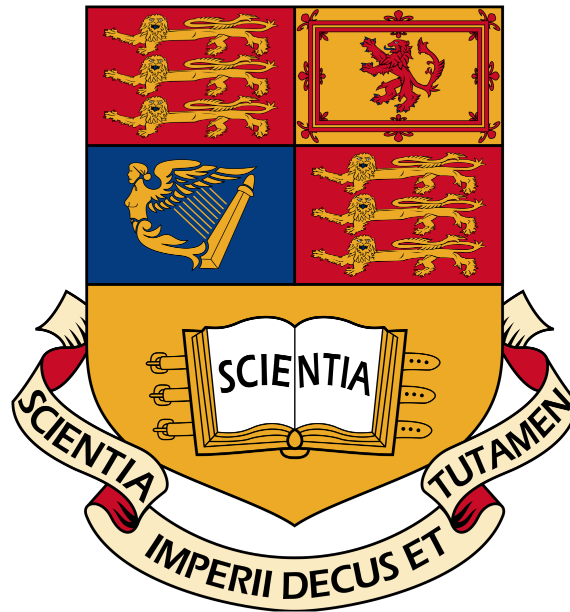
Imperial College London

Department of Computing



Cloud computing for big data experiments

MEng Individual Project

Author:
Adrian Bogdan Draghici

Supervisor:
Dr. Jonathan Passerat-Palmbach

Second marker:
Dr. Ben Glocker

June 2016

Abstract

Relentless technological advancements over the years have brought simulation and data processing to the core of science. The shift towards increasingly computationally intensive research matches the on-demand paradigm of cloud computing, but despite clouds offering virtually infinite pools of resources and abstractions ideal for running large-scale experiments, many scientists lack the time or expertise required to utilize them efficiently. Existing tools mediating the access to cloud environments are too specific to certain communities and require extensive configuration on behalf of the user.

In this project, we expand OpenMOLE, a scientific framework for remote execution of user-defined workflows, to support cloud environments under a generic design. We provide an fully operational implementation for Amazon EC2 and basic support for Google Compute Engine. The main novelty of the extension is the full automation of the resource provisioning, deployment and lifecycle management for the cluster of virtual instances running user tasks. Additionally, we introduce automatic bidding for low-price cluster nodes and translations from cumulative resource specifications to sets of virtual machines.

During evaluation, we benchmark response times to job events in the cluster, estimate costs of distributing data pipelines to the cloud and suggest ideas for optimising resource consumption. From a qualitative point of view, we demonstrate how execution environments are interchangeable and how cloud environments can be employed with zero user configuration.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Jonathan Passerat-Palmbach, for his guidance, contagious enthusiasm and all the long hours he has dedicated towards our discussions.

Lastly, I would like want to express my sincerest gratitude to my parents, Iulică and Melania, and my girlfriend, Daria, for their unconditional love and support throughout the years.

Contents

1 Introduction **9**

1.1 Motivation 10

1.2 Contributions 11

2 Background **13**

2.1 Grids and Cloud Computing 13

2.1.1 Amazon Web Services 14

2.2 Workflow Platform Ecosystem 15

2.2.1 Taverna 16

2.2.2 Galaxy 17

2.2.3 Tavaxy 19

2.2.4 Kepler 20

2.2.5 Pegasus 21

2.3 Cluster Deployment on Clouds 23

2.3.1 StarCluster 24

2.3.2 Elasticluster 26

2.3.3 CfnCluster 28

2.3.4 Apache Jclouds 30

2.3.5 Mesos 33

3 OpenMOLE & GridScale **37**

3.1 OpenMOLE 37

3.1.1 Architecture 38

3.1.2 Domain Specific Language 39

3.1.3 Environments 43

3.1.4 Job Distribution 44

3.1.5 CARE 46

3.1.6 Job Management 48

3.2 GridScale 49

3.2.1 Principles 49

3.2.2 Module Design 50

4 Implementation **53**
4.1 GridScale AWS Module . . . 54
4.1.1 Design Motivation . . . 54
4.1.2 Cluster Deployment . . . 56
4.1.3 Load Balancing . . . 61
4.1.4 Spot Instances . . . 62
4.2 OpenMOLE AWS Environment . . . 63
4.2.1 Job Service Lifecycle . . . 64
4.2.2 Spot Clusters . . . 64
4.2.3 Resource Mapping . . . 64
4.3 Other Cloud Platforms . . . 65
4.3.1 GCE Support . . . 66

5 Evaluation **68**
5.1 Fully Automated Experiments . . . 68
5.2 GridScale Benchmarks . . . 69
5.2.1 Methodology . . . 70
5.2.2 Results . . . 70
5.3 OpenMOLE Workflow Benchmarks . . . 73
5.3.1 π Computation . . . 73
5.3.2 Random Forest . . . 74
5.4 Spot Clusters . . . 75
5.5 Challenges and Limitations . . . 75

6 Conclusion **77**
6.1 Future Work . . . 78

References **79**

Appendices **87**

A π Computation **87**

B Random Forest **88**

Chapter 1

Introduction

Scientists have spent decades building intricate mathematical models for systems and phenomena observed in all areas of life sciences. Such models have greatly expanded our understanding of the complex systems they describe, but the dramatic developments in technology and the increase in widely available computational power in recent years have exposed another direction in which current research methodologies can progress.

Simulation in high performance computing environments is today the main approach used to validate scientific models in application domains ranging from astronomy to biomedical or earthquake science. Analytical solutions are often impossible to obtain, given the non-linear nature of these systems, so empirical observations drawn from running them in virtual settings is the only sensible option for further tuning and optimisation.

The prevalence of hypotheses developed by centralising and mining vast arrays of data sources has led to an era of data-centric research [1], where powerful cluster, grid and cloud computing platforms are open for scientific usage. However, the expertise required to operate this infrastructure is beyond the skill set of most researchers, so the efficiency of experimentation and the quality of insights drawn from it are heavily influenced by the performance of tools available to manage and process the available data.

Workflow management systems provide support for creating data processing pipelines and automating historically tedious tasks such as tuning program parameters by running them repeatedly against many different datasets, defining work units and their interdependencies, or collecting and persisting results of analyses [1, 3]. This is all achieved in the context of delegating the work efficiently over resources offered by distributed computing environments.

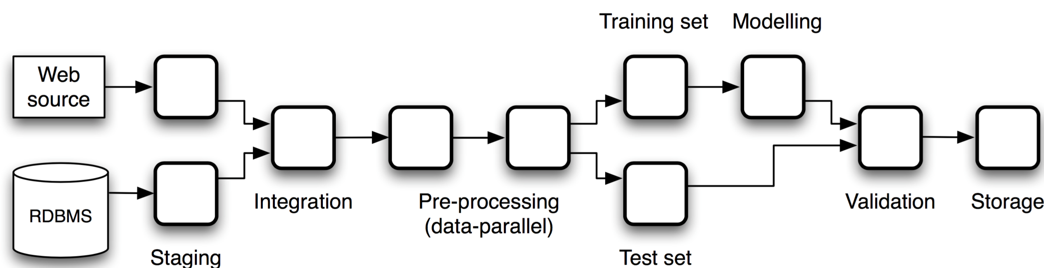


Figure 1: Workflow example [2].

1.1 Motivation

Although multiple workflow management systems such as Taverna [4], Kepler [5], Pegasus [6] or Galaxy [7] are already established, many of them focus on specific scientific areas. Since the bioinformatics community is particularly active in adopting the usage of scientific workflows, tools like Galaxy and Kepler have been historically tailored for the needs of researchers in this field.

OpenMOLE [8] is a scientific workflow engine that leverages the natural parallelism of simulation algorithms and targets distributed computing environments to run them. Compared to other existing platforms, OpenMOLE exposes some key features that make it unique:

- It is not intended for use within a single scientific community, allowing for formalisms that make it generic.
- It focuses on hiding the complexity and heterogeneity of hardware and software infrastructure that grids and clusters provide, separating the workflow definition from the underlying execution environment [9].
- By following a zero-deployment approach, it does not need any guarantees regarding the configuration of its runtime environment. It integrates a standalone packaging system, CARE [10], and it ensures reliable execution on remote hosts automatically by copying necessary resources on demand [11].
- It treats input models as black-boxes, allowing for interchangeable definitions in diverse languages or even packaged binaries [8]. This means that already existing applications are not limited by a pre-configured toolbox and run as expected by default.

- The focus on flexible design and scalability generates control flow structures inexistent in most rival platforms. Some of these include loops, conditional branching, or the ability to include whole workflows as subtasks [8]. This makes workflows themselves reusable components that can be published and distributed on the OpenMOLE marketplace [12].
- The DSL[1] it uses to create workflows is an extension of the Scala [13] programming language. This allows for type-safe job definitions and enables OpenMOLE to catch configuration errors earlier than other systems that rely on plain text descriptions.

Options that OpenMOLE currently offers to run workflows include:

- Multithreading on a local machine.
- SSH connections to remote servers.
- Grids, such as EGI[2] [14].
- Clusters managed by a wide range of software, including PBS[3] [15], Torque [16], SGE[4] [17], Slurm [18], HTCondor [19], or OAR [20].

However, at the moment OpenMOLE does not support distributing the workload to cloud computing environments. This is a significant disadvantage compared to other similar applications, since cloud providers are ubiquitous and cheap resources. Providing cloud support will enable a large user base to accelerate their research using OpenMOLE.

1.2 Contributions

The initial aim of this project was to enhance OpenMOLE by adding the ability to target computing clouds to the arsenal of available execution environments. More specifically, the main goal was to support running jobs on Amazon EC2[5] [21], while remaining generic enough to potentially integrate with other cloud providers and infrastructures (Google Cloud [22], OpenStack [23] etc.), including Imperial College's Department of Computing private cloud that runs on a CloudStack [24] deployment.

The key contributions of the project include:

[1] Domain Specific Language
[2] European Grid Infrastructure
[3] Portable Batch System
[4] Sun Grid Engine
[5] Amazon Elastic Compute Cloud

- A survey on the APIs[6] and frameworks suitable to instantiate and coordinate computations on cloud infrastructures, including researching the cloud distribution architecture of similar open-source scientific workflow engines, as well as industrial software.
- A fully operational module for distributed computation on Amazon EC2 as part of GridScale [25], the engine powering OpenMOLE's distributed computation service. We also provide a basic Google Compute Engine module, which is still under active development.
- The integration of the GridScale EC2 module in OpenMOLE, where we create a new remote execution environment allowing the delegation of entire workflows to the cloud.
- Complete automation of the process of provisioning and bootstrapping clusters of virtual instances in the cloud. These steps are invisible to the end user, who only needs to provide his access credentials, but can set extra preferences such as choosing the instance types or size of the cluster. Additional features include a bidding strategy for renting auctioned instances or the translation of traditional requirements such as processing power or memory into a set of machines covering the needs of the user.
- An evaluation of the Amazon EC2 remote execution environment. This is done by benchmarking its underlying GridScale library against different types of cloud instances and analysing response times. Additionally, we investigate the performance of workflows run on EC2 under various configurations and discuss possible bottlenecks and improvements.

[6]Application Programming Interface

Chapter 2

Background

This chapter starts with a brief comparison between grids and cloud computing. Next, it gives an overview of the current state of the workflow engine ecosystem by looking at some of the existing platforms with a focus on the execution environments they support, and cloud deployments in particular. This leads to the last section, where we investigate tools and frameworks that could allow deploying a cluster in the cloud and scheduling jobs on the respective instances.

2.1 Grids and Cloud Computing

Ever since the early days of high performance computing, researchers have been running resource-intensive experiments and simulations on supercomputers, machines with hundreds of thousands of cores and high parallelisation capabilities [26]. However, in the last two decades the extensive costs and restricted access to such systems have shifted computation towards local clusters and grids.

Grids are distributed networks of remote computers crowdsourced in general from academic communities. They harness unused resources in the infrastructure and are managed through middleware software (e.g. EMI[1] [27]) that coordinates incoming computation across the machines they interconnect [28]. Access to grids is usually free and restricted to research projects.

Although clouds evolve from grids and both aim to achieve similar goals in the area of distributed computing, there are some substantial differences in terms of features and how they operate [28–30]:

[1] European Middleware Initiative

- Clouds offer on-demand provisioning of resources. This enables users to easily scale infrastructures hosted in the cloud by simply requesting more instances as resource requirements grow. They create an illusion that resources can be provisioned infinitely by functioning at such a large scale that they can fulfil practically any demands.
- As opposed to grid operators, cloud providers are generally commercial and charge users on a pay as you go basis. Usage is usually measured in hours per instance and the business model creates no upfront planning or costs for users. Since they are paid services, cloud providers also offer better reliability and uptime guarantees. Private clouds can also be set up for users that want to abstract away their pool of resources but are concerned about security of their data.
- Clouds are particularly suitable for hosting long-running jobs, such as web servers. In this case, users can easily take advantage of features such as automatic scaling, where more servers are automatically provisioned to match abrupt increases or decreases in the number of incoming requests.
- Virtualisation is used extensively in clouds and it permits running legacy software that has very strict environment requirements. Although they might be sharing a physical machine with other consumers, users can fully customise the virtual runtime in isolation by choosing the operating system and libraries they need. Consolidating a homogeneous fleet of machines is also straightforward, since snapshots of a runtime environment can easily be ported to other instances.

Considering the advantages listed above and growing support for cloud services, they are a good fit for the deployment of workflow management systems. However, as discussed in the next section, current workflow platforms do not take full advantage of advanced cloud features such as automatic scaling.

2.1.1 Amazon Web Services

Since the project primarily aims at supporting Amazon EC2 as a target environment for running experiments via OpenMOLE, this subsection briefly describes some of the basic terminology related to Amazon's cloud services:

- *AWS*[2] [31] is the whole suite of services provided by Amazon as part of its cloud platform.

[2] Amazon Web Services

- *EC2*[3] [21] is the cloud service that provides on-demand computational resources.
- *EC2 Spot Instances* are normal EC2 instances that are temporarily free and are auctioned by Amazon. The highest bidder retains the right to use the resources.
- *S3*[4] [32] is a general-purpose persistent file storage service.
- *EBS*[5] [33] concerns storage volumes that are attached to machines provisioned through EC2.
- An *AMI*[6] is a snapshot of the environment on an EC2 machine. New EC2 instances can easily be created from a given AMI.

2.2 Workflow Platform Ecosystem

Chronologically, scientific workflow systems have emerged from the bioinformatics community along with the recent trend towards data-driven research. Their large number and segregation despite achieving similar purposes could be explained by many research groups independently trying to formalise, consolidate and generalise their workflows. Therefore, most systems achieve comparable goals, with slight variations. Common features include [1]:

- Creation and definition of reusable tasks or work units. A task can represent anything from processing an image to running an expensive computation or invoking a service over the web.
- A graphical user interface that simplifies the flow of tasks by allowing definitions via a simple visual representation. See Figure 2 for an example of this.
- An execution platform that runs the workflow, hiding the complexity of calling service applications, managing and storing data, setting up and consuming remote computational resources, dealing with failures and logging results and unexpected behaviours. This is the engine of the application.
- A collaboration platform, where users can interact and share workflows.

From the multitude of existing workflow systems, we have selected some of the most often referenced ones for closer inspection. Since our focus only spans the targeting of different

[3]Elastic Compute Cloud
[4]Simple Storage Service
[5]Elastic Block Store
[6]Amazon Machine Image

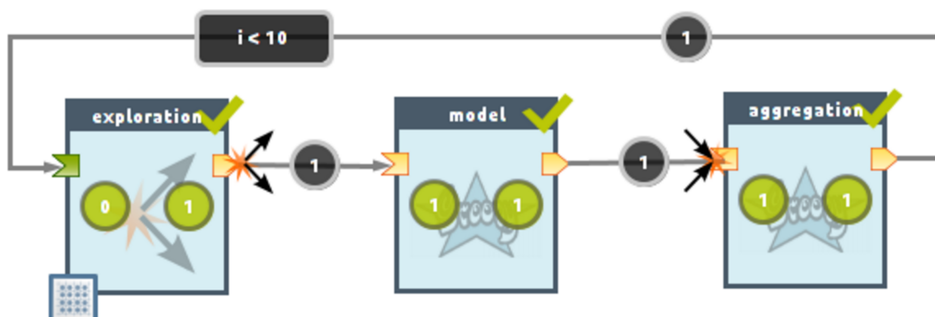


Figure 2: OpenMOLE graphical workflow [34].

remote execution platforms, we will generally omit the details of workflow definition and the underlying implementation, as well as the graphical design and collaboration factors. We are particularly interested in engines that support cloud environments and insights we can draw from their design and infrastructure.

2.2.1 Taverna

Taverna [35] is one of the most popular workflow systems. It was initially created as a framework for bioinformatics research and has remained used primarily in this field despite efforts from contributors towards expansion to other research areas.

The system has three main functional components:

- *Taverna Workbench* is the standard suite including the user interface and execution platform. However, this package alone is quite restricted since it only supports running the workflow locally and not distributing it remotely.
- *Taverna Server* is a suite that works on the principles of simple client-server interaction. A server instance stores workflow blueprints created by the community and the client is only allowed to trigger runs of the experiments via a web interface. In this model, only server administrators have permission to add workflow content, while regular users are not allowed to freely create and upload their own custom workflows to the server. Additionally, the need for a full installation and configuration of the server software in order to execute work remotely limits ease of deployment and creates an important entry barrier.
- *Taverna Player* is the web interface used by the client to send requests to Taverna Server.

Despite its maturity, Taverna does not, on its own, have built-in support for automatic server installations on grids or clouds. Instead, users need to develop custom orchestration infrastructure for these environments to allow deploying clusters coordinated by the instance where Taverna Server is installed. Both caGrid [36] and BioVeL [37] have implemented such solutions [38, 39] to take advantage of grid resources. On Amazon EC2, Taverna is only available as an Amazon Machine Image (AMI) runnable on a single instance, without support for distributed execution.

2.2.2 Galaxy

Galaxy [40] is another community specific web-based platform for managing workflows, focussing on genomic research. Conceptually, it is driven by the motivation to ensure accessibility of computational resources by providing researchers with simple web interfaces to interact with distributed environments, reproducibility of experiments by tagging and recording order and intent of each action users take, as well as transparency of findings by consolidating a robust collaboration framework.

CloudMan [41] is the cloud resource management tool used by Galaxy to instantiate prepackaged clusters on Amazon EC2 machines. To achieve this, the user needs to use the AWS Management Console to request an instance that will be used as the master node of a new SGE cluster. Next, the number of slave instances in the cluster can be adjusted using the CloudMan Console, as shown in Figure 3.

Figure 3: CloudMan cluster management console [41].

Since EC2 instances do not save data on disk by default, persistence on Amazon Elastic Block Storage can be explicitly turned on from the CloudMan Console. CloudMan also deals with cases when the initial capacity of an EBS volume attached to the master instance is exceeded by safely pausing activity in the cluster before reattaching a new expanded capacity volume and resuming work. However, the job submission system still does not achieve full automation, since it requires a human to manually turn on the master node in the cluster, as opposed to the system being brought up on the fly and turned off on workflow termination. This is a problem in the context of EC2 or other commercial clouds, since it means that the user might continue to be charged for resources that are no longer being used.

One major advantage of CloudMan is its modular architecture, under which instances only use a lightweight AMI and reference the tools they need from external storage such as EBS, as shown in Figure 4. This grants further flexibility in terms of updating the system, because the AMI does not need to be repackaged frequently and the state of machines can be modified by simply writing on persistent storage.

