and a number of execution nodes. Mesos, Marathon, Zookeeper and Calico run as Docker containers, while only Docker and Etcd are installed directly on the physical or virtual hosts. Since Etcd is the store from which Docker retrieves the network topology, we preferred to keep it independent from Docker itself. The entire infrastructure is deployed using the *Ansible* [15] provisioning and orchestration tool. We have developed Ansible *roles* for all components, which were made publicly available on the *Ansible Galaxy* website [16]. Each role supports two back-end operating systems: Enterprise Linux 7 and Ubuntu $> 14.x$. The roles that need to be installed on the front-end servers are: Docker, Etcd, Calico, Zookeeper, Mesos master and Marathon. On the execution nodes: Docker, Calico and Mesos slave. The complete infrastructure deployment can be configured and orchestrated via a single Ansible *playbook*.

In the current implementation, the HTCondor cluster is instantiated as a set of Marathon applications, using template files. This will be explained in more details in Section 3.3. The cluster deployment can be automated through the INDIGO-DC PaaS platform [4], using a single TOSCA [18] template submitted to the INDIGO-DC Orchestrator. The Orchestrator can also manage multi-site deployments on private Clouds as well as on AWS [19].

3. Architecture
3.1. Infrastructure
Each of the machines composing the cluster, both front-end and execution nodes, runs an instance of the *docker-engine* daemon and of the Calico node. Each of the front-end servers also runs an instance of Etcd, forming a fault-tolerant configuration in which one of the instances is elected as the leading master and it is replaced by any of the other instances in case of failure. Global L3 networks, i.e. that span over the entire cluster, are created via the docker-engine by specifying the Calico driver. The information is stored on Etcd and retrieved by all the docker-engine instances across the cluster. In turn, Calico nodes retrieve from Etcd the information about configured networks and workloads (containers) running on each host. According to this

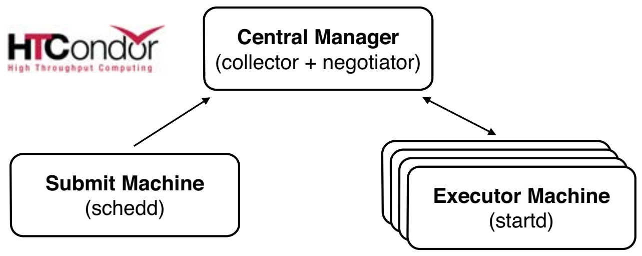

Figure 1. Services needed to deploy a complete HTCondor cluster are packaged inside Docker containers and deployed as Marathon Long Running Services. Each container is represented by a box in the figure. The HTCondor daemons required by each application are indicated within brackets.

information, the Calico's agent named Felix programs routing rules and ACLs into the Linux kernel of each host. Each instance of the BSaaS will have a dedicated Calico network.

3.2. Orchestration
Each of the front-end servers runs an instance of the Mesos and Marathon masters and of the Zookeeper store, in order to form a fault-tolerant configuration for these services as in the Etcd case. Each execution node runs an instance of the Mesos slave, and advertises all (or a configurable fraction) of its resources. Slaves can also be labeled with different tags, if needed, and applications can be pinned to a specific host type.

In our current approach, the various HTCondor daemons needed to deploy a complete cluster are packaged inside pre-configured Docker images and deployed as Long Running Service (LRS) through Marathon. This ensures fault-tolerance and scaling capabilities. Nevertheless, auto-scaling is not possible since the BSaaS is a *stateful* application, and the scaling policies provided by Marathon based on CPU or memory loads, are not suited to this case. Nevertheless, auto-scaling can be enabled for example with CLUES [20], an energy management system which monitors resource status and triggers the addition/removal of new slaves accordingly.

Another strategy on which we are currently working is to implement a dedicated HTCondor framework for Mesos, that can be used by itself or as a component in the more complex INDIGO-DC PaaS system, also in conjunction with an orchestration layer like Marathon. The new framework will consist of a scheduler to implement HTCondor policies on the resource offers provided by the Mesos master and a dedicated executor to launch tasks on the slave nodes. The benefits of an ad-hoc framework are first of all a fine-grained level of control on the tasks the application is responsible for. Moreover, it is possible to implement the preferred authorization rules and roles for multi-tenancy and to define application-specific scaling rules.

