Scalable Deployment of a LIGO Physics Application on Public Clouds: Workflow Engine and Resource Provisioning Techniques

Suraj Pandey, Letizia Sammut, Rodrigo N. Calheiros, Andrew Melatos, and Rajkumar Buyya

Abstract Cloud computing has empowered users to provision virtually unlimited computational resources and are accessible over the Internet on demand. This makes Cloud computing a compelling technology that tackles the issues rising with the growing size and complexity of scientific applications, which are characterized by high variance in usage, large volume of data and high compute load, flash crowds, unpredictable load, and varying compute and storage requirements. In order to provide users an automated and scalable platform for hosting scientific workflow applications, while hiding the complexity of the underlying Cloud infrastructure, we present the design and implementation of a PaaS middleware solution along with resource provisioning techniques. We apply our PaaS solution to the data analysis pipeline of a physics application, a gravitational wave search, utilizing public Clouds. The system architecture, a load-balancing approach, and the system's behavior over varying loads are detailed. The performance evaluation on scalability and load-balancing characteristics of the automated PaaS middleware demonstrates the feasibility and advantages of the approach over existing monolithic approaches.

1 Introduction

Cloud computing enables users to get virtually unlimited computational resources that can be accessed on demand from anywhere at any time. The main features of Clouds such as elasticity and pay-per-use cost model enable low upfront investment

S. Pandey
IBM Research Australia, Melbourne, Australia
e-mail: suraj.pandey@au.ibm.com

L. Sammut • A. Melatos
School of Physics, The University of Melbourne, Parkville, VIC 3010, Australia
e-mail: l.sammut@student.unimelb.edu.au; amelatos@unimelb.edu.au

R.N. Calheiros (✉) • R. Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia
e-mail: rnc@unimelb.edu.au; rbuyya@unimelb.edu.au

X. Li, J. Qiu (eds.), *Cloud Computing for Data-Intensive Applications*,
DOI 10.1007/978-1-4939-1905-5_1

and low time to market, which in turn enables small to large software applications to use the Cloud as a hosting platform, in contrast to traditional enterprise infrastructure settings. This makes Cloud computing a compelling technology to tackle the issues rising with the growing size and complexity of scientific applications. For instance, for a typical problem size, a single physics application may scale to a few thousand processors, and multi-physics applications not only are increasing in size, but are also requiring more sophisticated workflows for their execution [1].

Most large-scale applications, such as scientific applications, are characterized by high variance in usage, mixture of data and compute load, flash crowds, unpredictable load, and varying compute and storage requirements. This makes the management of the computational infrastructures supporting such applications a complex task, even when public Infrastructure as a Service (IaaS) Cloud resources—such as virtual machines—are used as the underlying system infrastructure.

The above situation can be mitigated with the utilization of *Platform as a Service* (PaaS). PaaS Clouds offer to users a complete platform for hosting user-developed applications, while hiding the underlying infrastructure. Therefore, complex operations such as automatic scaling, load balancing, and management of virtualized environments are completely transparent to users, and happen without their direct interference.

In this article, we describe the design and implementation of a system delivering a scalable solution to scientific workflow applications, specifically focusing on the data analysis pipeline underpinning a high-profile scientific (physics) application: gravitational wave searches. The proposed solution is a PaaS middleware that uses resources from public Cloud infrastructures (IaaS) for hosting the management and application services.

Gravitational waves (GW) are ripples in the fabric of space–time that result from galactic collisions, stellar explosions, or rapid acceleration of large and extremely dense objects such as neutron stars [22]. In principle, the ripples can be detected by measuring minute changes in the separation of test masses on Earth, for example the mirrors on a long-baseline, laser, Michelson interferometer. However, the changes in separation are so small—one part in 10^{21} for the strongest predicted sources—that they have not yet been detected. A worldwide effort is currently under way to achieve the first detection, led by a new generation of interferometric antennas like the Laser Interferometer Gravitational-Wave Observatory (LIGO) and partner facilities around the world like VIRGO, GEO600, and TAMA300 [2].