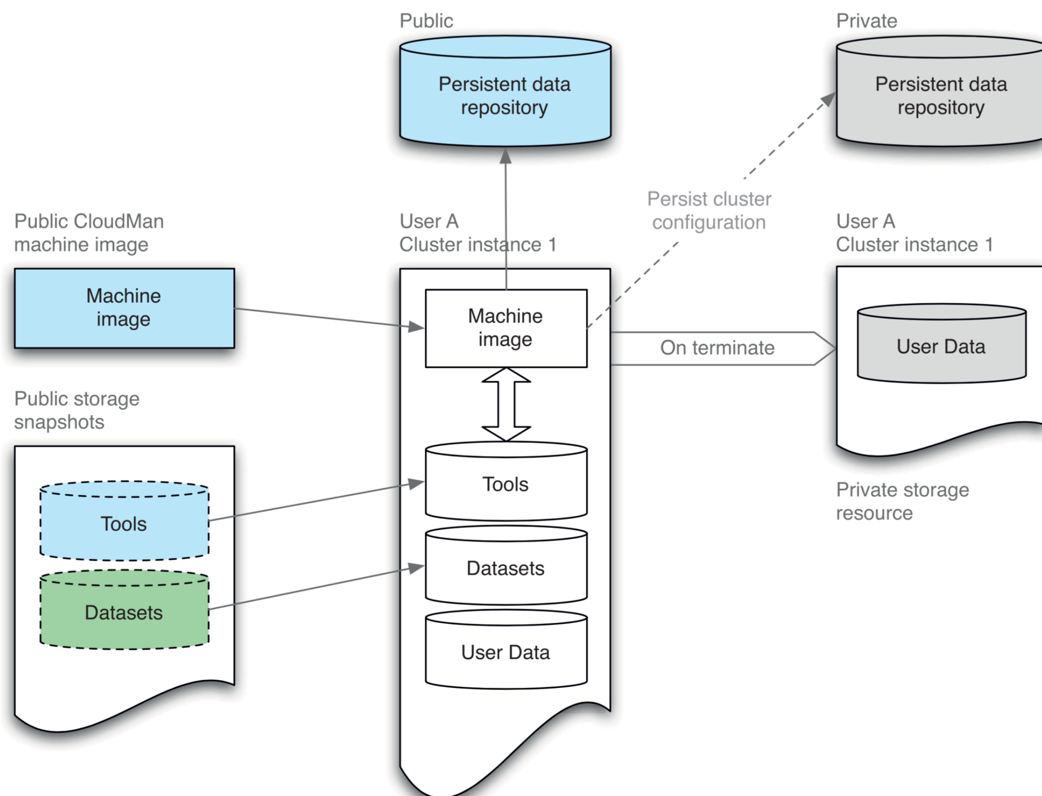


Figure 4: CloudMan modular architecture [41].

2.2.3 Tavaxy

Tavaxy [42] was created from the desire to ease the sharing of scientific workflows between the increasingly large user bases of Taverna and Galaxy within the bioinformatics community. The limited interoperability between the two systems was caused by differences in workflow description languages, as well as execution engines and overall design. Tavaxy consolidates Taverna and Galaxy workflows to run and be edited in a single environment, encouraging the community to create composite routines with building blocks from both worlds.

Tavaxy focuses on efficiently and transparently delegating workload to grid and cloud platforms. It can run a cluster when it is provided with a distributed file system similar to NFS and a standard job scheduler like SGE. The preferred cloud platform is Amazon EC2 and provisioning extra resources is done via a simple web interface, operated similarly as for Taverna and Galaxy.

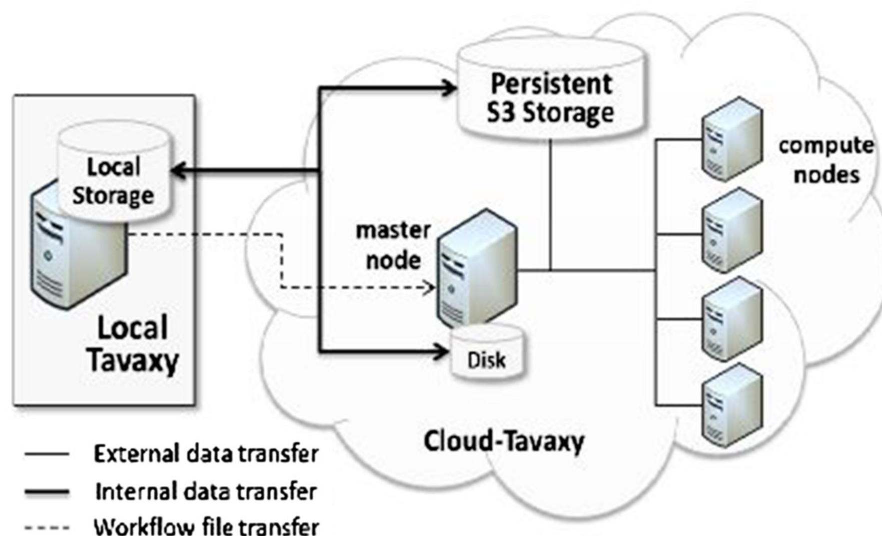


Figure 5: Tavaxy interaction between a local machine and Amazon EC2 [42].

Three different modes are available for delegating computation to EC2:

- *Whole system instantiation*, where the user has no local version of Tavaxy installed and can bootstrap a new instance from a provided AMI. This will automatically create and configure a cluster that the user can control through a web console. Amazon S3 [32] is used for persistent storage of shared data in the cluster.
- *Sub-workflow execution*, which presumes a local installation and Tavaxy used for

workflow design and allows the user to create a cluster in the cloud from a more lightweight AMI wrapping the runtime environment. The local machine sends the workflows remotely for execution and waits for results of the run. The user has two options for transmitting the input data and persisting the results. The first option is to send Inputs along with the workflow definition to master node machine and save outputs manually to local storage. The alternative is to upload input data to S3 and configure the cluster to direct reads and writes to S3 directly. The general architecture for this mode of operation can be observed in Figure 5.

- *Single task instantiation*, which is similar to sub-workflow execution, except that only a task in the workflow is delegated to the cloud.

2.2.4 Kepler

Kepler [5] is one of the first general-purpose scientific workflow systems, recognising the need for transparent and simplified access to high performance computing platforms more than a decade ago. It also underlined concepts such as reusable and reproducible workflow runs, scalability of models, as well as fault tolerance and reliability in the context of remote execution [43].

The novelty of Kepler's design resides in the actor-oriented model. Actors are basic independent components representing either data sources or operations in the workflow. They are interconnected via channels and own receivers that handle their external communication, as well as input and output ports.

However, the execution model differs from standard systems in that its flow is not directed by the topology of the network. Instead, a special component named *director* establishes the order of execution for actors, selects the operations they perform and orchestrates their communication by controlling their respective receivers [2]. This means that actors are not necessarily executed sequentially but are triggered by data received on incoming ports. Such as design reveals possibilities for concurrency semantics and renders the model fit for embedded systems simulations.

Although Kepler does not support execution on clusters in cloud environments out of the box, research groups using it have developed custom solutions to partially support this functionality. Wang and Altintas [44] propose EC2 actors capable of managing Amazon virtual machines and suggest using StarCluster [45] to build virtual clusters from the Kepler AMI they provide. This approach is sensible and can be used in conjunction with

any other workflow systems, but is not readily available for Kepler at the moment.

2.2.5 Pegasus

Pegasus [6] is a system that initially gained popularity for mapping complex workflows to resources in distributed environments without requiring input from the user [46]. Since its inception, many other similar applications have incorporated this feature, but Pegasus employs several optimisations that improve runtime performance and resource allocation.

Pegasus makes a clear distinction between high-level workflows defined by users from their actual executed form. Abstract workflows allow portability to many runtime platforms and free the user from explicitly indicating specific resources that should perform the work, while concrete workflows precisely bind execution stages to specific storage, computation, and network resources. This setup allows for multiple optimisations that would otherwise be impossible, particularly considering the fast dynamics of cloud and grid platforms. The latest release of Pegasus relies on four essential components [47, 48]:

- The *mapper* receives an abstract workflow as an input and produces a concrete workflow, defining the software and hardware requirements of the computation. Additionally, it performs metadata processing to enable data provenance tracking and modifies the structure of the workflow by grouping suitable tasks.
- The *workflow engine* ensures the execution in topological order. This responsibility is delegated to DAGMan [49], a meta-scheduler that runs on top of HTCondor [19] and allows ready jobs to be run.
- The HTCondor *job scheduler* manages the queue of individual jobs. It supervises the execution and restarts task runs in the case of failures.
- The *workflow monitor* is a daemon that parses output logs and notifies the end-user on the status of the submission.

Pegasus performs most of its important optimisations at the mapping stage, since this is the point where the workflow is broken down into single tasks. The improvements with significant impact on performance concern the following aspects [48]

- *Data movement.* This refers to ensuring that data required by a job is collocated with resources where it is executed. For example, copying input data from a user's local server to EBS is highly preferred when running jobs on EC2 because Amazon

guarantees low latency when accessing its own storage systems.

- *Data reuse.* Pegasus is able to reuse intermediate results of the workflow that have already been computed during previous runs of the workflow if the definitions of the tasks and input data have not changed. The process requires careful coordination with the *data cleanup* phase in order to simultaneously leverage already known results and avoid wasting storage capacity.
- *Job clustering.* For many short-lived jobs, orchestrating the transfer of task results across machines in a cluster and long queueing times can incur high latency costs. Grouping related tasks into larger entities helps alleviate this problem by reducing the load on the machine that hosts the job submission system. This strategy also improves the overall performance of workflows with a large number of tasks by over 90%, as previous studies have shown [50, 51]. *Level-based horizontal clustering* (Figure 6) and *label-based clustering* (Figure 7) are some of the most effective strategies, although the latter requires users to explicitly label tasks when defining the workflow [47].

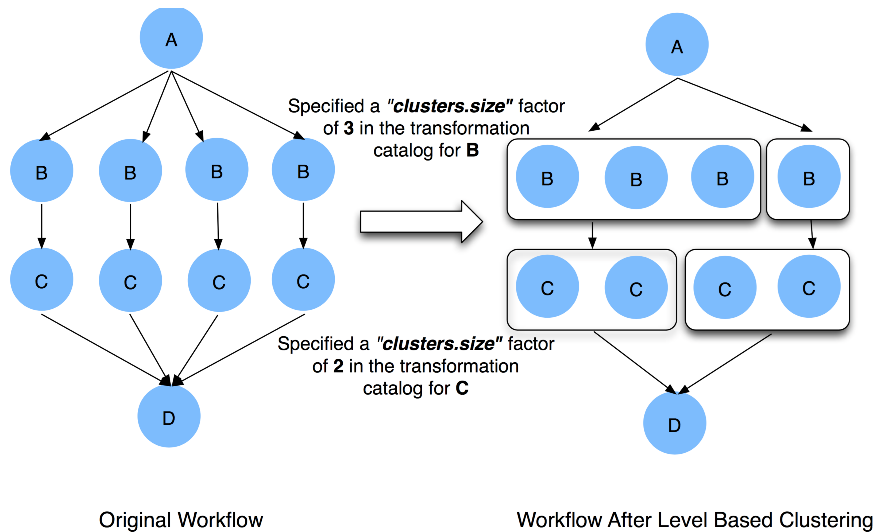


Figure 6: Level-based horizontal clustering targeting parallelisable tasks [47].

Although all the optimisations and design ideas discussed above apply to all distributed execution platforms, earlier deployments of Pegasus have confirmed several advantages of cloud environments. These include on-demand provisioning of resources and ability to easily ensure consistency of software installed on all the machines in a fleet [48].

However, further development of workflow management systems is needed in order to fully exploit cloud features such as automatic scaling of available resources. The main problem

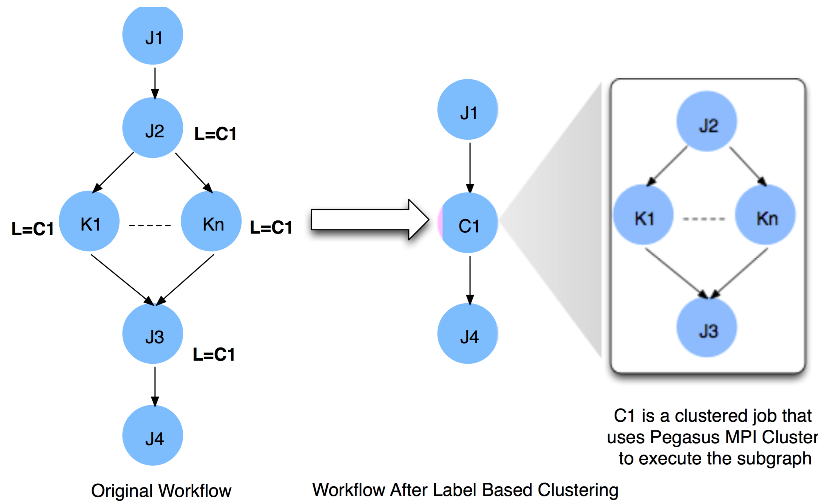


Figure 7: Label-based clustering [47].

is that the load of instances occurs in the case of long-running jobs. Indeed, upscaling usually leaves unused extra resources, while downscaling is even more challenging because long individual tasks cannot be split up any further. Basic solutions involve only allowing automatic scaling on idle instances or when having many short-lived jobs, but this is definitely an area open to future work.

Despite the trend towards cloud-based systems, the process of running Pegasus workflows in the cloud has still not been fully automated. Users are required to manually configure the job submission host and worker nodes to run the required software [52]. At the moment, the technical barrier for harnessing the cloud from Pegasus is higher than grid-based options, that are usually specifically designed for running experiments and provide a preconfigured software stack.

2.3 Cluster Deployment on Clouds

In this section, we analyse some of the tools that can be used for creating a cluster from a set of instances provided in a cloud environment. The main motivation behind the investigation is that deploying a cluster running one of the environments already supported in OpenMOLE allows leveraging the existing infrastructure by simply delegating the work to a cluster running in the cloud. We believe that this is a sane approach for developing a functional prototype, which can later be improved by using native cloud APIs in case this is required due to slowdowns caused by the additional clustering layer.

Throughout the investigation, we focus on the features relevant to managing cloud infrastructures. We are interested in a tool that:

- Supports most of the important cloud infrastructures (EC2, Google Cloud, OpenStack, CloudStack). We initially only target EC2, but we also intend to integrate with other platforms in the future.
- Is lightweight enough not to cause significant overhead on the performance of the instances in the cluster.
- Is open-source, since we plan to use it as part of GridScale. However, we also examine some commercial and proprietary systems to better educate our decision.
- Allows for effective automation in a concise manner by providing a clear and robust command line interface or API.

2.3.1 StarCluster

StarCluster [45] is an open-source cluster management tool that has been successfully used in both open-source and commercial products. It comes as a command line tool that specifically targets cluster deployment on Amazon EC2 and provides flexible high-level configuration options.

By default, StarCluster provides a set of public AMIs that include a lightweight software stack for job scheduling, intra-cluster communication and common scientific data manipulation tasks. From a system administration perspective, the out-of-the-box StarCluster configuration includes:

- NFS[7] [53] for sharing data between the instances in the cluster.
- The SGE job queuing and scheduling system.
- Security group setup for controlling the inbound and outbound traffic from the cluster.
- Password-less SSH between the nodes in the default security group.
- The possibility of attaching and mounting EBS drives to be used as NFS-shared storage by all the nodes in the cluster.

On top of this, the StarCluster AMI also contains tools like OpenMPI [54] and IPython [55] for writing natively parallel applications. Other features revolve around various preconfig-

[7] Network FIle System

ured software packages associated with high-performance computing development stacks, but they go beyond our use cases.

StarCluster relies heavily on the assumption that users prefer sensible defaults rather than extensive configuration. The standard installation requires a single `start` command to set up a cluster with a given number of nodes on EC2 and new clusters are automatically configured with NFS and SGE support. Listed below are some of the most useful commands exposed by StarCluster:

- `start` launches a cluster as per the specification provided in the configuration file. It allows for many different variations of cluster creation, where instances can also be simply started but not provisioned with the default software. The user can also simply perform a dry run and simply simulate starting the cluster to prevent unexpected issues in production.
- `terminate` completely purges the given cluster, shutting down all nodes and removing its previously created security group. If the cluster is backed up by EBS volumes, the root volumes of each node can also be deleted.
- `sshmaster` and `sshnode` allow for easy SSH access to all nodes in the cluster relying on the AWS keypair provided in the configuration file for authentication. Slave nodes are automatically numbered in a human-readable format - `node001`, `node002`, etc.
- `put` and `get` enable transferring files between the user's local machine and a running cluster, where they are instantly shared via NFS.
- `createkey` is a useful for creating a new SSH keypair and simultaneously both saving it locally and importing it to the user's AWS account.
- `addnode` and `removenode` allow for manual upscaling or downscaling by launching new nodes and attaching them as slaves to the cluster or tearing them down.
- `loadbalance` provides an automatic alternative for growing and shrinking the number of nodes in the cluster based on the load of jobs queued in SGE.

Elastic Load Balancing is an important feature based on data extracted periodically from the monitoring component of SGE. Depending on the load of the queuing system, StarCluster automatically scales the size of the cluster by adding or removing instance nodes. This ensures that the number of idle jobs in the system is never excessively high, although it is unclear how the heuristic will perform when handling numerous short-lived or few long-lived tasks.

The load balancing component allows choosing the minimum and maximum number of

nodes that the size of the cluster can vary between, as well as its growth rate. The user can align this with domain knowledge about the job submission patterns, meaning that a cluster used to run a highly volatile number of short-lived jobs should adapt faster to the increase in requests and provision more instances at each monitoring iteration than a cluster used to run long-lived jobs. Other useful options include plotting the number of instances used by the cluster over time, the ability to allow the cluster a stabilisation time period during which the load balancer does not run, or automatically killing the cluster after a specific duration or when the job queue is empty.

StarCluster also provides a bidding strategy for provisioning EC2 *spot instances* that allowed the StarCluster team to reduce instance renting costs by approximately 60% over longer periods of usage. Spot instances allow users to rent currently unexploited resources from idle machines for a significantly lower price than the standard flat rate, with the drawback that access may be suddenly terminated when the original owner needs the computing power back. The framework deals with the caveat of losing access to the machine due to the current spot price becoming higher than the maximum bid. However, the solution of simply rerunning the failed job on an on-demand instance might not be ideal in the context of long-running services or tasks.

Despite that tight coupling with specific components of the AWS ecosystem is undesirable, the close integration does bring quite a few advantages in terms of storage flexibility. StarCluster can use S3 and EBS interchangeably, with data from mounted EBS volumes being instantly accessible throughout the cluster.

2.3.2 Elasticluster

Elasticluster [56] is another open-source project aiming at simplifying managing clusters on cloud platforms. Although not as popular as StarCluster, it has similar goals and a focus on simplicity and is more generic by supporting all of Amazon EC2, Google Cloud and OpenStack. Since we do not intend to call Elasticluster's Python API from our code, we will focus on the functionality provided by the command-line tool.

Given the more generic approach, Elasticluster makes fewer assumptions about the intentions of the user and requires more details to be set via the configuration file. It supports all of SGE, Slurm, Torque and PBS as scheduling systems. It uses Ganglia [57] for monitoring and allows transparent configuration of its toolkit via Ansible [58]. Along with the division of nodes in a cluster in *frontend nodes* and *compute nodes*, the use of Ansible playbooks

allows combining different setups for master and slave nodes. Modularity is achieved by creating the specification of the different blocks of the cluster independently (*login*, *cloud*, *setup*, *cluster*, *storage*) and simply assembling them to deploy the instances.

Although not as feature-rich as StarCluster, Elasticluster offers the basic cluster management commands and some extra customisation options:

- `start` launches the cluster and automatically runs Ansible to configure it unless the user specifically requests not to.
- `stop` kills the cluster without checking whether it is currently in use, which makes the tool require extra precautions on behalf of the user when managing the lifecycle of the client application.
- `list-nodes` offers information about all nodes in a specific cluster.
- `setup` allows the user to efficiently reconfigure the cluster from scratch without having to restart it and wait for instances to be recreated. It simply runs Ansible with respect to playbooks that have been added or updated by the user.
- `ssh` and `sftp` are used to obtain control of specific machines, upload and download files to and from the cluster.
- `export` is used to save the local data about a cluster started from the current machine in a `zip` file. A sensible but useful option is the ability to also store the local private and public SSH keys to the file.
- `import` unzips and stores an exported cluster locally.