3.3. Application (BSaaS)
Each BSaaS instance is composed of three Marathon applications, sketched in Figure 1 and described below.
Central Manager - This application runs the HTCondor Collector and Negotiator daemons. The Collector is responsible for collecting the information about the pool status. All other daemons periodically send to the Collector ClassAd updates containing all the information about the state of the daemons, the resources they represent or resource requests in the pool. They also query the Collector for information required by their own operations.
The Negotiator is responsible for all the match making within the HTCondor system. Periodically, it queries the Collector for the current state of all the resources, it contacts the scheduling

daemons that have waiting resource requests and tries to match available resources with those requests. The Negotiator is also responsible for enforcing user priorities in the system. Submitter and Executor operations are coordinated by the Central Manager.
Submitter - It acts as access node and runs the Schedd daemon. This daemon represents resource requests to the HTCondor pool and manages the job queue. Any machine that is to be a submit machine needs to have a Schedd running. It advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a job has been matched with a given resource, the Schedd spawns a daemon called CondorShadow to serve that particular request. The Submitter machine also runs the *sshd* daemon, to allow the end-user to log-in and submit jobs to the pool. This application can be scaled to more than one instance.
Executor - It runs the Startd daemon, which represents a given resource to the HTCondor pool, as a machine capable of running jobs. It advertises certain attributes about machines that are used to match it with pending resource requests. The Executor application should be scaled to the number of instances required to cope with the number of queued jobs.

All applications also run an instance of the HTCondor Master daemon, which is responsible for keeping all the other daemons running.

An example of the Marathon templates used to instantiate these applications can be found on GitHub [21], together with the files needed to build the Docker containers. The three applications are based on the same base container, which is configured when the application is instantiated through a set of parameters passed to the *entrypoint* script. All daemons are started inside the container using the *Tini* [22] lightweight init system and *Supervisor* [23] to monitor and control the different processes. The BSaaS can also be enabled to run Message Passing Interface (MPI) applications by starting the sshd daemon on all the execution nodes and by proper HTCondor configuration (i.e. configuring the DedicatedScheduler) passed as mounted volumes to the Docker containers.

All the BSaaS instances run within a private network, and users normally do not have access to all the machines in the data center, but rather to a restricted number of access nodes. One possible solution to this problem is to instantiate a container with two NICs, one for the host network and one on a dedicated Calico network, with proper ACLs to be able to communicate with each BSaaS cluster. The container runs an sshd daemon configured to forward each user request to the appropriate Submitter machine, possibly with the help of a service discovery application (i.e. Mesos-DNS).

4. Geographical deployment

We have studied the possibility to deploy an HTCondor cluster that spans over different sites. For this purpose, we have set up a front-end server at each of the INFN sites of Bari, Bologna and Torino, each server also running an instance of the Mesos slave. The three Autonomous Systems (AS) are connected by the 10Gb/s GARRX NREN [24] and the IP over IP protocol has been enabled between the front-end servers. The network topology is a full-mesh among all hosts. For larger deployments we envisage a more scalable configuration with iBGP peering within the single AS and a Route Reflector (RR) per AS, which is eBGP peered with the RR on the other AS.

Within the BSaaS application, we have exploited the Condor Connection Brokering (CCB) component that allows communication across a private network boundary or firewall.

	Torino	Bologna	Bari
Torino	0.07 ± 0.02	11 ± 3	18 ± 6
Bologna	32 ± 10	0.09 ± 0.03	14 ± 4
Bari	18 ± 6	14 ± 4	0.18 ± 0.06

Table 1. Container-to-container communication: Round Trip Time (RTT) in ms measured between the sites of Bari, Bologna and Torino. Column headings indicate the site starting the connection.

4.1. First results

We have tested the container-to-container connectivity across the three sites. There is a distance of approximately $300km$ between the cities of Torino and Bologna and of approximately $600km$ between the cities of Bologna and Bari. Table 1 shows the Round Trip Time (RTT) in ms measured between the three sites. Column headings indicate the site starting the connection. The error is the standard deviation over 10 measurements. As expected, the matrix of results is symmetric, with the exception of the connection from Torino to Bologna, which is due to a known instability in the connection between the two sites at the time the measurement was recorded. The RTT increases by about two orders of magnitude when going from local connection to a connection between the two most distant cities (Torino and Bari), but is still compatible with efficient execution of batch jobs (excluding MPI workloads).

We also ran some stress tests using the Linux *stress* utility to simulate a heavy workload on the BSaaS and recorded no instabilities in the system.

5. Monitoring

We have developed a system to monitor the Mesos cluster status, which gives us more extensibility in collecting metrics and selecting time periods than the built-in Mesos Graphical User Interface (GUI). Data are collected with a custom Python script, through http GET requests to the Mesos server. Metrics are injected in *Graphite* [25], a monitoring system of which we exploit the capability to store numeric time-series data. Stored data are visualized through *Grafana* [26]: a tool for querying and visualizing time series and metrics which easily integrates with Graphite. The metrics collected are those provided by the Mesos server, such as: number of active slaves, number of running tasks, memory and CPU usage.