Numerous search algorithms have been applied to the GW data from the above detectors, all of them computationally intensive. There are four main types of GW signal: stochastic, burst, continuous, and compact binary coalescence. Each search for a specific type of source covers a wide parameter space, with an optimal balance required between parameter space mismatch and computational resources. The search space is especially large for blind, all-sky searches where the electromagnetic counterpart of the source is unknown. In this paper, we concentrate on a search for periodic gravitational waves from Sco X-1. Sco X-1 is the brightest X-ray source in the sky. It is thought to be an accreting neutron star [22]. Theoretical analysis indicates that it may also be a strong GW candidate [4, 16, 24].

GW searches can be represented as a workflow consisting of tasks linked through data dependencies. Execution of the workflow can be parallelized in such a way that each parallel instance operates in a different multi-dimensional parameter set. Therefore, with an appropriate support from a platform, numerous scientists can simultaneously and independently use these workflows to analyse and search for GWs using their own parameter sets. As the number of concurrent workflow executions grows and shrinks, the platform can automatically increase and decrease the number of infrastructural resources deployed to support the platform, in such a way that the execution time of each individual workflow is not affected by the number of running workflows. Without support for scheduling and management of data and tasks, in the worst case, the parallel execution of these workflows will be reduced to sequential execution due to insufficient resources.

Parallel executions of workflows can lead to resource contention, as each workflow instance often requires the same set of data as input, requires a specific number of compute resources, which can be limited, and are bound by deadlines set by users. Hence, the challenges are to:

1. allocate Cloud resources to tasks, workflows, and users effectively to avoid resource contention—dynamic resource provisioning problem;
2. minimize execution time of individual workflows—task/workflow scheduling problem;
3. dynamically expand or shrink Cloud services based on varying load.

In order to tackle the above challenges, we designed a scalable PaaS middleware and built a prototype system that facilitates the search for Sco X-1. This article describes the PaaS middleware design, implementation, and performance evaluation with the support of the GW data analysis application use-case. Specifically, this paper makes the following **novel contributions**:

1. **Dynamically Provisioning of Multiple PaaS middleware Pools:** Our PaaS middleware is composed of workflow engines that manage a pool of workers in the Cloud. Instances of the workflow engine can be added and removed on demand in order to adapt to the observed demand of the system.
2. **Load Balancing and Distribution:** Our system contains a layer that distributes user requests to PaaS middleware pools and maintains load balance on each pool of workers by scaling load across recently spawned PaaS middleware, releasing resources when not in use.
3. **Cloud-Enabled LIGO Software Application (LALApps):** We describe how we used a LIGO software application and executed its operations using our propose system hosted in a public Cloud infrastructure.

The remainder of the paper is organized as follows: Sect. 2 presents closely related work in workflow systems and deployment of scientific applications in Cloud computing environments. We describe the scalable system design in Sect. 3. We then present the description of the GW data analysis pipeline in Sect. 4. Using the case-study as workload, we present performance evaluation in Sect. 5. We conclude and present future directions in “Conclusions and Future Work” section.

2 Related Work

Efforts for accelerating the execution of LIGO applications in distributed systems date back to 2002 [7]. Such a project established the workflows and data access policies used for earlier generation of LIGO experiments. After the rise of Clouds as suitable platforms for execution of scientific operations, Zhang et al. [25] developed an algorithm for execution of a LIGO workflow in a public Cloud. The application differs from ours in the method used for detecting the gravitational waves, and the algorithm is customized for the particular application. Our work proposes a two-level provisioning approach to scale either the application or the workflow execution platform. Therefore, Zhang's LIGO application and the corresponding scheduling algorithm could be integrated in our proposed system. Chen et al. [6] proposed an approach for generation of virtual machine images for the LIGO project. This virtual machine images can be used by platforms (such as the one proposed in this paper) or directly by researchers wanting to deploy their LIGO application in the Cloud.

Auto scaling of Cloud services and infrastructure results in significant cost reduction, green energy use, and sustainability. Dougherty et al. [9] proposed a model-driven configuration of Cloud auto-scaling infrastructure and applied it to an e-commerce application running on Amazon EC2 platform. Mao and Humphrey [14] used auto scaling of Cloud resources to minimize deployment costs while taking into account both user performance requirements and budget concerns.

In the context of platform support for execution of Workflow applications in Clouds, Workflow Management Systems that were originally proposed for Grids, such as Pegasus [8, 18], Askalon [20], Kepler [13], Taverna [19], and Cloudbus Workflow Engine [21] were extended to support utilization of Cloud resources. However, these systems have limited scalability regarding the total number of resources and application that can be simultaneously managed by them. Therefore, our proposed architecture groups such systems in a Platform as a Service layer and enable the deployment of multiple of such engines to increase the overall system scalability. In this sense, any of the above systems could be used in the PaaS layer of our architecture, even though in this paper we used Cloudbus Workflow Engine for this purpose.