Although SGE is supported as in StarCluster, the main Ansible playbook we are interested is the one setting up Slurm, since it is the more popular, modern and actively maintained scheduling system. The Slurm playbook exports the */home* filesystem from the master node to all the slaves, ensuring shared storage. However, extra performance can be achieved by combining the Slurm playbook with the OrangeFS parallel virtual filesystem and mounting OrangeFS nodes as shared storage in the main cluster.

A disadvantage of Elasticluster is that it does not support the native storage systems for its supported cloud platform providers, since Amazon and Google cloud instances usually report much better performance when coupled with storage on the same platform. Instead, Elasticluster has default support for GlusterFS [59], Ceph [60] and the OrangeFS [61] filesystem. Although these options are performant, they suffer from not being as widely popular as the previously mentioned ones.

Elasticluster facilitates adding and removing nodes from the cluster, but does not provide the flexibility of StarCluster to dynamically control the load by provisioning instances on demand. This is a major disadvantage, since the responsibility for implementing the behaviour is transferred to the user, despite the fact that failure strategies would be more robust if incorporated within the tool itself.

2.3.3 CfnCluster

CfnCluster is a cluster management tool built by Amazon specifically to support running high performance computing clusters using the entire stack of cloud services provided by AWS. This command line tool is free and open-source on its own, but it delegates most of the logic for administering the cluster to paid AWS services such as Amazon SNS[8] [62], SQS[9] [63], CloudWatch [64] and Auto Scaling [65].

The default installation is based on a provided AMI that ships with standard tools used for AWS administration. The configuration file allows extensive customisation and the possibility to enable various features needed for a job submission cluster:

- A scheduler that can be any of SGE, Slurm, Torque and OpenLava [66].
- Shared filesystem by mounting an EBS volume on all nodes at a user specified location. Read-write access to resourced owned by the user in S3 is also enabled.
- Amazon Virtual Private Cloud (VPC) provisioning in order to operate a cluster whose networking layer is in complete control of the user and is in a logically isolated section of AWS.
- Security groups for a cluster-level firewall.
- Custom bootstrap actions that allow the user to run pre and post-startup configuration scripts on the master node.

Figure 8 illustrates the sandboxing feature that isolates the master and compute nodes into subnets of the VPC.

[8] Simple Notification Service
[9] Simple Queue System

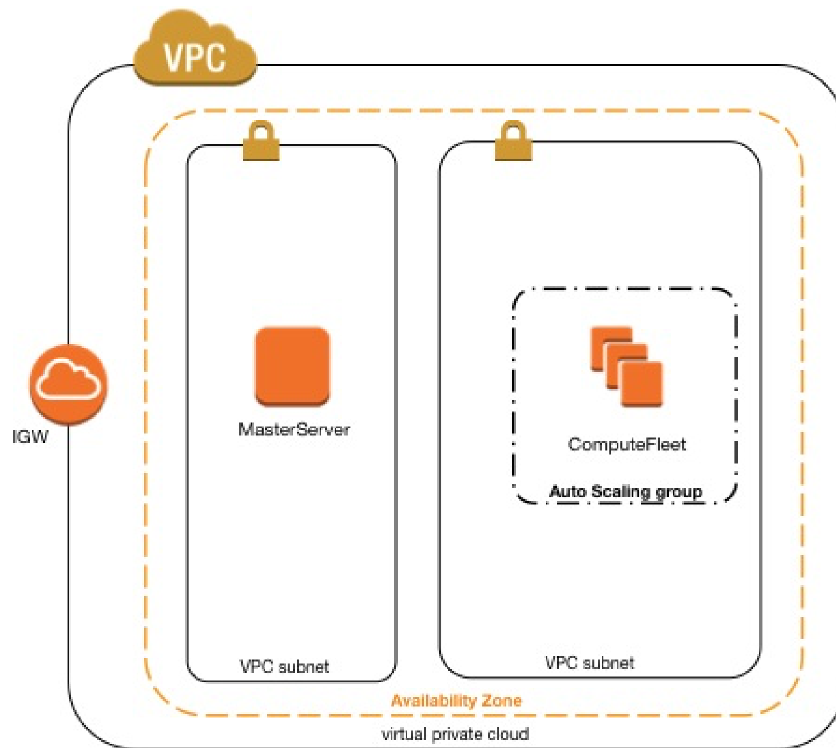
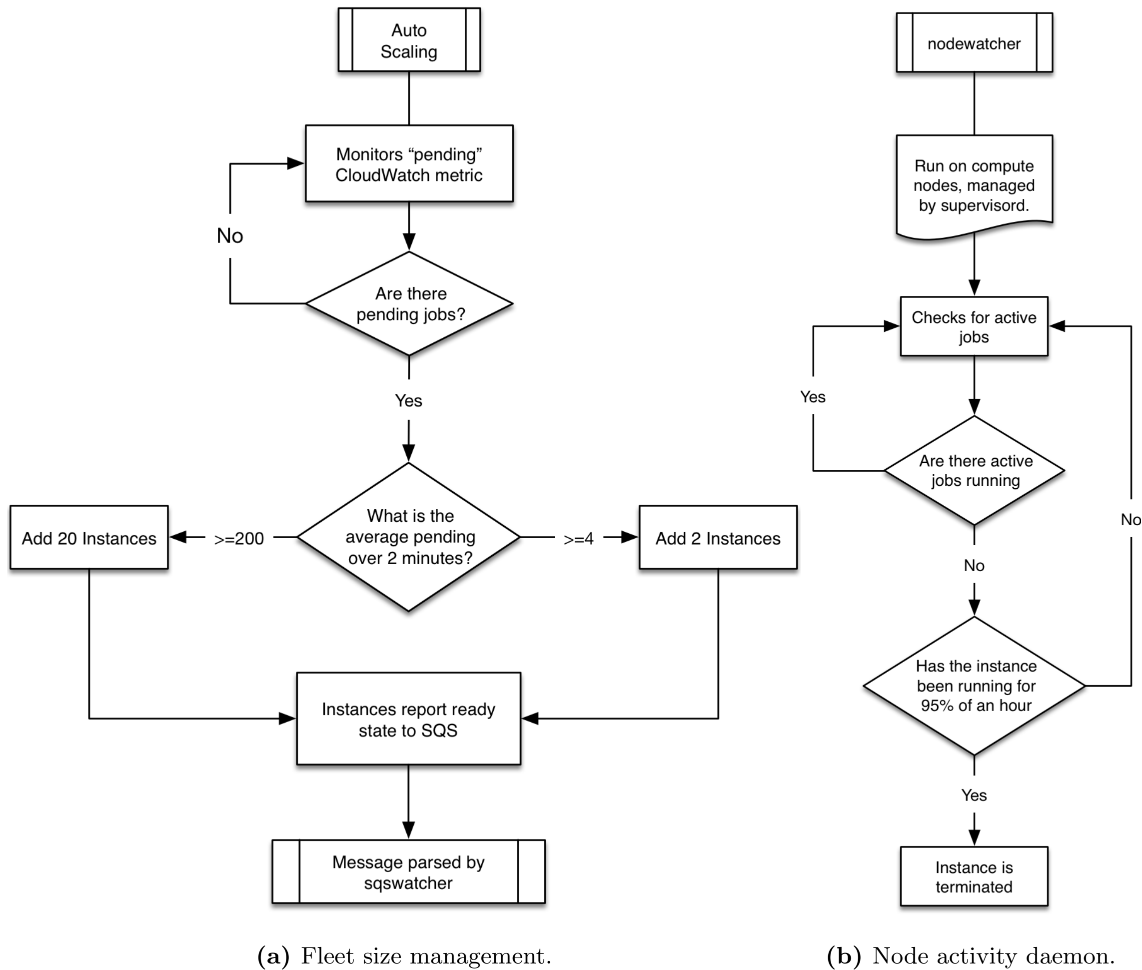


Figure 8: Master and slave nodes isolated in two different Virtual Private Clouds [67].

Elasticity is achieved through the interplay of various Amazon cloud services. On cluster creation, a CloudWatch alarm monitoring the number of pending jobs is created. When the alarm is triggered due to many queued jobs, instances are added by the Amazon Auto Scaling service up to a maximum defined by the `max_queue_size` configuration parameter. On the other hand, idle machines are identified using a `nodewatcher` daemon that monitors load and signals the possibility to reduce the compute fleet size to a minimum of `min_queue_size` instances. Figure 9 demonstrates the processes described above.

CfnCluster is a production-ready and reliable tool. However, it is completely one-dimensional since it focuses entirely on Amazon software and incurs extra costs by using many other AWS services, which could have otherwise been implemented as features of the tool itself. Although the costs of the individual cloud services supporting CfnCluster is not high, the overall cost is higher. An estimate for the added cost introduced by ClowdWatch alarms only is $3.50 per month per instance, compared to the $18.72 per month for renting a `t2.small` instance [68]. This implies an added cost of 19% that can by avoided by instead using StarCluster or Elasticluster.



(a) Fleet size management.

(b) Node activity daemon.

Figure 9: Auto Scaling monitoring diagrams [69].

2.3.4 Apache Jclouds

Apache Jclouds [70] is an open-source Java library that unifies the cloud services APIs for most mainstream commercial and open-source cloud providers, providing a starting point for an implementation that would be concerned with more fine-grained control of cloud specific features.

Although it requires more extensive configuration, it supports most commercial and open-source clouds (Amazon EC2, Google Cloud, Microsoft Azure, OpenStack, CloudStack), while not compromising highly provider-specific features. This is achieved by having highly flexible services for computation or advanced tasks like load balancing by using on-demand provisioning features of the clouds. For EC2, our main point of interest, Jclouds supports all type of storage provided by AWS, including EBS for low-latency services, S3 for low

costs and Glacier for long-term storage.

Jclouds currently provides three main abstractions that facilitate writing code agnostic to particular cloud vendor APIs:

- The *ComputeService* layer unifies services providing computational power offered by different providers like Amazon EC2 or Google Compute Engine. The abstraction builds on top of the native APIs provided by the vendors and offers various features:
  - The location aware API ensures that accessing resources placed in different geographic locations does not require maintaining multiple connections and managing multiple access objects. This permits a more clear organisation of code that oversees running virtual instances in multiple regions.
  - Basic cluster management support is offered by allowing the creation of groups of nodes instead of individual machines. Each group benefits from an automatically bootstrapped firewall that can be configured to regulate the traffic to the group as desired. Running scripts on all the machines in a group is also simplified through the existence of execution primitives and sensible exception types used to handle remote errors.
  - SSH key management for nodes launched using the API - specified SSH keys are copied to instances on startup.
  - Focus on testability - the library provides a stub compute service that mimics the behaviour of a remote execution environment for each provider, simplifying the task of writing tests for the top-level application.
- The *BlobStore* abstraction provides a portable way of accessing key-value stores such as Amazon S3, Google Cloud Storage or Microsoft Azure Blob Storage. Features are also location-aware and focus on unifying methods of accessing different types of storage:
  - The *filesystem* provider can be used to write data to memory, local disk or cloud blob storage using the same API.
  - The *in-memory* provider allows testing without the need for real credentials by stubbing in the memory of the local machine for the remote storage.
- The *LoadBalancerService* abstraction is a beta feature that aims to distribute the workload among nodes created in groups.

Listing 1 shows an example of using the `ComputeService` abstraction. After the access object is built from the provider preference and credentials, launching the instance from

the template configuring options such as the operating system, memory requirements or opening specific ports for external communication.

```
1  ComputeService compute = ContextBuilder.newBuilder("aws-ec2")
2      .credentials("identity", "credential")
3      .buildView(ComputeServiceContext.class)
4      .getComputeService();
5
6  Template template = compute.templateBuilder()
7      .osFamily(OsFamily.UBUNTU)
8      .minRam(2048)
9      .options(inboundPorts(22, 80))
10     .build();
11
12 compute.createNodesInGroup("jclouds", 1, template);
```

Listing 1: Instantiating an Amazon EC2 instance using ComputeService [70].

Obtaining the access object for a `BlobStore` is similar. Uploading data to the blob requires first creating a container and is straightforward after input data is parsed.

```
1  blobStore = ContextBuilder.newBuilder("aws-s3")
2          .credentials("identity", "credential")
3      .buildView(BlobStoreContext.class)
4          .getBlobStore();
5
6  blobStore.createContainerInLocation(null, "test-container");
7
8  ByteSource payload = ByteSource.wrap("test-data".getBytes(UTF_8));
9  blob = blobStore.blobBuilder("test")
10     .payload(payload)
11     .contentLength(payload.size())
12     .build();
13
14 blobStore.putBlob("test-container", blob);
```

Listing 2: Uploading data to Amazon S3 using BlobStore [70].

Jclouds is a useful toolkit for managing computing and storage resources rented from different cloud providers without delving too deep into vendor libraries. However, it does not provide full cluster functionality, such as installing and configuring a job scheduling system or a shared filesystem. Implementing the whole cloud access layer using Jclouds

would require duplicating a large part of the logic existing in tools already tested in production settings like Elasticluster, Starcluster or CfnCluster.

2.3.5 Mesos

Mesos [71] is an open-source cluster management system that provides abstractions of resources like processing power, memory or storage gathered from individual machines. Its aim is to offer a platform for creating safe, scalable and fault-tolerant distributed systems while programming against a simple API that allows treating an entire fleet of cloud instances as a shared supply of resources.

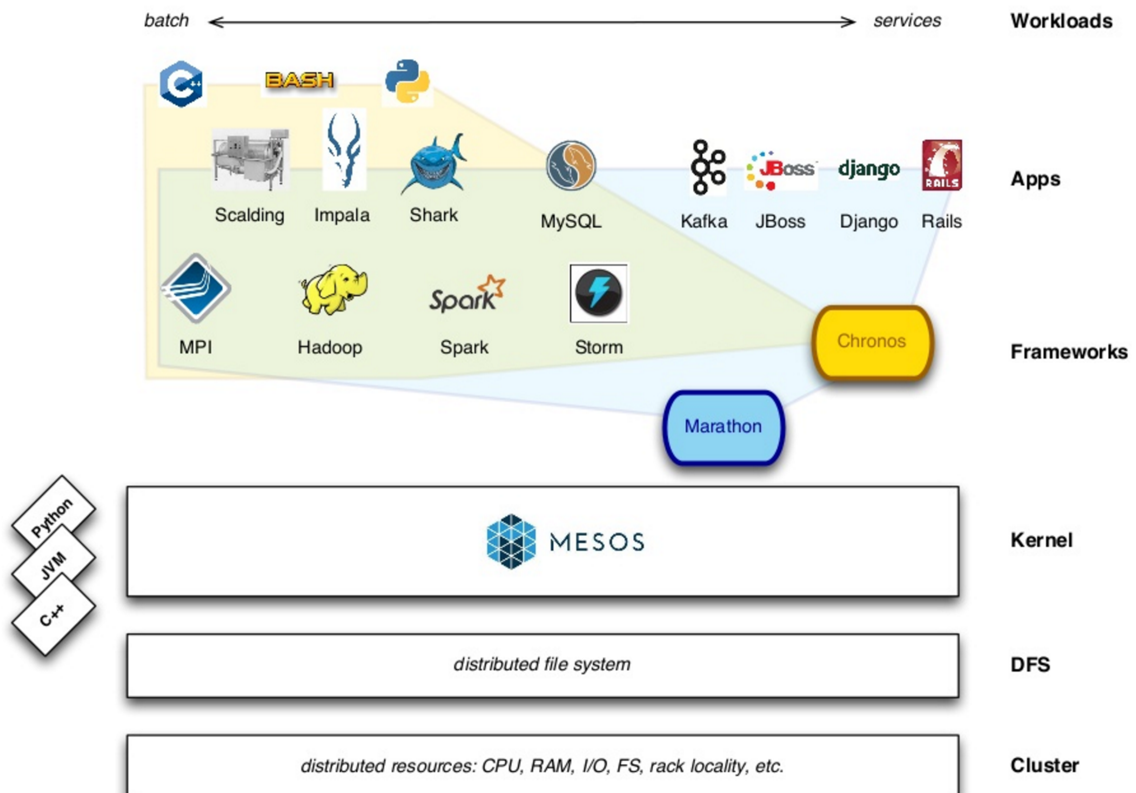


Figure 10: Mesos architecture [72].

Mesos can be described as a kernel for distributed systems. Its architecture is similar to the one of the Linux kernel, but revolves around working with distributed resources. Applications that take advantage of the APIs exposed by Mesos are known as frameworks and common examples include platforms for batch job scheduling, processing big

data, managing data storage or hosting long-running jobs such as web servers. Figure 10 shows the how Mesos mediates the access of high-level frameworks to fragmented low-level resources.

The key advertising points for Mesos are its scalability, tested in commercial applications to up to 10.000 nodes, and its layered architecture that supersedes classic resource scheduling and distribution systems that maintain the whole state of the underlying infrastructure. Instead, Mesos completely handles resource unification and only requires blocks on top of it to deal with primitives such as resource offers and describing tasks that need to be executed on slave nodes in a cluster.

Chronos

Chronos [73] is a job scheduler that aims to be a modern redesign of the classic Unix utility Cron [74]. Although Cron is mainly intended for performing single repetitive tasks and is typically used for scheduling system administration and maintenance tasks at regular time intervals, Chronos expands the feature space by also supporting job execution chains where jobs are triggered by finalisation of their dependencies.

Chronos is built as a framework on top of Mesos and it benefits from the fault tolerance and scalability provided by the underlying distributed kernel and resources.

Figure 11 shows how Chronos fits in a system where it communicates with the Mesos master to allocate resources, which are then used to execute designated jobs. In the case when a Mesos cluster is deployed on AWS, the tasks leverage highly segregated compute power to perform big data processing using Elastic MapReduce [75] and storage backed up by S3.

Chronos can aggregate statistics regarding the status of scheduled jobs batches, as well as details on individual executions. It also provides a web dashboard that can be used to manage jobs interactively, but our interest lies along the REST API for automation purposes. The simplest way to deploy and configure Chronos along with Mesos is by using Mesosphere DC/OS [76].

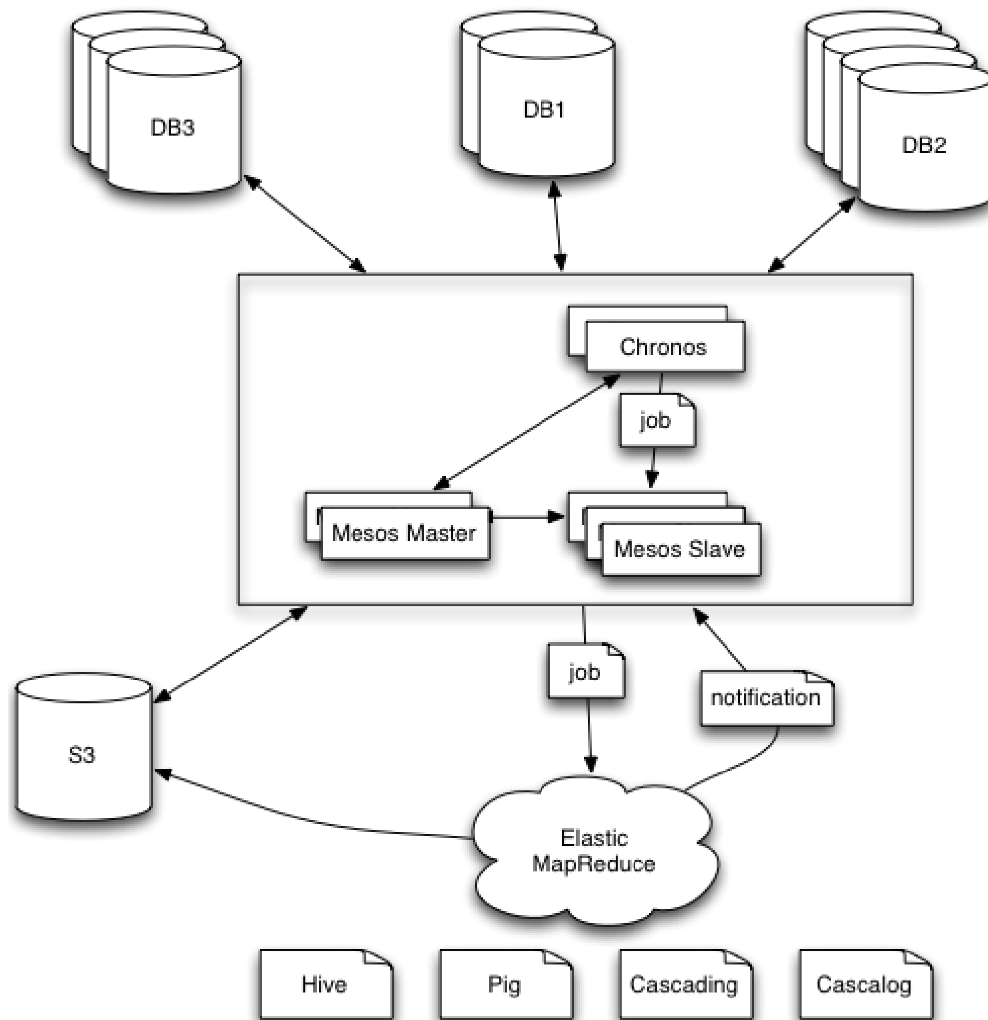


Figure 11: Chronos architecture [73].