In order to monitor the application (BSaaS) performance, we rely on the specific HTCondor python API [27] to send data to the *Elasticsearch* analytics engine. Metrics are visualized with Grafana, using the specific Docker plugin. In this way we can collect metrics concerning the container activity (i.e. CPU or memory usage) or the HTCondor jobs (i.e. idle or running jobs, accumulated execution time). An example of BSaaS dashboard displaying these metrics is shown in Figure 2.

6. Conclusions and outlook

In this paper we described the implementation of the *Batch System as a Service* (BSaaS) application. Our efforts were directed at providing a well consolidated computing framework, while conforming to modern computing paradigms. The tools used for the automated deployment of a complete batch system on-demand are: Docker, Calico, Mesos, Marathon and HTCondor. The entire infrastructure is automatically deployed with Ansible.

We have studied the possibility of a geographical deployment of the system. For this purpose, we set up a VPN connection between the three INFN sites of Torino, Bologna and Bari. Both

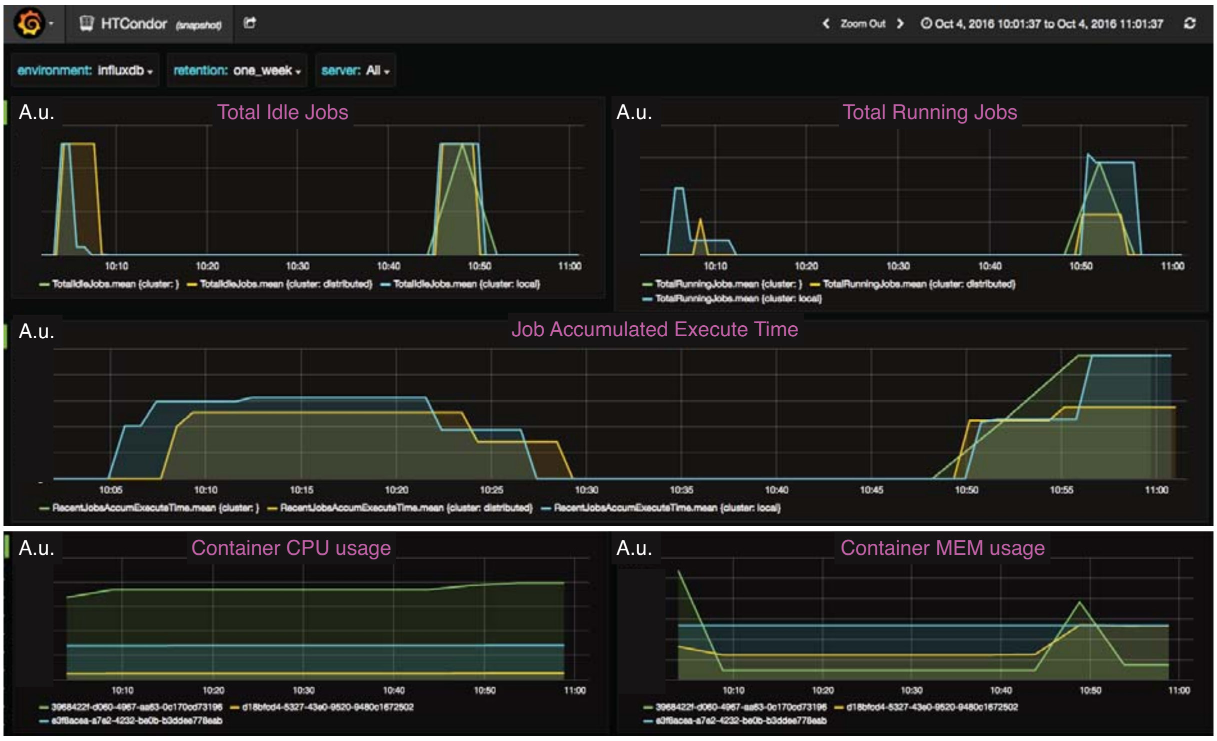
HTCondor (snapshot)
Zoom Out
Oct 4, 2016 10:01:37 to Oct 4, 2016 11:01:37
environment: influxdb
retention: one_week
server: All
A.u.
Total Idle Jobs
TotalIdleJobs.mean {cluster: }
TotalIdleJobs.mean {cluster: distributed}
TotalIdleJobs.mean {cluster: local}
A.u.
Total Running Jobs
TotalRunningJobs.mean {cluster: }
TotalRunningJobs.mean {cluster: distributed}
TotalRunningJobs.mean {cluster: local}
A.u.
Job Accumulated Execute Time
RecentJobsAccumExecuteTime.mean {cluster: }
RecentJobsAccumExecuteTime.mean {cluster: distributed}
RecentJobsAccumExecuteTime.mean {cluster: local}
A.u.
Container CPU usage
A.u.
Container MEM usage
3968422f-d060-4967-aa63-0c170cd73196
d18bfcd4-5327-43e0-9520-9480c1672502
e3f8acea-a7e2-4232-be0b-b3ddea778eab