Lu et al. [12] proposed a workflow for large-scale data analytics and visualization with emphasis in spatio-temporal climate data sets that targets public Cloud environments as the source of resources for workflow execution. However, the target scenario of such a tool is one user operating over one dataset, whereas our proposed solution targets multiple users accessing multiple data sets concurrently.

Kim et al. [11] proposed a system supporting execution of workflows in hybrid Clouds. This approach differ from our proposal in the sense that the main objective of such tool is typically keeping the utilization of local infrastructure as high as possible and keep utilization of public Clouds low, in order to reduce the extra costs related to public Clouds. Such approach has also to work in the selection of workloads to be moved to the public Clouds and the workloads to be kept on premises. Furthermore, it scales only the number of workers, while our approach is able to scale the number of engines to support more simultaneous users and resources.

On the topic of automatic scaling of applications in Clouds, Vaquero et al. [23] presents a survey on the topic. It categorizes how scalability can be achieved on IaaS and PaaS Clouds. According to their classification for the problems, our work is classified as PaaS scaling via container replication.

Mao and Humphrey [15] proposes a solution for the problem of auto-scaling Clouds for execution of workflow applications. The approach considers a single workflow engine that is able to scale resources available for processing workflow applications. Our approach, on the other hand, considers a two-layers scaling approach where the number of workflow engines can also be scaled to further increase the total capacity of the system in managing and executing multiple simultaneous applications.

Casalicchio and Silvestri [5] explore different architectures for monitoring and scaling of applications in Clouds. The architectures explore different mixes of public Cloud provider services with local services for achieving scalability of VM applications. The proposed architectures operate at the IaaS layer, and utilize with arbitrary metrics for scalability decisions (for example, application throughput). The architectures are not aware of dependencies between tasks in workflow applications, and therefore they are not optimal for this type of application, unlike our approach.

Finally, it is worth noticing that public Cloud providers such as Amazon,[1] Microsoft,[2] RightScale,[3] and Rackspace[4] also offer solution for auto-scaling based on web services or APIs. They allow users to determine simple rules, typically based on monitored performance metrics (CPU and memory utilization, application response time), that trigger the auto-scaling process. Rules are used to determine the amount of machines to be added or removed from the system, typically proportional to the amount of resources in use (e.g., *increase number of resources by 20 % if average memory utilization is above 80 %*) or fixed (e.g., *reduce the number of resources to 5 if utilization is below 40 %*). Our approach enables more complex decisions that are determined algorithmically, and performed at two different levels (platform and application).

3 System Architecture and Design

In this section, we detail the design of the proposed PaaS middleware for execution of scientific workflows. Table 1 defines the symbols used in the rest of the article.

The system has a layered design in order to process multiple users and their workflows in a scalable manner, as depicted in Fig. 1. The bottommost layer is composed of virtualized resources, provided by public IaaS Cloud service providers,

[1] http://aws.amazon.com/autoscaling/.

[2] http://www.windowsazure.com/.

[3] http://www.rightscale.com/products/automation-engine.php.

[4] http://www.rackspace.com/cloud/loadbalancers/.

Table 1 Description of symbols used in the article

Symbol	Description
Q	Queue containing the list of tasks submitted to the system for execution as applications by end-users
$E_{engines}$	Set of compute resources where the workflow engine has been installed. Resources in this set compose the PaaS middleware
VM	A virtual machine deployed to support our platform
$R_{workers}$	Set of VMs that are configured to execute end-user applications
W_{pE}	Workers per Engine. This constant directs the algorithms to allocate up to the W_{pE} compute resources (workers) to each workflow engine that is running. For example if there are three engines and $W_{pE} = 5$, then each engine will have five worker VMs under its management
N_{tW}	Number of Tasks submitted to each worker. Higher values enable multiple tasks to be submitted to a worker to run in parallel
$sizeof(Array)$	This function returns the length of the array that is passed to it as a parameter
$CCE(integer)$	This is the capacity calculation algorithm. Its argument is the length of the task queue Q
$MAXCompTime$	This value signifies the maximum completion time for a task submitted by the user