Mesosphere DC/OS

DC/OS is a distributed operating system using Mesos as its kernel and it facilitating the installation and orchestration of frameworks consuming the Mesos API. Figure 12 shows the analogy between this setup and operating systems running on the Linux kernel.

DC/OS comes with out-of-the-box support for configuring clusters relying on instances provisioned from AWS and Microsoft Azure. By having DC/OS install Chronos, we can start executing jobs remotely using AWS resources. The community edition of the product is fully open-source and free to be used for non-enterprise purposes. However, for

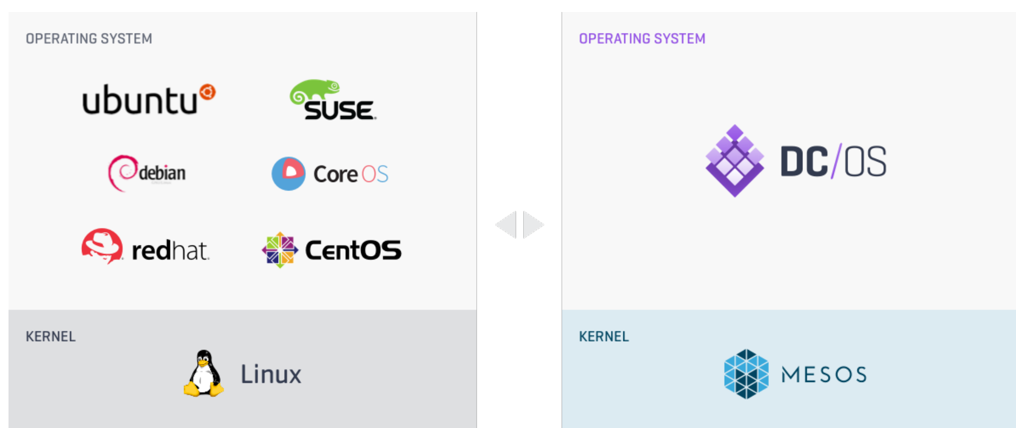


Figure 12: Mesosphere DC/OS architecture [76].

our use case this solution suffers from the same problem as CfnCluster, namely the extra cost induced by delegating all the elastic scaling work to services like Amazon Auto Scaling.

Although DC/OS would not add extra costs directly, it does require more powerful bootstrap, master and agent nodes than either of Elasticluster, Starcluster or CfnCluster. Table 1 shows the hourly price of operating an Amazon EC2 cluster with the minimum required specifications. In comparison, the cost of a `t2.small`— instance that can be used as both a master and compute node in a Starcluster or Elasticluster installation with EBS volumes mounted for shared storage is $0.026 per hour. This leads to a 10x cost increase only for operating a single master node and ignoring the fact that DC/OS in fact recommends running 3 simultaneous master nodes for optimal performance[10].

Node role	Required instance	Price ($ per hour)
Bootstrap	m3.2xlarge	0.532
Master	m3.xlarge	0.266
Agent	r3.large	0.166
		0.964
		Total

Table 1: Prices of EC2 nodes with DC/OS minimum performance requirements [68, 77].

In conclusion, DC/OS is a highly reliable tool for operating clusters in the cloud, but from cost-effectiveness perspective its features only shine when used at massive scale. In our use case, where jobs are more numerous rather than particularly CPU-intensive, Starcluster and Elasticluster are better candidates and can produce similar results for a lower investment.

[10]Prices as of June 2016 for instances in US East

Chapter 3

OpenMOLE & GridScale

The first part of this chapter expands on the goals, features, architecture and components of the open-source OpenMOLE project. We also explore use cases of the system by looking at a simple workflow. The second part of the chapter describes the structure and design of GridScale, the library OpenMOLE relies on to leverage distributed computing resources from grids and clusters.

3.1 OpenMOLE

OpenMOLE [78] is a workflow execution engine that focuses both on allowing expressive definitions of data processing pipelines and delegation of those tasks to remote execution environments [79]. Compared to other existing workflow management systems like Kepler [5], Taverna [4], Galaxy [7] or Pegasus [6], it does not target a specific scientific community and instead aims at offering formalisms that can be used to create generic pipelines.

One of the main objectives of OpenMOLE is embedding workflow models and definitions provided by users in many different forms [8]. Generally, other workflow engines have rigid, text-based rules used to describe individual tasks and their connections, while providing limited support for calling external programs.

However, OpenMOLE uses a DSL[1] built on top of Scala [13] to embed a wide variety of tasks defined in any programming language based on the Java Virtual Machine (Java, Scala, Clojure, Groovy, Kotlin). Additionally, prepackaged binaries, C++ executables and Python scripts that depend on shared libraries or pre-loaded packages are seamlessly integrated into workflows. Benefits of relying on Scala's type system include more mean-

[1] Domain Specific Language

ingful task descriptions and early error detection since potential mistakes are caught at compile-time, instead of only after submission to grids or clusters.

Possible execution environments include the user's local machine, SSH servers, grids or self-hosted clusters operated by one of the many supported schedulers: SGE [17], Slurm, [18] PBS [15], HTCondor [19], OAR [20] or Torque [16]. These are all enabled by GridScale [25], the self-contained library that is shipped by default with OpenMOLE and handles job management on distributed computing environments.

The OpenMOLE platform is now mature and has engaged a loyal user base. It is regularly used for large scale experiments and its robustness in combination with GridScale was proven by experiments where it has been used to run half a billion tasks on EGI[2] [80].

3.1.1 Architecture

The design of the application has been guided by the total decoupling between the creation of the pipelines describing the scientific algorithms and their execution on remote environments [79]. As shown in Figure 13, this led to a layered structure, where the actual experiments are independent from how the tasks they incorporate are run and managed or how the resource requirements are serviced.

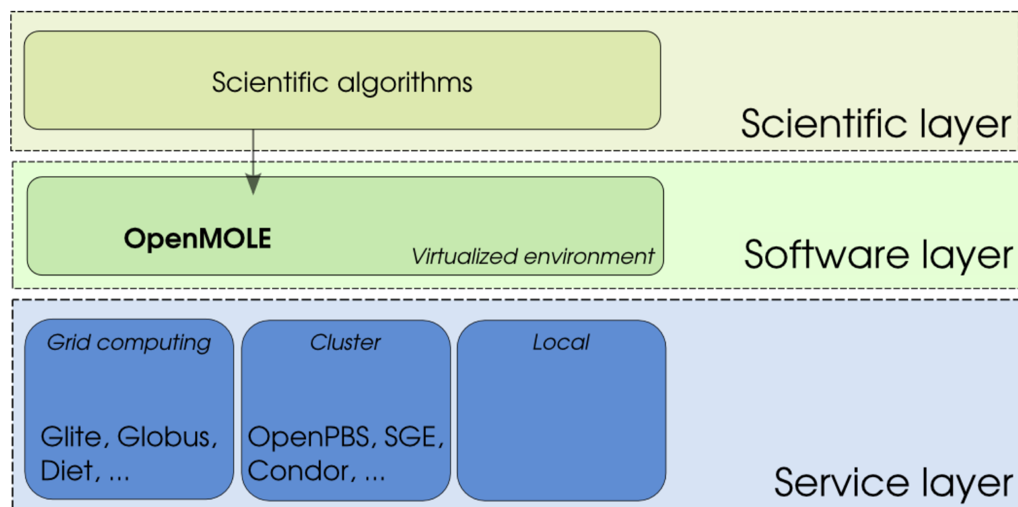


Figure 13: OpenMOLE layered architecture [9].

In this context, the user is not concerned with how the execution environment is provi-

[2]European Grid Initiative

sioned and can change it easily without altering the workflow description. Additionally, this allows for a fine granularity of task distribution since individual tasks can be sent to environments tailored specifically for their requirements. A clear use case for this are jobs that need to be accelerated using GPUs, with other parts of the workflow running on CPUs as usual.

The structural layers fulfil completely different roles and are modular blocks with clearly defined roles:

- The scientific layer relies on the DSL. It is where the user defines the workflow and specifies data sources and execution environments.
- The software layer is concerned with the translation of the workflow specification to an internal representation consisting of runnable jobs and their dependencies. It uses the Scala actor model [81] to coordinate the whole process of installing the OpenMOLE runtime on each remote execution site, continuously monitoring the status of outstanding jobs, or feeding results of completed tasks to the subsequent ones using them as input. New remote environments are simply added as plugins and they are essentially adapters to the resources exposed by the service layer.
- The GridScale library is the service layer, presented in Section 3.2.

3.1.2 Domain Specific Language

Tasks are the basic construct of workflows in OpenMOLE. They essentially consist of a computation that is run against a set of *inputs* to produce a set of *outputs*. Together with *transitions*, which transfer outputs of individual tasks to inputs of subsequent ones, they establish the simplest form of data pipeline.

Listing 3 shows a very basic workflow that computes the square of each number from 1 to 100. It begins by declaring variables `i` and `res`, which represent the dataflow and are used to carry results between tasks that are connected via transitions. Since the DSL is build as a set of extensions on top of the Scala programming language, the variables are statically typed, which enables formal verification of the workflow at compile time [8]. This ensures that runtime type mismatches of data transmitted between tasks cannot occur.

Lines 6-10 define the actual model that we want to compute. In this case, the specification is a `ScalaTask`, where for each input of type `i` the output is a tuple `(i, res)`. Concretely, for input `5`, the output will be `(5, 25)`. Apart from Scala code, a `ScalaTask` can embed

```
val i = Val[Int]
val res = Val[Int]

val exploration = ExplorationTask(i in (1 to 100 by 1))

val model =
  ScalaTask("val res = i * i") set (
    inputs += i,
    outputs += (i, res)
  )

val env = LocalEnvironment(8)

exploration -< (model on env hook ToStringHook())
```

Listing 3: Simple OpenMOLE workflow.

code from any programming language running on the JVM (Java, Clojure, Groovy). Other types of tasks include `MoleTask`, used to embed entire predefined workflows, or `CARETask`, used to run any type of prepackaged binary and discussed in section 3.1.5.

One of the central objectives of OpenMOLE is to allow parameter optimisation for the models it executes. The sampling of parameters is achieved through the use of *explorations* in the language. Every parameter sampled from a set is combined with all the other parameters to generate the workflow input. In our case, the exploration at line 4 only samples parameter `i`, so the model will be replicated 100 times and executed with an integer between 1 and 100 as input.

Line 12 instantiates the environment where the workflow will be run. Here we select a `LocalEnvironment` and allow the computation to create and use 8 threads on the local machine. Line 14 ties all the building blocks together. The exploration generates all possible values of `i` and the model is executed on the given environment for each one of them. The `-<` symbol represents a divergent transition between the exploration and the task, each output of the sampling being fed to the model.

In OpenMOLE, *hooks* are used to extract the results of running an experiment. They are executed every time the task that they are assigned to terminates and can perform actions like simply displaying the data or saving it to CSV[3] files. Here the `ToStringHook` simply prints the output of each task.

[3]Comma-separated values

Figure 14 shows a graphical representation of the combination of primitives described above. This is how a workflow is designed using the graphical user interface instead of the specification language.

Figure 14: OpenMOLE workflow [78].

Going beyond the simple example, Listing 4 shows a sampling that explores all combinations of a discrete parameter `i`, a continuous parameter `j`, a file `f` from the given `workDirectory` and a random seed `s` sampled from a uniform distribution. This is achieved using the *x* combinator, which unrolls the domain for each parameter before combining all the possibilities, essentially taking a cartesian product of the parameter sets [82].

```
1  val i = Val[Int]
2  val j = Val[Double]
3  val f = Val[File]
4  val s = Val[Long]
5
6  val exploration =
7    ExplorationTask(
8          (i in (0 to 10)) x
9          (j in (0.0 to 100.0 by 10.0)) x
10         (f in (workDirectory / "inputs")) x
11         (s in (UniformDistribution[Long]() take 10))
12   )
```

Listing 4: Advanced exploration [79].

Listing 5 shows different transitions that can be used to combined task. Note that, for brevity, we did not include any hooks that would collect the results at any stage.

The execution on line 13 explores the parameter space and feeds each possible value sequentially through each of the 3 tasks. On line 14, `t2` and `t3` are run in parallel on the inputs received from `t1`. Line 15 demonstrates a convergent transition, where the `>-` operator is used to allow an aggregation task to collect and process the results of the run.

```
val i = Val[Int]

val t1 = ScalaTask("i = i * 2") set ( inputs += i, outputs += i )
val t2 = ScalaTask("i = i * 3") set ( inputs += i, outputs += i )
val t3 = ScalaTask("i = i * 4") set ( inputs += i, outputs += i )

val exploration = ExplorationTask( i in (0 to 100) )
val aggregate = ScalaTask("val i = input.i.sum") set (
  inputs  += i.toArray,
  outputs += i
)

exploration -< t1 -- t2 -- t3
exploration -< t1 -- (t2, t3)
exploration -< t1 -- t2 -- t3 >- aggregate
```

Listing 5: Transition types [82].

Our initial example showed the simple case of running a workflow on the user's machine using a `LocalEnvironment`. However, this is only usually done for testing locally before scaling the experiment and other types of environments are used for real experiments. Section 3.1.3

The user can be authenticated either with a login and password combination, or via a private key. Here we define the authentication by specifying the path to the private key, the associated login and the remote machine's full address. The `encrypted` parameter references a function that will prompt the user for the key's password in case it is protected. This process is done once and for all and stores the new authentication in OpenMOLE's preferences folder. It is consequently rarely seen in actual workflow scripts that will reuse an already defined authentication.

3.1.3 Environments

Remote execution environments are are the engines providing computational power for running experiments. Generally, environments set up on grids or clusters are created by providing a shell login on the remote machine as well as its address, but this complexity needs to be hidden from the user in the case of a cloud environment.

The end goal of the project is to provide an `AWSEnvironment`, which can be instantiated by only receiving the user's AWS credentials as parameters. This should automatically create a cluster backed up by EC2 instances and configure it with a scheduler that distributes job submissions generated by the workflow.

Listing 6 shows examples of instantiating SSH servers and cluster environments. For the SSH server, we also specify the number of cores that can be used by the workflow. Cluster environments require that the target machine can act as the master of the cluster, being able to take commands for submitting and querying job status.

```
SSHAuthentication +=
  PrivateKey(
    "path/to/private/key",
    "login",
    encrypted,
    "machine-address")

val sshEnv = SSHEnvironment("login", "machine-address", 8)
val condorEnv = CondorEnvironment("login", "master-address")
val sgeEnv = SGEEnvironment("login", "master-address")
```

Listing 6: Usage of various environments.

In OpenMOLE, environments are added through a plugin system by implementing the `BatchEnvironment` trait. This comes with a built-in `JobManager` and various other defaults useful for modelling a generic job submission environment. `ClusterEnvironment` is wrapper that ensures steady traffic flow to a cluster by limiting the number of outgoing connections and adds an interface for accessing storage via SSH.

Each concrete environment in OpenMOLE has a corresponding job service. The low-level mechanics of interacting with the scheduler and running individual jobs remotely are abstracted away in GridScale, so job services attached to environments decorate this

behaviour with batch submission capabilities.

```
1  trait SGEJobService
2      extends ClusterJobService with SSHHost with SharedStorage { self ⇒
3
4    def environment: SGEEnvironment
5    val jobService = new GridScaleSGEJobService with SSHConnectionCache {...}
6
7    protected def _submit(serializedJob: SerializedJob) = {
8      val (remoteScript, result) = buildScript(serializedJob)
9      val jobDescription = new SGEJobDescription {
10       val executable = "/bin/bash"
11       val arguments = remoteScript
12       val workDirectory = serializedJob.path
13       override val queue = environment.queue
14       override val wallTime = environment.wallTime
15       override val memory = Some(environment.requiredMemory)
16     }
17     val jobId = self.jobService.submit(jobDescription)
18     ...
19   }
20 }
```

Listing 7: Job service used to submit batch jobs to the SGE scheduler.

Listing 7 shows the core of the `SGEJobService`. Note that objects used to manage individual jobs at the underlying GridScale level are also called job services, so we must always consider the distinction. On line 8, a shell script embedding the task to be run is created from a serialized job. On line 11, the script is assigned to be run by `bash` as part of an `SGEJobDescription`. The job is then submitted to the underlying GridScale job service on line 17.

3.1.4 Job Distribution

Compared to other workflow platforms, OpenMOLE follows a zero-deployment approach, meaning that it does not rely on any software being installed on the target machines that the task-generated jobs will run on [8]. In order to support this, a setup phase is required before the task execution step itself. This involves uploading several components to the remove environment [9]:

- The OpenMOLE runtime, which includes the OpenMOLE framework and the Java

Virtual Machine used to run it.

- Task descriptions along with their serialized execution contexts, which describe variables used by the task to transport data.
- Resource files used by the tasks.

After all the dependencies are in place, a job is packaged to reference the runtime, task and context it corresponds to. Once it is assigned to an execution node by the environment's native submission system, it downloads the runtime and runs the task in the given context.