Figure 2. Example of BSaaS dashboard displaying application metrics. Data are injected in Elasticsearch and visualized through Grafana.

the services cluster and the execute nodes were distributed over the three sites. The first tests performed confirmed the feasibility of a geographical deployment.

Finally, we set up a cluster and application monitoring system, relying on Graphite, Elasticsearch and Grafana.

In the next steps, we will focus on consolidating the framework by testing it with real use-cases. For instance, the BSaaS model was chosen as one of the provisioning modes used at the University of Torino HPC cluster *OCCAM* [29].

The final work will appear in the *INDIGO-DataCloud Service Catalogue* [30].

Acknowledgments

The authors would like to acknowledge the support of the INDIGO-DataCloud project, funded by the European Commission's Horizon 2020 Research and Innovation Programme (grant agreement RIA 653549).

References

[1] Modernize Application Development to Succeed as a Digital Business *GARTNER analysis* `https://www.gartner.com/doc/3270018/modernize-application-development-succeed-digital`
[2] DevOps *GARTNER IT glossary* `http://www.gartner.com/it-glossary/?s=Devops`
[3] Public Cloud Computing *GARTNER IT glossary* `http://www.gartner.com/it-glossary/public-cloud-computing/`
[4] Salomoni D, Campos I, Gaido L, Donvito G, Antonacci M, Fuhrman P, Marco J, Lopez-Garcia A, Orviz P, Blanquer I et al. 2016 *Preprint* arXiv:1603.09536
[5] Barbera R, Campos Plasencia I, Donvito G, Dutka L, Fuhrmann P, Gaido L, Hardt M, Marco J, Oliveira

Gomez J, Plociennik M and Salomoni D INDIGO-Datacloud: a Cloud-based Platform as a Service oriented to scientific computing for the exploitation of heterogeneous resources *In these proceedings*
[6] CernVM File System (CVMFS) `https://cernvm.cern.ch/portal/filesystem`
[7] The Parrot Virtual File System `http://ccl.cse.nd.edu/software/parrot/`
[8] Kryza B, Dutka L, Fuhrmann P and Kryza B Unified data access to e-Infrastructure, Cloud and personal storage within INDIGO-DataCloud *In these proceedings*
[9] Apache Mesos `http://mesos.apache.org/`
[10] Mesosphere Marathon framework for Apache Mesos `https://mesosphere.com/blog/2014/11/10/docker-on-mesos-with-marathon/`
[11] Apache Zookeeper `https://zookeeper.apache.org/`
[12] Project Calico `https://www.projectcalico.org/`
[13] CoreOS Etcd `https://github.com/coreos/etcd`
[14] Docker `https://www.docker.com/`
[15] Ansible `https://www.ansible.com/`
[16] Ansible Galaxy `https://galaxy.ansible.com/indigo-dc/`
[17] HTCondor `https://research.cs.wisc.edu/htcondor/`
[18] TOSCA specifications `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`
[19] Amazon Web Services `https://aws.amazon.com/`
[20] CLUster Energy Saving `http://www.grycap.upv.es/clues/eng/index.php`
[21] HTMesos `https://github.com/DS-CNAF/HTCondor-docker-debian`
[22] Tini `https://github.com/krallin/tini`
[23] Supervisor `http://supervisord.org/`
[24] GARRX NREN `http://www.garr.it/it/infrastrutture/rete-nazionale/infrastruttura-di-rete-nazionale` (website in Italian)
[25] Graphite `https://graphiteapp.org/`
[26] Grafana `http://grafana.org/`
[27] HTCondor Python API `https://github.com/fifemon/probes`
[28] Elasticsearch `https://www.elastic.co/`
[29] Aldinucci M, Bagnasco S, Lusso S, Pasteris P and Rabellino S OCCAM: a flexible, multi-purpose and extendable HPC cluster *In these proceedings*
[30] INDIGO-DataCloud Service Catalogue `https://www.indigo-datacloud.eu/service-component`