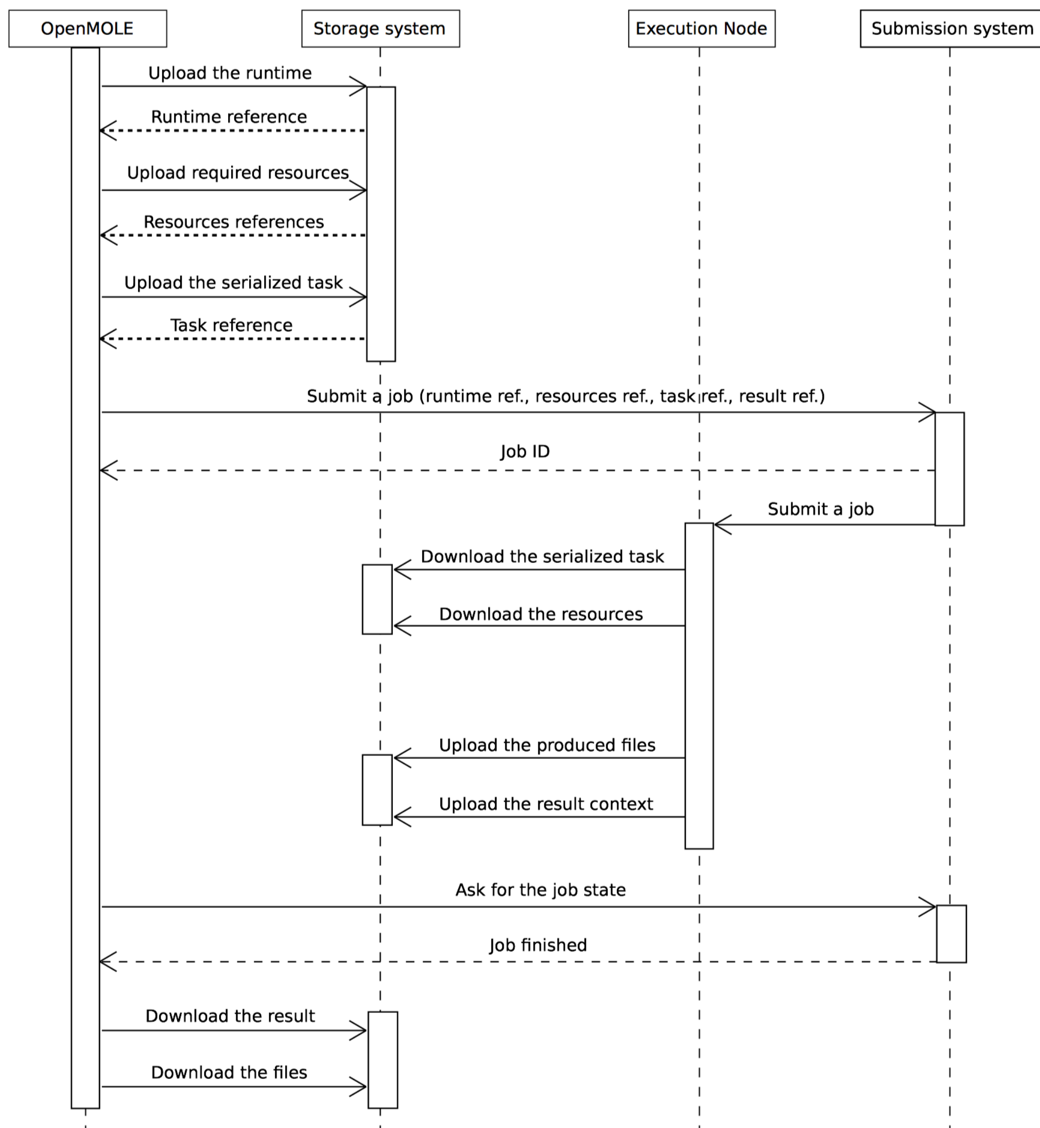


Figure 15: Delegation of a task to remote execution environment in OpenMOLE. [9].

Note that the potentially expensive step of copying the runtime on each node is rarely necessary in practice, since clusters and grids usually operate on filesystems shared across all nodes. OpenMOLE also maintains a cache of the file replicas already uploaded to remote storages, so that a file never gets uploaded twice to the same storage as long as it's not been modified on the host system.

Once finished, jobs upload their results to the storage system. Meanwhile, the OpenMOLE framework continuously tracks the state of the job by querying the submission system and downloads the outputs on the local machine of the user upon completion. The whole process is presented as a sequence diagram in Figure 15.

3.1.5 CARE

Resolution of dependencies is a classical problem in the context of job distribution to remote environments, especially in the case of having little to no control over their configuration. Compiled binaries like C++ applications require shared libraries at runtime, while interpreted languages depend on various packages to be present on the execution node.

Users of grids and clusters do not usually have access to install the needed dependencies. Grids are particularly tricky, since they represent shared pools of heterogeneous resources that are likely to be configured and deployed differently than on a local machine that a workflow is initially tested on [79].

The first partial solutions were application specific and not always feasible. For example, C++ programs can be built as static binaries that package all the dependencies, but this is a problem in the case of applications using proprietary libraries without publicly available source code.

A practice that has recently gained traction in software engineering communities particularly via Docker [83] is the use of containers. A container consists the entire software stack required to run an application, including dependencies as packages, binaries or configuration files. Containers accomplish a similar purpose to virtual machines, with the main advantage that they are designed to be lightweight. They are smaller in size than full-fledged virtual machine images, can be started faster and numerous instances can be hosted by a single operating system, making their deployment straightforward in comparison with the hypervisor configuration required to host virtual machines.

However, the use of Docker containers also presumes the existence of the Docker engine on the target machine and this can not be ensured for environments over which scientists only have user access. This leads to the choice for CARE [10], an open-source application for reproducible executions that only relies on being run on a Linux platform. CARE works by intercepting all the requirements of an application during an initial run and repackaging all the dependencies in a self-extracting binary.

CARE also ensures full interoperability between packaging and execution environments, meaning that all modern Linux-based operating systems support running archives packaged under a different Linux distribution. This allows OpenMOLE to remotely distribute the application packaged on the user's machine without concerns about the particularities of Linux flavours present in a grid or cluster.

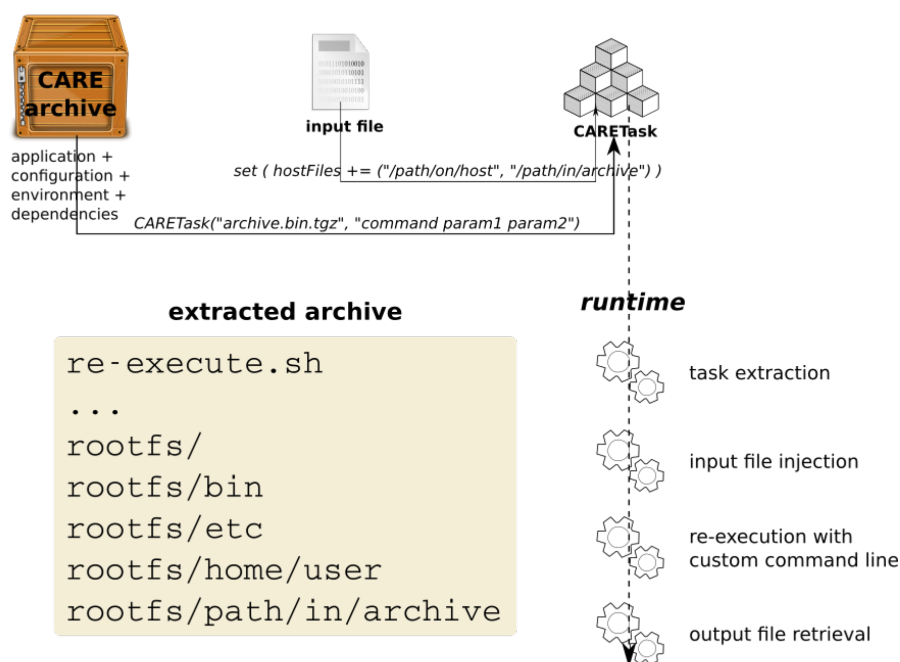


Figure 16: Delegation of a task to remote execution environment in OpenMOLE. [79].

More specifically, binaries are embedded in OpenMOLE via a `CARETask`, which takes as parameters the location of the archive and the specific command that needs to be executed in the packaged runtime. OpenMOLE then runs the given command in the unpackaged directory on the remote node, copying the input files in the process as usual. Figure 16 depicts the interplay between a CARE archive created by the user and the `CARETask` in OpenMOLE. Listing 8 shows a sample task embedding a Python script.

```
val output = Val[String]
val error = Val[String]
val value = Val[Int]

val pythonTask =
  CARETask("hello.tgz.bin", "python hello.py /data/fileA.txt") set (
    stdOut := output,
    stdErr := error,
    returnValue := value,
    hostFiles += ("/home/user/fileA.txt", "/data/fileA.txt")
  )
```

Listing 8: Example Python CARETask [79].

3.1.6 Job Management

Most of the logic of coordinating the execution of jobs and collection of results is handled by a `JobManager` class. It uses a message queue to direct commands to separate actors responsible for uploading resources to execution nodes, submitting, querying or purging jobs, as in Listing 9.

```
class JobManager {
  ...
  def !(msg: JobMessage): Unit = msg match {
    case msg: Upload   ⇒ messageQueue.enqueue(msg)
    case msg: Submit   ⇒ messageQueue.enqueue(msg)
    case msg: Refresh  ⇒ messageQueue.enqueue(msg)
    ...

    case Manage(job) ⇒
      self ! Upload(job)

    case Uploaded(job, sj) ⇒
      job.serializedJob = Some(sj)
      self ! Submit(job, sj)

    case Resubmit(job, storage) ⇒
      killAndClean(job)
      job.state = ExecutionState.READY
      messageQueue.enqueue(Upload(job))
    ...
}
```

Listing 9: Job lifecycle management.

The message queue is consumed iteratively by a unique dispatcher actor that routes messages to the specialised receiver actors. This model ensures a clear separation between individual actions carried out to monitor the jobs.

3.2 GridScale

GridScale is the library part of the OpenMOLE ecosystem that mediates the access to distributed computing environments. Being written in Scala, it can be used as part of any application running on the Java Virtual Machine and it is designed around the strict type system of the Scala programming language. This leads to improved safety checks for job definitions at compile time, since their members can now be more refined than plain strings [25].

3.2.1 Principles

Historically, the scientific community has relied on specifications developed by the OGF[4] [84] to establish the guidelines for libraries used to access grids or clusters. However, a problem that standards like DRMAA[5] [85] or SAGA[6] [86] encounter is that they require the commitment of multiple parties interacting with an environment. Users need to implement a particular protocol that matches the version of the specification deployed by the administrators of the infrastructure. Additionally, the API is often slow to evolve and inflexible to user requirements.

GridScale chooses to stay away from particular standards and favours an approach where very few assumptions are made about the configuration of the target infrastructure. In particular, it only requires that the accessed environment runs a Linux distribution and the user has access to the *bash* shell via SSH. This is reasonable to expect from most machines, since *bash* is the default in most cases and a login shell is not needed.

Jobs are submitted and monitored using the standard command line tools that would be manually invoked by the user. For example, SGE jobs are managed using `qsub`, `qstat` and `qdel`, while Slurm jobs are managed using `sbatch`, `scontrol` and `scancel` for submission, state querying and termination, respectively.

[4] Open Grid Forum
[5] Distributed Resource Management Application API
[6] Simple API for Grid Applications

3.2.2 Module Design

Each environment accessed by GridScale is serviced by its own module in the implementation. Modules are packaged as independent OSGi[7] [87] bundles so that they can be included individually by applications servicing only a specific infrastructure. This enforces a modular design and reduces the footprint of the library [25].

Every module corresponding to an environment consists of four different components wired together into a unique block using the cake pattern [88]:

- A *Job Description* used to define the executable, arguments and other parameters for the job.
- A *Job Service* responsible for the job submission mechanisms and resource acquisition.
- A *Storage* component that can be shared across modules.
- An *Authentication* rule for granting access rights using a login and password combination, SSH or certificate authentication.

Figure 17 illustrates the creation of a Slurm module from particular implementations of the four component interface. Although the underlying technique in the cake pattern is different, it achieves a similar result to standard dependency injection in languages like Java by offering modules easy to assemble.

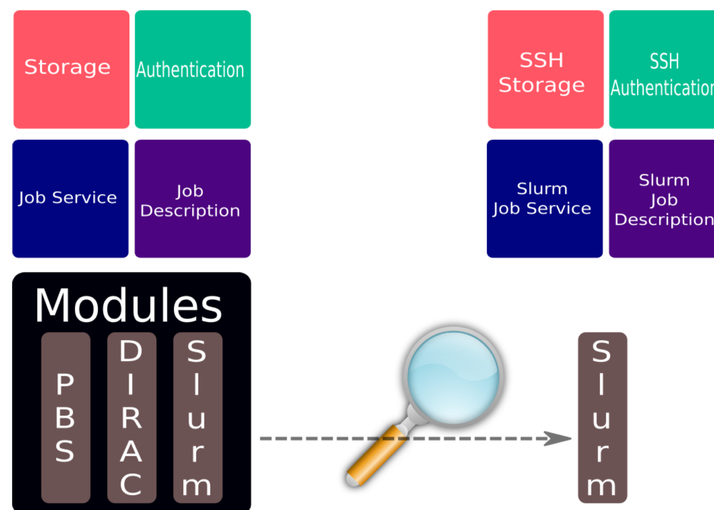


Figure 17: GridScale architecture and instantiating the Slurm module [25].

[7] Open Service Gateway Initiative

Job Description

Job descriptions are typesafe wrappers in Scala that translate to the plaintext scripts accepted by schedulers to queue jobs. The main advantage is that this ensures syntactical correctness of jobs and errors that would prevent jobs from being submitted are caught early. Silent issues such as misusing measurement unit for time or resources are also avoided by relying on a strong type system.

Listing 10 compares the definition of a simple Slurm job using the standard approach in a text `bash` script and its equivalent counterpart in GridScale. Note that, even for a relatively short description, the plaintext version can easily become ambiguous in sections where durations or paths are specified.

```bash
#!/bin/bash
#SBATCH -o job.out
#SBATCH -e job.err
#SBATCH --cpus-per-task=1
#SBATCH --time=01:00:00
#SBATCH -D /home/foo/bar

/bin/echo success
```

```scala
val description = new SLURMJobDescription {
  def executable = "/bin/echo"
  def arguments = "success"
  def workDirectory = "/home/foo/bar"
  override def wallTime = 1 hour
}
```

Listing 10: Comparison between a Slurm job description in plaintext and in the OpenMOLE DSL.

Job Service

The job service is the component that manages the lifecycle of jobs. Listing 11 shows the creation of a job service capable of interacting with Slurm. To connect to the remote environment, the service relies on the presence of a target host and login, along with a possibly password-protected SSH private key.

```scala
1 val slurmService = new SLURMJobService with SSHPrivateKeyAuthentication {
2   def host = "host"
3   def user = "user"
4   def password = "password"
5   def privateKey = new File("/path/to/private/key")
6 }
```

Listing 11: Job service used to submit batch jobs to the Slurm scheduler.

The interface exposed by a job service consists of four main methods: `submit`, `state`, `cancel` and `purge`. These allow initiating jobs, checking their status, terminating them and cleaning up temporary data.

In contrast with other libraries used for distributed resource management that retain caches of the system status, GridScale job services promote a functional approach and are designed to be immutable and hold as little state as possible. Therefore, all the public methods are pure and simply forward the request to the remote scheduler.

Storage

Storage wrappers abstract operations with files on the target environment. They are used by job services to set up the runtime by uploading files to remote machines and download results when jobs are done.

Standard POSIX file operations are also enabled in order to complement the usage of command line tools to delegate work. The implementation of all the connections and commands relies on the SSHJ [89] library, which handles SSH connections and provides the SFTP[8] primitives directly to Java or Scala.

Authentication

The authentication component provides access to all environments covered by GridScale. Private clusters are managed by opening an SSH connection to the master node, so the authentication consists of the address of this machine, the name of the user and a password or private key.

On the other hand, grids often require authentication methods that cover the security model of the middleware managing them. This is achieved in GridScale by allowing the installation of P12[9] and PEM[10] certificates [25].

[8] Secure File Transfer Protocol
[9] PKCS - Public-Key Cryptography Standards
[10] Privacy Enhanced Mail

Chapter 4

Implementation

This chapter describes the design and implementation of a new service adding support for running workflows on clouds, with the specific target infrastructure being AWS EC2 machines. The deployment, configuration and lifecycle management of the underlying nodes executing the workflow should be transparent to the user, who is only required to provide his credentials to an AWS account.

In accordance with the considerations presented in Chapter 3 about the architecture and implementation details of both OpenMOLE and GridScale, we take a layered approach in adding support for a new cloud environment.

The first part of the chapter discusses the approach taken to bootstrap and configure a cluster consisting of EC2 machines. The cluster needs to be coordinated by a scheduler that can take job submissions and distribute them for execution.

We then proceed to analyse the integration of the cloud service with the overall OpenMOLE application. Here we treat issues such as triggering the teardown of the cluster depending on the batch submission activity and the inherent differences between the new cloud environment and the already existing types of environments.

Although we focus on deployment on AWS, the same high-level technique can easily be extended to support other cloud providers, so at the end of the chapter we briefly discuss the details of adding support for the Google Compute Engine infrastructure.

Throughout the chapter, we motivate choices made along the development process and detail problems or advantages of alternative approaches. Since the main challenge of the implementation is reliably wiring together different libraries, command line tools and cloud services, we favour the use of diagrams and code listings to show the interaction between core system components.

4.1 GridScale AWS Module

As explained in Section 3.2, access to each target environment in GridScale is implemented via a module. For the purpose of implementing an AWS module, we need to provide each of the four components: a *Job Description*, a *Job Service*, a *Storage* interface and an *Authentication* method.

4.1.1 Design Motivation

The initial idea was to simply implement an AWS specific version of each component. This is though not necessary, as we can reuse some of the already existing components from other modules and GridScale in fact encourages this approach through its modular packaging using OSGi bundles.

In particular, GridScale already supports cluster environments managed by specific schedulers. By building a cluster from a set of machines provisioned from AWS and configuring to emulate the required behaviour, we are able to partially delegate the job management work to an already functional cluster job service. This means that, in contrast with a regular cluster service that acts as a DAO[1] for an existing infrastructure, we also need to provide interface methods through which users of the library can manage the lifecycle of the deployed cluster. We argue that this is a sane choice from an engineering perspective, since we are already familiar with reliably supported schedulers like SGE and Slurm.

The type of the job description depends on the job service, since it performs a translation to a script that can be sent to the scheduler. Therefore, forwarding the submission and monitoring methods to a cluster service as discussed above implies that we need to directly use the description corresponding to the job service.

As long as we can deploy a cluster with a filesystem shared across the network, the `SSHStorage` trait allows accessing the same data volume for the master and slave nodes. EC2 instances can be backed up by either EBS volumes or internal storage, but the solution of installing NFS on the cluster is agnostic to the underlying storage choice and enables using a wide range of instance types.

Authentication is indeed based on the AWS credentials stored locally by the user and

[1] Data Access Object

requires a new model. However, the standard authentication implementation relying on SSH keys can still be reused as a lower layer of this new implementation in order to access the master node that controls the job submission queue.

External applications make use of the AWS module by instantiating an `AWSJobService`, which publicly exposes the following methods:

- `start` launches and configures a job submission cluster to be used by the service.
- `close` completely tears down the job cluster.
- `submit`, `cancel`, `state` and `purge` fulfil their regular functions within the service.

Listing 12 shows an example usage of the service. The service takes as parameters the region the job should run in, as well as the number of EC2 instances that the service should create under the `clusterSize` parameter. The other authentication-related parameters are discussed in Section 4.1.2.

```
val awsService = AWSJobService(AWSJobService.Config(
    region = "eu-west-1",
    awsUserName = "adrian",
    awsCredentialsPath = "/Users/adrian/.aws/credentials.csv",
    awsUserId = "434676269080",
    awsKeypairName = "openmole",
    privateKeyPath = "/Users/adrian/.ssh/id_rsa",
    clusterSize = 1))

awsService.start()

val description = new AWSJobDescription {
  def executable = "/bin/sleep"
  def arguments = "5"
  def workDirectory = aws.home
}

val job = awsService.submit(description)
while (awsService.state(job) != Done) {
  Thread.sleep(WAIT_TIME)
}

awsService.purge(job)
awsService.close()
```

Listing 12: Submitting a job to the cloud using the AWS module.

4.1.2 Cluster Deployment

The design of the module presented above outlines the requirement for a system that can create clusters of AWS EC2 machines and configure them with a job scheduler and a shared filesystem across all nodes.

Tool Choice

During the investigations in Section 4.1.2, we found that most cost-efficient tools and frameworks for creating the cluster are StarCluster, Elasticluster and Jclouds. The main reason for excluding Mesosphere DC/OS was that it is generally heavyweight and requires running more expensive machines to power the cluster. CfnCluster was dismissed for being too reliant on the AWS software stack, which incurs various extra costs for operations that can easily be performed without Amazon resources.

Jclouds allows mounting various cloud storage devices, but it did not fit our use case since it does not have built-in support for NFS installations. Although Elasticluster can perform all the required tasks, we opted for StarCluster as the cloud deployment tool, since it also provides a load balancer that can be used to optimize the cost or run time of a workflow.

Coordinator Node

The choice for StarCluster raises the problem that it is only offered as a command line tool and it needs to be on the machine it is run on. Although most cluster orchestration tools require manual installation and interaction, forcing the user of the library to install an external is both unreasonable and impractical from GridScale's perspective.

An initial idea could have been running a Docker container with StarCluster, but we can, once again, not assume the presence of a Docker engine on the user's machine.

The solution we picked was to create an EC2 instance with StarCluster already installed. We call this particular instance the cluster *coordinator* or *orchestrator*, since it controls the whole cluster activity by executing the StarCluster commands to operate it.

The coordinator is launched using Jclouds within the `start` method of the `AWSJobService` created on the local machine. Its configuration is based on a prebuilt AMI[2] with an

[2]Public AMI ID: ami-b7d8cedd

installation of StarCluster and maintained by the developers of GridScale. After the coordinator is in a running state and its public IP address has been established, the `AWSJobService` can start sending commands to it via an SSH channel. This represents step 1 in Figure 18.

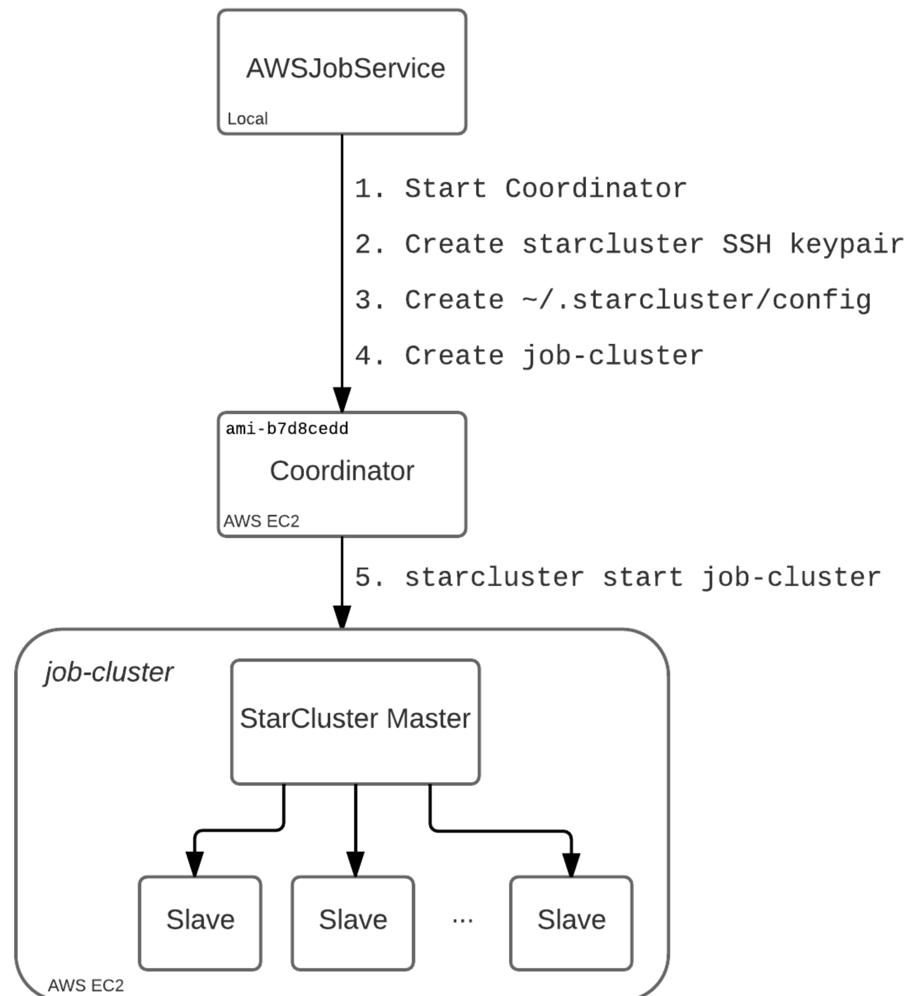


Figure 18: Creating a new cluster using a proxy coordinator node and StarCluster.

In order to start the coordinator node, Jclouds needs to authenticate the user. Amazon encourages using IAM[3] roles for authentication. This means that the user does not need to provide the root account password, but can instead create a user associated with GridScale, which is granted access to manage EC2 instances. Jclouds then uses the access key ID and secret key corresponding to the user role to manage machines via the API. The file containing the keys can be downloaded by the user from the Security Credentials page in the AWS Console [90]. The path to the CSV credentials file must be passed as the

[3] Identity Access Management

`awsCredentialsPath` parameter when creating a new `AWSJobService`.

Apart from using the IAM credentials to make EC2 API calls, we are also opening SSH channels to deliver commands to the coordinator. To do this we need to register the SSH key we are using to initiate the connection to the coordinator on AWS. This is achieved by importing the public key on AWS and giving it a name. The `awsKeypairName` associated with the private key is then always mentioned when openning an SSH connection.

Although it might seem that introducing a proxy node for passing StarCluster commands is inefficient, the actual overhead is minimal. The reason for the limited impact is that StarCluster is only explicitly invoked during the initialisation and destruction of the cluster. All job scheduling interactions with the job submission controller bypass the coordinator, since the job service can obtain the address of the master node and send it jobs directly. The coordinator does indeed incur a cost for being provisioned for as long as the cluster runs, but this is insignificant since we only need one of the cheapest instance types to run it.

StarCluster Configuration

StarCluster relies on a configuration file instead of command line parameters to specify the characteristics of clusters it is creating. The file must be located on the machine where StarCluster is being run, so we need to ensure that it is present on the coordinator node. Since the configuration parameters depend on the preferences of the user, the file cannot be statically embedded in the AMI that the coordinator is constructed from and needs to be generated dynamically. Listing 13 shows an example StarCluster configuration file constructed on-the-fly.

The authentication method used for bringing up EC2 instances is the same as the one used by Jclouds and StarCluster also needs an SSH keypair in order to set up SSH access to the master node and password-less SSH within the nodes in the cluster. Therefore, as step 2 in Figure 18, we first create a new private key, saved in this case as `starcluster-d4c9b13a`, and import it into AWS.

Step 3 is creating the `config` file on the coordinator node from the parameters passed to the `AWSJobService`. After the job service triggers the cluster initialization at step 4, the coordinator performs the launch using the `starcluster start` command at step 5.

The actions above leave the system in a state where the AWS cluster is initialized and

```
[global]
DEFAULT_TEMPLATE = jobcluster
ENABLE_EXPERIMENTAL = True

[aws info]
AWS_REGION_NAME = eu-west-1
AWS_REGION_HOST = ec2.eu-west-1.amazonaws.com
AWS_ACCESS_KEY_ID = <access-key-id>
AWS_SECRET_ACCESS_KEY = <secret-access-key>
AWS_USER_ID = <user-id>

[key starcluster-d4c9b13a]
KEY_LOCATION = .starcluster/starcluster-d4c9b13a

[cluster jobcluster]
KEYNAME = starcluster-d4c9b13a
CLUSTER_SIZE = 1
CLUSTER_USER = sgeadmin
CLUSTER_SHELL = bash
NODE_IMAGE_ID = ami-044abf73
MASTER_INSTANCE_TYPE = m3.medium
NODE_INSTANCE_TYPE = m1.small
```

Listing 13: StarCluster configuration file.

configured with the SGE scheduler, so we can start submitting jobs to the master node, which acts as a submission controller. For the purpose of job management, the job service should now ignore the coordinator and communicate directly with the master of the cluster. Figure 19 illustrates how this is achieved.

After obtaining the address of the master, the local machine needs to establish an SSH connection with it. Instead of creating a new set of SSH keys, we chose to reuse the keypair used by the coordinator to communicate with the cluster and transfer the private key locally under a temporary `~/.gridscale` directory. This keypair is already set up for connections to all nodes in the cluster and helps avoid the complexity of managing even more keys.

SGE Delegation

StarCluster deploys SGE as its scheduler and, as discussed in Section 4.1.1, we are able to delegate all job submission methods of the `AWSJobService` to an `SGEJobService`. This is possible thanks to full compatibility between the interfaces of different GridScale modules.

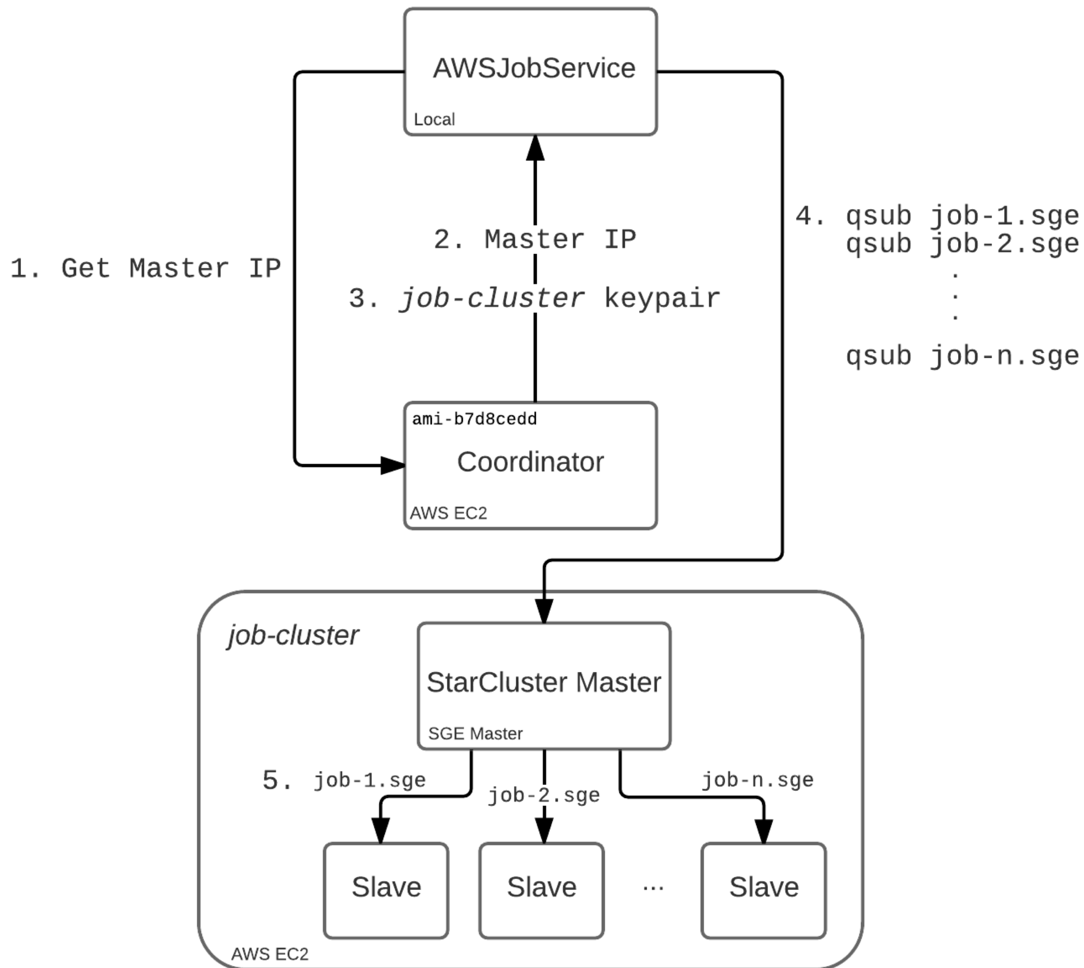


Figure 19: Connecting the local machine with the master node via SSH and distributing jobs.

A side effect is that `AWSJobDescription` is just a subtype of `SGEJobDescription`, as we are in fact submitting SGE jobs. The `AWSJobService` could have used `SGEJobDescriptions`, but we chose to hide this module dependency from the user of the library.

Tasks are specified in the same way as for plain SGE and are eventually translated to SGE jobs. Steps 4 and 5 in Figure 19 show the normal job submission mechanism after the setup phase is finished. Once a queue of jobs is populated, the work is distributed by the scheduling system to all nodes in the cluster.

Cluster Lifecycle

The coordinator node and the cluster are tightly coupled in our design because the cluster cannot be altered without the coordinator, which, in turn, is not relevant on its own. Therefore it makes sense for the lifecycle of the two components to be managed together, so the `close` method of `AWSJobService` shuts down the cluster, destroys the orchestrator and kills Jclouds communication channels to AWS.

Even though `start` and `close` could have been implicitly called respectively when an instance of the job service is created and when all jobs are finished, keeping them in the public API allows the user of a library more flexibility towards reusing the service for more sessions without having to repeatedly wait for the initialization steps.

4.1.3 Load Balancing

We use the built-in StarCluster load balancer to decrease or increase the number of nodes in the cluster between limits specified by the user. We start the daemon that performs job monitoring right after launching the cluster and keep the standard 60 seconds polling interval.

If the user does not specify a `minimumClusterSize` or `maximumClusterSize` for elastic scaling purposes when creating the job service, then we allow the cluster to shrink or extend in size between 1 node and its initial size. We argue that this is the cautious choice, since it can potentially only reduce costs and not incur any unexpected ones.

Another option we expose is to automatically killing the cluster (by setting the `autoClose` flag) if it is stays completely idle for a certain period after all jobs have been terminated. Some consumers of the library may prefer this as the default option rather than having to explicitly estimate when work is done and call `close`.

To obtain a highly elastic cluster, users can also reduce the 3 minute lookback window via the `lookbackWindow` parameter, which represents the time after which an idle node is considered for elimination. However, note that nodes will never be eliminated unless they have already been running for at least 45 minutes of the hour they have been rented for. Due to Amazon's hourly rental policy, an overly eager resource cutting approach would be non-optimal, since we would be giving away power that has already been paid for and might be employed in case of a sudden spike in job submissions.

4.1.4 Spot Instances

We support running a spot cluster by leveraging StarCluster to bid for the desired type of worker instances. An important consideration is that only slave nodes in the cluster are allowed to be spots, since we can easily deal with their downscaling by requeuing failed jobs. The master is always a dedicated on-demand instance, because having it suddenly killed would require restarting the whole cluster.

GridScale automates the spot bidding decision by using a simple heuristic based on the current bids for the instance and the average for the past month. We assume that the correct price will always tend towards the average, so we choose a bid depending on the relation between the current price and the average.

If the current market price is lower than the average, then we are ahead of the bidders that will soon lose their machines as prices will rise. In the opposite scenario, we know that the equilibrium price is lower than the current one, so we can afford to bid less than the market price while still remaining confident to be competitive.

If the current market price is lower than the average, then we are safe to use the instance for at least a couple of hours by simply bidding the average value, since spot prices fluctuate at a relatively steady rate. In the opposite scenario, we bid the market price plus an additional 10% to account for unexpected change, but we assume that the price will tend towards the average.

Overall, the strategy is rather conservative. We believe that no aggressive bids are needed to reduce renting costs, because prices for spot instances are already approximately 6 times lower than for on-demand ones as of June 2016 in US East, as shown in Table 2. Section 4.2 also details how OpenMOLE can gracefully deal with spot instances being lost, so complicated strategy to optimise bidding only has diminishing returns.

	Price ($ per hour)	
Instance type	**Spot**	**On-Demand**
c3.large	0.0177	0.105
m3.medium	0.0108	0.067
m3.xlarge	0.037	0.266
m3.2xlarge	0.0842	0.532

Table 2: Differences in prices between spot and on-demand instances [68].

4.2 OpenMOLE AWS Environment

The `AWSEnvironment` in OpenMOLE adapts the GridScale `AWSJobService` to the requirements of OpenMOLE and decorates it with extra batch submission and failure recovery capabilities.

Since user credentials and preferences are passed directly to the GridScale module, the environment's constructor parameters are, in the simplest case when default instance types are used, identical to those of the job service. Listing 14 shows how we modify the simple workflow from Listing 3 to run on an AWS cluster instead of the local machine.

```
1  val i = Val[Int]
2  val res = Val[Int]
3
4  val exploration = ExplorationTask(i in (1 to 100 by 1))
5
6  val model =
7    ScalaTask("val res = i * i") set (
8      inputs += i,
9      outputs += (i, res)
10   )
11
12 val env = AWSEnvironment(
13   region = "eu-west-1",
14   awsUserName = "adrian",
15   awsUserId = "434676269080",
16   awsKeypairName = "openmole",
17   awsCredentialsPath = "/Users/adrian/.aws/credentials.csv",
18   privateKeyPath = "/Users/adrian/.ssh/id_rsa",
19   clusterSize = 8
20 )
21
22 exploration -< (model on env hook ToStringHook())
```

Listing 14: Workflow running on an AWS cluster.

An important feature of the cluster built using GridScale is that it provides a network shared `/home` directory that is mounted on every node. This facilitates the runtime distribution in OpenMOLE, since it only needs to be copied remotely once, after which every machine can use it to run tasks in their packaged context. Section 6.1 discusses a future improvement that will allow us to avoid transferring the runtime remotely for every cluster

instantiation.

4.2.1 Job Service Lifecycle

The GridScale `AWSJobService` was designed specifically to allow for its lifecycle to be managed externally. In the OpenMOLE integration, we take advantage of this feature and turn off the entire environment based on events occurring in the message queue.

Specifically, we do not terminate the job service immediately after all the initial jobs have been consumed, but instead track all types of events occurring in the `JobManager` allocated to the environment and only kill the cluster once all potentially failed jobs have been successfully rerun, results have been copied and assembled back on the user's machine.

4.2.2 Spot Clusters

OpenMOLE's mechanism for job submission and dealing with unexpected failures of execution nodes plays well with the concept of spot clusters and automatic scaling of the number of worker nodes.

While adding or removing nodes based on the load in the cluster does not impact the runs of any jobs because only machines that have already been idle long enough are removed, tasks running on spot instances can be unexpectedly killed when the market price of the instance rises over the current bid. This case is dealt with by continuously polling for job results and resubmitting the job if the target machine is unreachable or the job has been cancelled for external reasons. The new submission is automatically picked up by the message queue and directed to one of the healthy nodes.

4.2.3 Resource Mapping

Although the default approach followed by OpenMOLE for running workflows is to try providing sane defaults for users without deep technical expertise, one of parameters that can be specified when constructing an environment is `OpenMOLEMemory`, which indicates the amount of memory that the workflow is expected to need.

For the `AWSEnvironment`, resources are allocated in terms of the number and types of instances chosen to form the underlying cluster, so we allow users unwilling to rely on the default `m3.medium` master and `m1.small` slave nodes to specify their own preferences for the cluster machines.

Additionally, to allow seamless transition from workflows that were previously being specified using the memory requirements to cloud-based executions, we introduce a *resource mapper* that translates the memory specification to a set of instances covering the desired configuration.

4.3 Other Cloud Platforms

Although the main focus of the implementation was to provide a fully functional system for running OpenMOLE experiments on AWS, the technique we used is not tied to a single cloud provider. Command line tools like CfnCluster, DC/OS and Elasticluster are similar with StarCluster in terms of interfaces and capabilities, so the cluster configuration layer can easily be replaced.

Additionally, most of the behaviour and structure of the AWS GridScale module and OpenMOLE environment can be extracted to interfaces for cloud services, especially since cloud providers themselves offer similar functionality. For instance, the `AWSJobService` can be split in multiple platform agnostic components that can be combined in a generic `CloudJobService`:

- The coordinator node is already created using Jclouds, so it can be deployed to many commercial clouds.
- All the mechanisms for transferring the OpenMOLE runtime remotely, packaging jobs and configuring SSH connections to nodes in the cluster only require access to a shell, so they do not depend on where resources are provisioned from.
- The job submission controller can delegate responsibility for translating and monitoring jobs to existing implementations such as the `SGEJobService`, the `SLURMJobService` or others based on the requirements of the underlying cluster.
- OpenMOLE cloud environments are virtually identical, since resource access is encapsulated in the GridScale modules with matching interfaces. Once multiple providers are supported, we only need a single cloud environment that instantiates different job services.

4.3.1 GCE Support

Taking Google Compute Engine as an example, we outline the main aspects that need to be considered when adding support for a new cloud platform. We do not yet provide a full working implementation for GCE, but we analyse the main differences and components that need to be changed in comparison with AWS.

The `GCEJobService` acts as the interface of the GridScale GCE module and it needs to provide the same public methods as the `AWSJobService`. The implementation details of starting and stopping the service depend on the tool chosen to manage the underlying cluster, while the `submit`, `cancel`, `state` and `purge` calls can be forwarded to the scheduler controlling the cluster.

Both Elasticluster and Mesosphere DC/OS provide primitives for creating and managing a GCE cluster, but we dismiss DC/OS due to the more elevated costs incurred by the usage of more powerful instances. On the other hand, Elasticluster's operating paradigm makes it very similar to StarCluster, with a few minor differences pointed out in Section 2.3.2.

Since we want to preserve the assumption of no configuration being necessary on behalf of the user and Elasticluster is a command line tool, it needs to be run by a remote coordinator node. The proxy node is started using the Jclouds API and SSH pairing between the local machine and the coordinator is performed by importing an SSH keypair associated with a service accounts. These serve a similar purpose as Amazon IAM roles and essentially regulate programmatic access to resources associated with the account.

The Elasticluster configuration file is based on the interplay between different Ansible playbooks and also needs to be generated dynamically. The example file shown in Listing 15 exposes similarities with StarCluster. It begins by specifying the details of the desired cloud provider, login details and Ansible playbook that provides a Slurm installation and an NFS-shared storage volume. Line 19 bring the components together and assembles a cluster with a single master node acting as an SSH entry point and two worker nodes.

After the cluster is created, the job submission workload can be forwarded to a `SLURMJobService`. A caveat is that Elasticluster does not provide a built-in way of expanding or shrinking the cluster depending on the load, but this can be implemented using the existing calls for adding and removing individual nodes.

```
[cloud / google]
provider = google
gce_client_id = <client-id>
gce_client_secret = <client-secret>
gce_project_id = <project-id>
zone = europe-west1-b

[login / google]
image_user = <google-username>
user_key_name = elasticluster
user_key_private = ~/.ssh/id_rsa
user_key_public = ~/.ssh/id_rsa.pub

[setup / ansible-slurm]
provider = ansible
frontend_groups = slurm_master
compute_groups = slurm_clients

[cluster / gridscale-cluster]
cloud = google
login = google
setup_provider = ansible-slurm
image_id = ubuntu-1404-trusty-v20160509a
flavor = f1-micro
frontend_nodes = 1
compute_nodes = 2
ssh_to = frontend
```

Listing 15: Elasticluster configuration file.

One advantage of running clusters on GCE is that instance initialisation times tend to be lower compared to AWS, resulting in a faster startup of the whole system. GCE does not have spot instances that vary in renting costs depending on demand. Instead, it introduces the concept of preemptible instances [91], which are even 70% cheaper than normal ones but can be interrupted under specific conditions.

A `GCEEnvironment` in OpenMOLE would wrap the job service and is not even necessary in the case of factoring out a generic `CloudEnvinronment` that receives the required job service as a parameter. However, the resource mapping feature needs to be overriden in order to account for Google-specific instance types and prices.

Chapter 5

Evaluation

In this chapter, we describe our approach to evaluating the new cloud-based execution environment in terms of performance, operating costs, as well as benefits brought to the end user. We describe the results of various benchmarks run against clusters with different configurations and discuss the challenges involved in correctly identifying the most influencing factors for the performance of distributed systems.

Although the specific end goal of the project was providing a new environment for running experiments on AWS from OpenMOLE, we also benchmark the underlying GridScale AWS module separately, since it can be used as a standalone component. We use DoC's[1] HTCondor and Slurm deployments to compare the speed of the new system with a locally hosted cluster.

Throughout the chapter, we delve into considerations about the costs of running specific workflows on AWS and advocate the use of different types of clusters depending on the nature of the workload. To illustrate the benefits of the fully automated job delegation to the cloud, we contrast it with the manual steps previously needed to obtain similar results with other experiment frameworks.

5.1 Fully Automated Experiments

OpenMOLE is now, to our knowledge, the only scientific experimentation framework that allows users to run experiments on commercial cloud environments without any form of configuration beyond providing their credentials. Listing 16 shows how the switch to the cloud is made by injecting the new environment in the workflow instantiation.

[1] Imperial College's Department of Computing

```
val (t1, t2, t3) = (EmptyTask(), EmptyTask(), EmptyTask())

val localhost = LocalEnvironment(threads = 8)
val aws = AWSEnvironment(
  region = "eu-west-1",
  awsUserName = "adrian",
  awsUserId = "434676269080",
  awsKeypairName = "gridscale",
  awsCredentialsPath = "/Users/adrian/.aws/credentials.csv",
  privateKeyPath = "/Users/adrian/.ssh/id_rsa",
  clusterSize = 8
)

val localMole = t1 -- (t2 on localhost) -- t3
val cloudMole = t1 -- (t2 on aws) -- t3
```

Listing 16: Creating an experiment ready to run both locally an in the cloud.

As described in the background chapter, other workflow platforms such as Taverna, Galaxy, or the Humman Connectome project have recently started cloud initiatives, but they still rely on a tedious setup on behalf of the user, who is required to follow long and complicated instruction steps [92–94].

Although quantifying the ease of use is a subtle task, we have so far been encouraged by feedback. We believe that features such as automatically mapping resources to cloud instances and generating bidding strategies for spot instances bring a quality of life improvement and encourage more users to take advantage of commercial clouds for research.

5.2 GridScale Benchmarks

As the foundation layer OpenMOLE's access to remote resources, GridScale bears significant importance for the overall performance of the application. We believe that benchmarks at this level offer valuable insights into the impact of choosing an appropriate instance type for the master node of the cluster, since the job submission system is evaluated in isolation from other configuration and setup routines employed by higher level systems.

5.2.1 Methodology

In this section, we particularly focus on the rate of at which jobs can be submitted, queried or cancelled on a cluster deployed on Amazon EC2. At this stage, we are not interested in the overall execution time of the submitted jobs, so we do not delegate real work and jobs are just busy-waiting to be cancelled.

Our setup consists of 5 different types of clusters with different types of instances as the single master node. The master is the only job submission controller, so no worker nodes are needed. For each run, we instantiate an `AWSJobService`, submit, query and eventually cancel a number of jobs between 100 and 1000 increasing in steps of 100.

To ensure a better estimation of the real duration, we repeat each run 10 times and average the results. Table 3 shows the specifications and prices of all types of instances used in this chapter.

					Price ($ per hour)	
Instance Type	**Memory (GB)**	**ECUs**[2]	**vCPUs**[3]	**Network Performance**	**Spot**	**On-Demand**
m1.small	1.7	1	1	Low (125 Mbps)	0.01	0.047
m1.medium	3.75	2	1	Moderate (250 Mbps)	0.01	0.095
m3.medium	3.75	3	1	Moderate (300 Mbps)	0.01	0.073
c1.medium	1.7	5	2	Moderate (250 Mbps)	0.02	0.148
m1.xlarge	15	8	4	High (1000 Mbps)	0.04	0.379
c3.xlarge	7.5	14	4	Moderate (500 Mbps)	0.043	0.239
c3.4xlarge	30	62	16	High (2 Gbps)	0.168	0.953
c3.8xlarge	60	132	36	Very High (10 Gbps)	0.327	1.906

Table 3: Performances and prices of AWS EC2 instances. Prices as of June 2016 in EU West [21].

At the moment, not all GridScale modules support batching commands sent to the submission controller via multiple sessions within the same SSH connection, so we consider two main scenarios based on whether operations to the master are transmitted using only one or more sessions.

5.2.2 Results

In the case of a single session per connection, we also compare the 5 master nodes of the GridScale SGE cluster with the manager of the HTCondor in DoC. This is an Intel Xeon

[2]An EC2 Compute Unit is the approximate equivalent of a 1.0-1.2 GHz 2007 Intel Xeon
[3]Virtual CPUs

E5-2470 with 32GB RAM and 16 virtual cores running at 2.3GHz.

Figure 20 shows that submissions to the HTCondor server are significantly faster for all instructions, partly due to the high bandwidth on the local network, but mostly thanks to the very low latency, which reduces the time needed to establish an SSH connection.

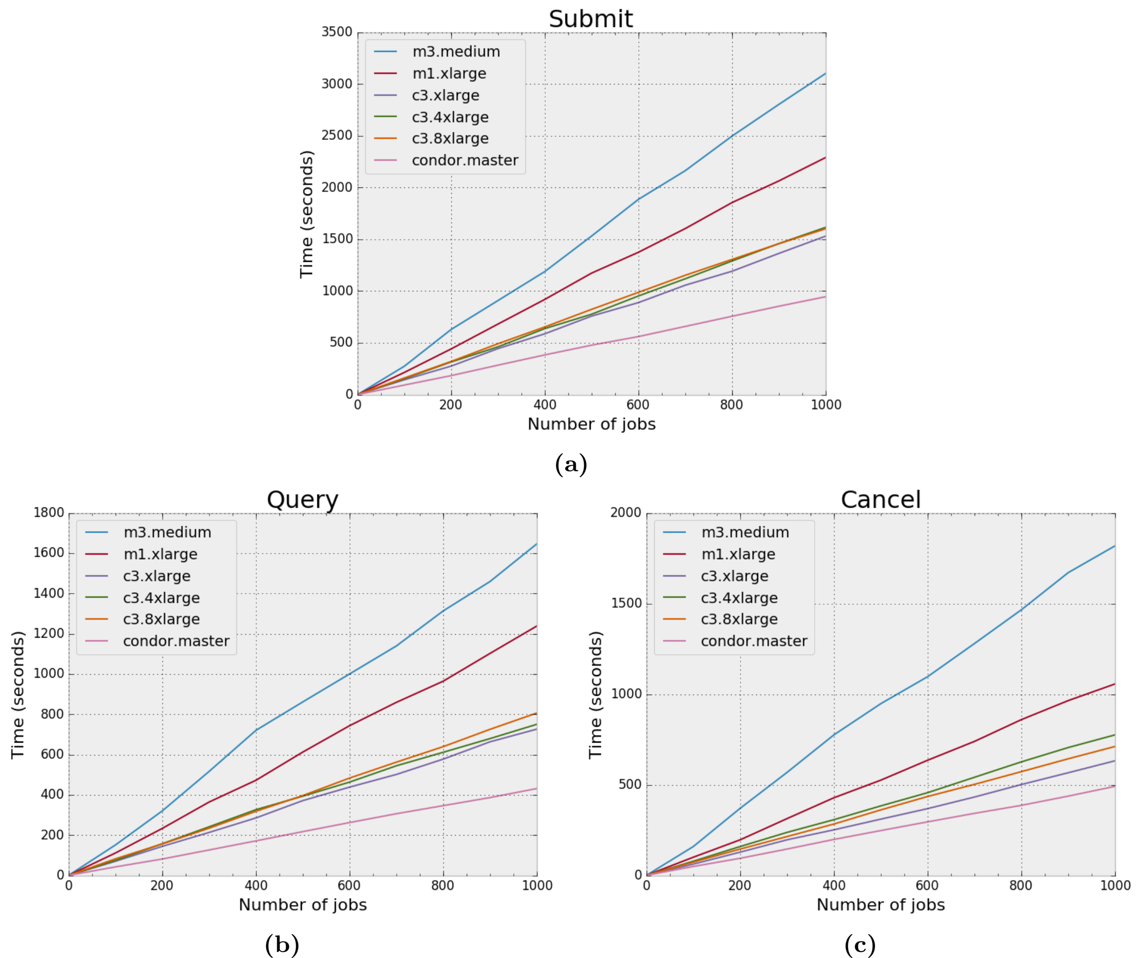


Figure 20: Comparison for submission, query and cancellation times in the case of only one session per SSH connection.

While the `m3.medium` clearly lags behind given its low bandwidth, the three high-CPU `c3` instances exhibit close performance and are faster than the high-throughput `m1.xlarge` instance, indicating that bandwidth only has diminishing returns and the CPU is more important, while latency remains a constant factor.

When connections to master nodes are allowed to open multiple sessions, Figure 21 shows the rate of commands per second improves dramatically across the board. For each of the three commands, multi-session SSH connections bring 15x improvements to the execution

time. The `c3` instances continue to exhibit similar performance, which is mildly surprising due to the vastly superior specifications of the `c3.8xlarge` instance.

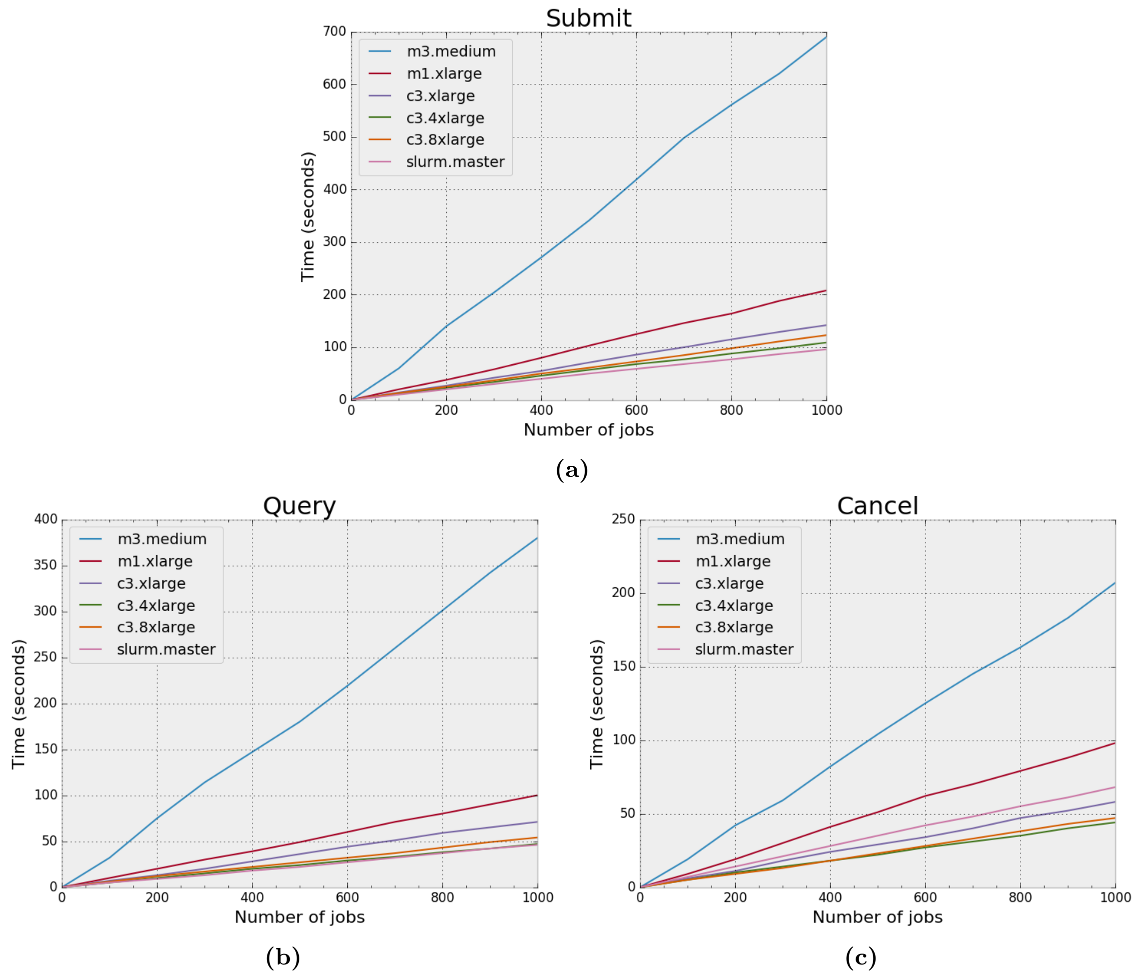


Figure 21: Comparison for submission, query and cancellation times in the case of multiple sessions per SSH connection.

In this case, we are comparing the command latencies with the Slurm master node, which has the same characteristics as the HTCondor one. Despite being on the local network when the benchmark was run, it responds to *cancel* requests slower than the `c3` instances.

5.3 OpenMOLE Workflow Benchmarks

On the OpenMOLE level, we are interested in how the application behaves as a whole when distributing real algorithms over a cloud cluster. In this section, we look at the performance of the system when engaged in running two full different workflows with different underlying cluster configurations. We then estimate the cost of running each of the workflows.

5.3.1 π Computation

The algorithm presented in Appendix A, Listing 17 computes a Monte-Carlo estimation of π. It first computes the result of 100 parallel executions of the `ScalaTask` starting each time from a generated random seed, after which it collects the results and averages them to give a final estimation. This is still a simple algorithm, but more substantial than the ones we have presented so far.

Table 4 summarises the results of running the algorithm 10 times with different types of master and worker nodes, while also using different number of estimations to obtain the final result. Note that each estimation corresponds to a random seed in the workflow. The total cost of each run is computed using the prices in Table 3 and we also show prices for on-demand cluster, even if all the experiments were carried out using full spot clusters.

						Total Price ($)	
	Master Instance	**Worker Instance**	**Worker Nodes**	**Estimations**	**Time (s)**	**Spot**	**On-Demand**
1	m1.medium	m1.small	1	200	5054	0.04	0.284
2	c1.medium	m1.small	1	100	3565	0.02	0.142
3	c3.8xlarge	c3.xlarge	4	100	929	0.499	2.862
4	c3.8xlarge	c3.xlarge	8	100	1032	0.671	3.818
5	c3.8xlarge	c3.xlarge	8	100	915	0.671	3.818
6	c3.8xlarge	c3.xlarge	8	100	2820	0.671	3.818
7	c3.8xlarge	c3.xlarge	16	100	792	1.015	5.73
8	c3.4xlarge	c3.xlarge	4	100	1142	0.34	1.909
9	c3.4xlarge	c3.xlarge	8	100	964	0.512	2.865
10	c3.4xlarge	c3.xlarge	16	1000	4827	1.712	9.554

Table 4: Summary of various runs of the π computation algorithm, including the price of each execution.

We can already observe large differences in execution times between runs with similar cluster configurations. For example, run 6 is over 300% slower than run 5, given identical instance types and the same number of jobs. By inspecting the load of the system during

the workflow run, we observed that the available CPU and memory often fluctuated aggressively without the instances being flooded with requests. We partially attribute this to the fact that in the end we are working with virtual instances that share the same physical machine.

Very long runs like 7 occur occasionally when instances in the cluster are not reliable. An occasional pattern occurring in particular with spot clusters was instances rejecting to receive jobs due to insufficient disk space, although we are not storing anything specifically on the machines themselves and the working directory is always set to `/home`, the volume shared on the network by StarCluster.

Since each run spends the first approximately 330 - 380 seconds setting up the cluster, we consider total running times of about 1000 seconds acceptable. This is because the simple task executions used here actually require the same networking resources as genuinely CPU-intensive ones to reach their destinations execution nodes, so more complicated workflows are not necessarily more time-consuming.

5.3.2 Random Forest

The random forest workflow in Appendix B, Figure 18 explores the parameters of the `forest.py` script, which performs random forest image classification [95] to distinguish 3 types of leaves. The Python script is packaged as a CARE task and receives as parameters the location of the dataset, the number of trees in a forest and the depth of each tree. The output of the script is a single floating point number representing the precision of the k-fold trained classifier.

As before, we ran the workflow multiple times with different configurations, but eventually only chose the results in Table 5 for display.

						Total Price ($)	
	Master Instance	**Worker Instance**	**Worker Nodes**	**Generated Jobs**	**Time (s)**	**Spot**	**On-Demand**
1	c3.8xlarge	c3.xlarge	4	337	3219	0.499	2.862
2	c3.8xlarge	c3.xlarge	8	337	2625	0.671	3.818
3	c3.4xlarge	c3.xlarge	8	337	4117	1.024	5.73

Table 5: Summary of various runs of the random forest classifier training, including the price of each execution.

Here we can notice that, despite only having 337 generated jobs to execute, the tasks were indeed more time consuming, which is in accordance with the training of the classifier in

the Python script. It would probably be expected that the second cluster configuration is the fastest one, but it is interesting to compare run 1 and 3. This leads us to the conclusion that a powerful instance as the master is indicated to having a larger pool of workers, since the master can easily become a bottleneck for very demanding workflows.

5.4 Spot Clusters

Throughout this section, we have learned that clusters consisting entirely of spot instances are not always reliable. Although we can deal with instances being revoked by EC2 via OpenMOLE's resubmission mechanisms, this has only been necessary once during the whole development and testing process.

Spot prices do not usually change abruptly, so instances being suddenly revoked is not a very common problem. Additionally OpenMOLE can already recover from sudden job failures and non-critical instances failing.

However, we do not recommend ever letting the master node of a cluster be a spot instance, since it being revoked would result into an irrecuperable state for OpenMOLE. Apart from the possibility of it being revoked, the biggest concern is in fact the potential flakiness of spot instances. Theoretically, spots are advertised as being the same as on-demand instances, but in practice we have found them to be more unreliable.

Despite all the potential issues with spot instances, we do recognize that they offer great value and workflows run on spot clusters are, as seen in Table 4, on average between 5 and 8 times cheaper than on-demand clusters. This is particularly true if generating many short-lived jobs whose recovery in case of failure is cheap.

Another issue discovered during the evaluation was that Amazon limits by default the number of spot instances a user can bid for or run simultaneously to 20. This is, however, not a hard requirement and can be changed by negotiating directly with the provider.

5.5 Challenges and Limitations

The evaluation of our implementation on both the OpenMOLE and the GridScale level was influenced by several factors. We did manage to overcome some of the problems,

while others issues require deeper changes in the structure of the project in order to be solved.

One of the main drawbacks of our evaluation process was that it required extensive manual intervention and supervision. Although we were able to write a framework that automatically collects data about job submission statistics as part of GridScale, the holistic evaluation at the OpenMOLE layer was mostly performed using either the console mode of the command line tool or the web interface. This did not provide us with too much control over testing scenarios and we were limited to running the application as a simple user. More granular access to the internals of the application would have enabled us to more easily identify bottlenecks.

A challenge adjacent to the first point we made was the lack of integration testing present in both GridScale and OpenMOLE. Although setting up an integration testing framework is an inherently hard problem in the context of applications dealing with distributed systems, the presence of a reliable test suite would certainly increase the confidence of the developers in the functional correctness of the product and eventually speed up development cycles.

One of the reasons for which the experiment results may be seriously skewed is the volatility of Amazon's EC2 resource pool. Especially in the case of spot instances, which we have used almost exclusively for experiments, differences in performance can become a rather serious problem when trying to diagnose issues with the application.

Our comparison with job submission systems supported by the Department of Computing may also suffer from inaccuracies. In fact, this is a more general problem, since comparing systems relying on different resource pools will always cause slight differences. One concrete example is the comparison between workflows run on an EC2 cluster and on DoC's Slurm or HTCondor deployment. The average machine in the department is the rough equivalent of the `c3.4xlarge`, the second most powerful machine we rented for testing purposes. The network speed of the local connection also generally skews results against the cloud approach, although we always accounted for it when trying to explain performance differences.

Chapter 6

Conclusion

In this report, we describe the approach taken towards adding cloud platforms to the arsenal of execution environments supported at levels of both OpenMOLE and GridScale. We discuss in detail the specifics of adding support for Amazon EC2 and present a generic design, under which new cloud providers can be added easily. We then proceed to evaluate our implementation in a real-world setting by using the new environments in various benchmarks and present the cost implications and practical considerations of operating a cluster in the cloud.

We believe that we have accomplished the main goals set out for this project. We have conducted an extensive analysis on tools used to manage and leverage cloud resources, followed by the successful integration of the GridScale AWS module with OpenMOLE and the evaluation of our results. Overall, we can draw multiple insights about our current setup and cloud computing in general from the experiments we have conducted. One of them is that cloud instances can and will experience slowdowns or failures. Unless renting more expensive dedicated instances, additional care needs to be taken for dealing with bottlenecks in a cluster.

Furthermore, choosing the right environment to distribute work to is contextual. As we have seen, network performance is an important factor for workflows dominated by light short-lived jobs, since in our benchmarks local servers almost always performed better, while remote machines with vastly different specifications but same network performance fared similarly.

Before embarking on the project, we had not considered some of these factors and were initially surprised that cloud environments do not translate automatically to improved performance in relation to our local clusters. However, we now have several ideas for improving the speed and resilience of our implementation, as discussed in Section 6.1

6.1 Future Work

Some of the ideas we have for future improvements of the project include:

- Adding support for multiple cloud providers, including continuing development for the GCE module that is already under development. We believe that this is the aspect that will attract the most users, since most other workflow management systems only support Amazon as the single cloud provider, if any.
- Implementing support for SGE Arrays as a way of batching jobs sent for submission to SGE clusters as part of StarCluster. We already know that workloads with a large number of short-lived jobs are common in when performing parameter explorations, so this could significantly decrease average submission times to SGE.
- One of the main slowdown reasons at the start of OpenMOLE's workflow execution on clusters in the cloud is the fact that it always needs to transfer its runtime remotely since a fresh machine is provisioned on each cluster instantiation. This could potentially be avoided by creating an EBS volume containing the required packages and mounting it on every instance in the cluster. This would avoid uploading approximately 400MB of data, saving therefore significant time.
- A more open ended problem is the issue of dealing with nodes in a cluster that become bottlenecks over the course of a run. Simple potential solutions include cancelling the jobs the node is currently running and giving it time to recover or even kill it in extreme cases.
- An important factor for the long-term development of the project would be creating a thorough integration testing framework. Both developers and users would vastly benefit from this move, as it would strongly enforce the resilience of the application.

References

[1] Carole Goble and David De Roure. The Impact of Workflow Tools on Data-centric Research. *The Fourth Paradigm: Data-Intensive Scientific Discovery*, pages 137–145, 2009. ISBN 978-0-9825442-0-4. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/default.aspx.

[2] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all? *2008 Cairo International Biomedical Engineering Conference*, pages 1–9, 2008. ISBN 978-1-4244-2694-2. ISSN 1424426944. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4786077.

[3] Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew S. Shields. Workflows for e-Science: Scientific Workflows for Grids. *Workflows for e-Science: Scientific Workflows for Grids*, pages 1–523, 2007. ISBN 978-1-84628-519-6. ISSN 1849966192. URL http://link.springer.com/10.1007/978-1-84628-757-2.

[4] Taverna - Open source and domain independent Workflow Management System. URL http://www.taverna.org.uk/.

[5] The Kepler Project. URL https://kepler-project.org/.

[6] Pegasus. URL http://pegasus.isi.edu/.

[7] The Galaxy Project: Online bioinformatics analysis for everyone. URL https://galaxyproject.org/.

[8] Romain Reuillon, Mathieu Leclaire, and Sebastien Rey-Coyrehourcq. OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981–1990, 2013. ISSN 0167739X.

[9] Romain Reuillon, Florent Chuffart, Mathieu Leclaire, Thierry Faure, Nicolas Dumoulin, and David Hill. Declarative task delegation in OpenMOLE. *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages

55–62, 2010. ISBN 9781424468300.

[10] Yves Janin, Cédric Vincent, and Rémi Duraffort. CARE, the comprehensive archiver for reproducible execution. *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering - TRUST '14*, pages 1–7, 2014. ISBN 9781450329514. URL http://dl.acm.org/citation.cfm?doid=2618137.2618138.

[11] Romain Reuillon and Jonathan Passerat-Palmbach. Model Exploration Using OpenMOLE a workflow engine for large scale distributed design of experiments and parameter tuning. *IEEE High Performance Computing and Simulation conference 2015.*, 2015. ISBN 9781467378123.

[12] OpenMOLE Marketplace. URL https://github.com/openmole/openmole-market.

[13] The Scala Programming Language. URL http://scala-lang.org/.

[14] EGI - European Grid Infrastructure. URL http://www.egi.eu/.

[15] The PBS job scheduler. URL http://www.arc.ox.ac.uk/content/pbs.

[16] TORQUE Resource Manager. URL http://www.adaptivecomputing.com/products/open-source/torque/.

[17] Open Grid Engine. URL http://sourceforge.net/projects/gridscheduler/.

[18] Simple Linux Utility for Resource Management. URL http://slurm.schedmd.com/.

[19] HTCondor. URL https://research.cs.wisc.edu/htcondor/.

[20] OAR. URL https://oar.imag.fr/.

[21] Amazon Elastic Compute Cloud (EC2) Cloud Server & Hosting. URL https://aws.amazon.com/ec2/.

[22] Google Cloud Platform. URL https://cloud.google.com/.

[23] OpenStack Open Source Cloud Computing Software. URL https://www.openstack.org/.

[24] Apache CloudStack: Open Source Cloud Computing. URL https://cloudstack.apache.org/.

[25] Jonathan Passerat-Palmbach, Daniel Rueckert, and Romain Reuillon. GridScale: distributed computing for applications running in the Java Virtual Machine. 2016.

[26] TOP500 Supercomputer Sites. URL http://www.top500.org/featured/top-systems/.

[27] European Middleware Initiative. URL http://www.eu-emi.eu/.

[28] Judith M. Myerson. Cloud computing versus grid computing, mar. URL http://www.ibm.com/developerworks/library/wa-cloudgrid/.

[29] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific workflow applications on Amazon EC2. *2009 5th IEEE International Conference on E-Science Workshops*, pages 59–66, 2009. ISBN 978-1-4244-5946-9. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5408002.

[30] Clouds and grids compared - Cloud lounge. URL http://www.cloud-lounge.org/clouds-and-grids-compared.html.

[31] Amazon Web Services. URL https://aws.amazon.com/.

[32] Amazon Simple Storage Service (S3) - Object Storage. URL https://aws.amazon.com/s3/.

[33] Amazon Elastic Block Store (EBS) AWS Block Storage. URL https://aws.amazon.com/ebs/.

[34] Romain Reuillon. OpenMOLE: a DSL to explore complex-system models, 2012. URL http://www.openmole.org/files/openmole-com/slides/2012/dsl/.

[35] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013. ISSN 0305-1048. URL http://nar.oxfordjournals.org/lookup/doi/10.

1093/nar/gkt328.

[36] caGrid. URL http://www.cagrid.org/display/cagridhome/Home.

[37] BioVeL. URL https://www.biovel.eu/.

[38] Taverna caGrid. URL https://github.com/NCIP/taverna-grid.

[39] Giacinto Donvito. BioVeL: Taverna Workflows on distributed grid computing for Biodiversity. URL https://indico.egi.eu/indico/event/1222/session/34/contribution/53/material/slides/0.pdf.

[40] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010. ISBN 1465-6914 (Electronic)\r1465-6906 (Linking). ISSN 1465-6906.

[41] Enis Afgan, Dannon Baker, Nate Coraor, Brad Chapman, Anton Nekrutenko, and James Taylor. Galaxy CloudMan: delivering cloud compute clusters. *BMC bioinformatics*, 11 Suppl 1(Suppl 12):S4, 2010. ISBN 1471-2105 (Electronic)\n1471-2105 (Linking). ISSN 1471-2105. URL http://www.biomedcentral.com/1471-2105/11/S12/S4.

[42] Mohamed Abouelhoda, Shadi a Issa, and Moustafa Ghanem. Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support. *BMC bioinformatics*, 13(1):77, 2012. ISBN 1471-2105 (Electronic)\n1471-2105 (Linking). ISSN 1471-2105. URL http://www.ncbi.nlm.nih.gov/pubmed/22559942.

[43] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. ISSN 1532-0626. URL http://doi.wiley.com/10.1002/cpe.994.

[44] Jianwu Wang and Ilkay Altintas. Early Cloud Experiences with the Kepler Scientific Workflow System. *Procedia Computer Science*, 9:1630–1634, 2012. ISSN 18770509. URL http://linkinghub.elsevier.com/retrieve/pii/S1877050912003006.

[45] STAR: Cluster. URL http://star.mit.edu/cluster/.

[46] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Pegasus: Mapping scientific workflows onto the grid. *Grid Computing*, 3165/2004:131–140, 2004. ISBN 978-3-540-22888-2. ISSN 1541-1672. URL `http://www.springerlink.com/index/95rj5e2fgqqpkaha.pdf`.

[47] Ewa Deelman. Pegasus, a Workflow Management System for Large-Scale Science. *Journal of Chemical Information and Modeling*, page 160, 2013. ISBN 9788578110796. ISSN 1098-6596.

[48] Ewa Deelman, Karan Vahi, Mats Rynge, Gideon Juve, Rajiv Mayani, and Rafael Ferreira da Silva. Pegasus in the Cloud: Science Automation through Workflow Technologies. *IEEE Internet Computing*, 20(1):70–76, 2016. ISSN 1089-7801. URL `http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=7373501`.

[49] DAGMan. URL `https://research.cs.wisc.edu/htcondor/dagman/dagman.html`.

[50] E. Deelman. Grids and Clouds: Making Workflow Applications Work in Heterogeneous Distributed Environments. *International Journal of High Performance Computing Applications*, 24(3):284–298, 2010. ISBN 1094342009356. ISSN 1094-3420.

[51] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel Katz, and Gaurang Mehta. Workflow task clustering for best effort systems with Pegasus. *Mardis Gras Conference*, (c):8, 2008. ISBN 9781595938350. URL `http://portal.acm.org/citation.cfm?doid=1341811.1341822`.

[52] Running Workflows on Amazon EC2 using Pegasus. URL `https://confluence.pegasus.isi.edu/display/pegasus/Cloud+Tutorial`.

[53] NFS: Network File System Protocol specification. URL `https://tools.ietf.org/html/rfc1094`.

[54] Open Source High Performance Computing. URL `https://www.open-mpi.org/`.

[55] IPython. URL `https://ipython.org/`.

[56] Elasticluster. URL `https://gc3-uzh-ch.github.io/elasticluster/`.

[57] Ganglia Monitoring System. URL `http://ganglia.info/`.

[58] Ansible. URL `http://www.ansible.com/`.

[59] Storage for your Cloud. Gluster. URL https://www.gluster.org/.

[60] Ceph. URL http://ceph.com/.

[61] Orange File System. URL http://orangefs.org/.

[62] Amazon SNS. URL https://aws.amazon.com/sns/.

[63] Amazon SQS. URL https://aws.amazon.com/sqs/.

[64] Amazon CloudWatch. URL https://aws.amazon.com/cloudwatch.

[65] Amazon Auto Scaling. URL https://aws.amazon.com/autoscaling/.

[66] OpenLava. URL http://www.openlava.org/.

[67] CfnCluster Networking. URL https://cfncluster.readthedocs.io/en/latest/networking.html.

[68] AWS Pricing. URL https://aws.amazon.com/pricing/services/.

[69] CfnCluster Processes. URL https://cfncluster.readthedocs.io/en/latest/processes.html.

[70] Apache Jclouds. URL https://jclouds.apache.org/.

[71] Mesos. URL https://mesos.apache.org/.

[72] Paco Nathan. Datacenter Computing with Apache Mesos, 2014. URL http://www.slideshare.net/pacoid/datacenter-computing-with-apache-mesos.

[73] Chronos. URL https://mesos.github.io/chronos/.

[74] Cron. URL https://en.wikipedia.org/wiki/Cron.

[75] Amazon Elastic MapReduce. URL https://aws.amazon.com/elasticmapreduce/.

[76] Mesosphere DC/OS. URL https://dcos.io/.

[77] DC/OS System Requirements. URL https://dcos.io/docs/1.7/administration/installing/custom/system-requirements/.

[78] OpenMOLE. URL http://www.openmole.org/.

[79] Jonathan Passerat-Palmbach, Romain Reuillon, Mathieu Leclaire, Antonios Makropoulos, Emma Robinson, Sarah Parisot, and Daniel Rueckert. Large-scale neuroimaging studies with the OpenMOLE pipeline engine. 2016.

[80] Clara Schmitt, Sébastien Rey-Coyrehourcq, Romain Reuillon, and Denise Pumain. Half a billion simulations: evolutionary algorithms and distributed computing for calibrating the SimpopLocal geographical model. *Environment and Planning B: Planning and Design*, 0(0):0, 2015. ISSN 0265-8135, 1472-3417. URL http://www.arxiv.org/pdf/1502.06752.pdf.

[81] Scala Actors. URL http://www.scala-lang.org/old/node/242.

[82] Open Mole Environments. URL http://www.openmole.org/current/Documentation_Language_Environments.html.

[83] Docker. URL https://www.docker.com/.

[84] OGF. URL https://www.ogf.org/.

[85] DRMAA. URL https://www.drmaa.org/.

[86] Open Grid Forum. A Simple API for Grid Applications (SAGA). 71:324, 2007. URL http://www.ogf.org/documents/GFD.90.pdf.

[87] OSGi. URL https://www.osgi.org/.

[88] Mark Harrison. Cake pattern in depth. 2011. URL http://www.cakesolutions.net/teamblogs/2011/12/19/cake-pattern-in-depth.

[89] SSHJ. URL https://github.com/hierynomus/sshj.

[90] AWS Security Credentials. URL https://docs.aws.amazon.com/general/latest/gr/getting-aws-sec-creds.html.

[91] GCE Preemptible VM Instances. URL https://cloud.google.com/compute/docs/instances/preemptible.

[92] Taverna AWS Setup. URL https://wiki.biovel.eu/display/doc/Taverna+

Server+AMI#TavernaServerAMI-Furtherinformation.

[93] Galaxy CloudMan AWS Setup. URL https://wiki.galaxyproject.org/CloudMan/AWS/GettingStarted.

[94] The Human Connectome AWS Pipeline. URL https://goo.gl/x5YLvf.

[95] Random Forest. URL https://en.wikipedia.org/wiki/Random_forest.

Appendix A

π Computation

```
val seed = Val[Long]
val pi = Val[Double]
val piAvg = Val[Double]

val exploration =
  ExplorationTask(seed in UniformDistribution[Long]() take 5)

val model =
  ScalaTask("""
    |val random = newRNG(seed)
    |val points = 100000
    |val inside =
    |  for {
    |    i <- (0 until points).toIterator
    |    x = random.nextDouble()
    |    y = random.nextDouble()
    |  } yield { (x * x) + (y * y) }
    |  val pi = (inside.count(_ < 1).toDouble / points) * 4
    |""".stripMargin) set (
      name := "pi",
      inputs += seed,
      outputs += pi
    )

val average =
  ScalaTask("""val piAvg = pi.sum / pi.size""") set (
    name := "average",
    inputs += pi.toArray,
    outputs += piAvg
  )

val env = AWSEnvironment(...)

exploration -< (model on env) >- (average hook ToStringHook())
```

Listing 17: Workflow computing an approximation of π using a Monte Carlo algorithm [12].

Appendix B

Random Forest

```
val images = Val[Array[File]]
val nbTrees   = Val[Int]
val treeDepth = Val[Int]
val kFold = 10

val testDir = File("/Users/adrian/randomforest")
val random = new util.Random(42)
val imagesArrays =
  (0 until kFold).map(i => random.shuffle((testDir / "images").listFiles.toSeq).toArray)

val parameterExploration =
  ExplorationTask(
    (nbTrees in (5 to 25 by 5)) x
    (treeDepth in (3 to 18 by 3))
  )

val imagesExploration =
  ExplorationTask(images in imagesArrays) set (
    inputs  += (nbTrees, treeDepth),
    outputs += (nbTrees, treeDepth)
  )

val learningOutput = Val[String]

val learning = CARETask(
  testDir / "archive_python2.bin",
  "python forest.py /Users/adrian/randomforest/images ${nbTrees} ${treeDepth}"
) set (
  inputs += (nbTrees, treeDepth),
  inputFileArrays += (images, "/Users/adrian/randomforest/images/image", ".jpg"),
  stdOut := learningOutput,
  outputs += (nbTrees, treeDepth)
)

val env = AWSEnvironment(...)

val pointsHook =
  AppendToCSVFileHook(
    testDir / "points.csv",
    nbTrees, treeDepth, learningOutput)

parameterExploration -< imagesExploration -< (learning on env hook pointsHook)
```

Listing 18: Workflow exploring the parameters of a random forest image classifier written in Python. [12].