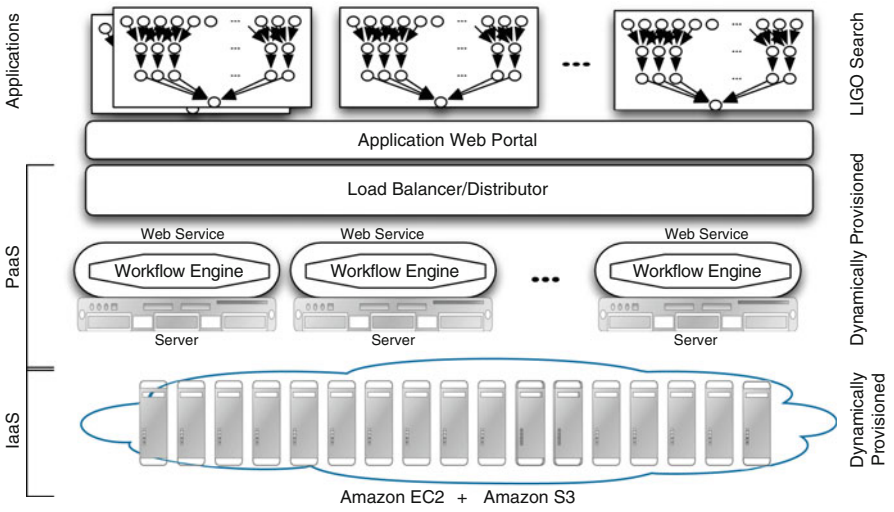

Fig. 1 Scalable PaaS middleware for scientific workflows

where the application is actually executed. In particular, for supporting LIGO data analysis we choose Amazon AWS as the public IaaS provider. The entire system is deployed on Amazon EC2, so that data transfers happen within the same Cloud provider with lower latency than when using multiple Cloud providers. Virtualized resources are managed by software components at the next level, which

we name *platform services*. To implement this layer, we use our existing middleware solution—Workflow Engine [21]—for managing application workflows submitted by end-users for execution on the Cloud resources.

Because the overhead incurred to each workflow engine increases with the number of managed workflows, better scalability and response times can be obtained if multiple workflow engines are deployed and the load is balanced among them. Thus, the next layer is composed of a Load Balancer/Distributor that is responsible for enabling dynamic scaling of the platform services. The Load Balancer can dynamically create workflow engines instances, each running on a separate VM, at run-time. The provisioning of additional platform services is based on: (a) the number of waiting jobs (the difference between user requests arriving to the server and the request-level parallelism) over a period of time, and (b) average completion time of workflow applications submitted by users.

Finally, at the topmost layer, the Application Web Portal is the interface provided to end-users, who submit workflow application execution requests and monitor their progress.

Our architecture enables independent resource scaling at two different levels—the platform level composed of workflow engine instances that can manage the actual execution of tasks—and at the infrastructure level, where resources are deployed to execute the tasks. With the coordination of platform services provisioning and compute resource provisioning at the infrastructure level, the system is able to efficiently manage multiple workflows submitted by large number of users.

Figure 2 depicts the sequential interaction among different entities in order to achieve automatic scaling of Cloud resources. The interaction starts with a user sending an authentication request to access the web portal. After the authorization is granted, the user sets the parameters of the application workflow, determines the configuration files, and submits the workflow for execution. Depending on the number of tasks in the application, the load balancer calculates the number of VMs (engines) and computing machines (workers) needed and sends the invocation request to the public Cloud resource provider, as detailed in the next section. Workers are assigned to a specific workflow engine, and therefore all the tasks executed from a worker belong to jobs managed by its corresponding engine.

Once the number of required engines and workers is defined, the Load balancer submits requests for machines to Cloud resource provider, which starts the type and number of virtual machines according to the request. Once there are enough available virtual machines to start execution of tasks, the load balancer sends tasks to the workflow engine, which in turn forwards them to its associated workers for actual execution.

The worker sends the end result to its assigned workflow engine in order to direct it to store the data on the Cloud storage. The user can monitor the process of application execution through the web portal, which is able to supply statistics such as submission time, allocated resources, execution status, total execution time, and total stage-out time. Once results of workload execution is available, the user can download it directly from the Cloud storage. During the whole process, the load balancer continuously enforces distribution of applications among workflow

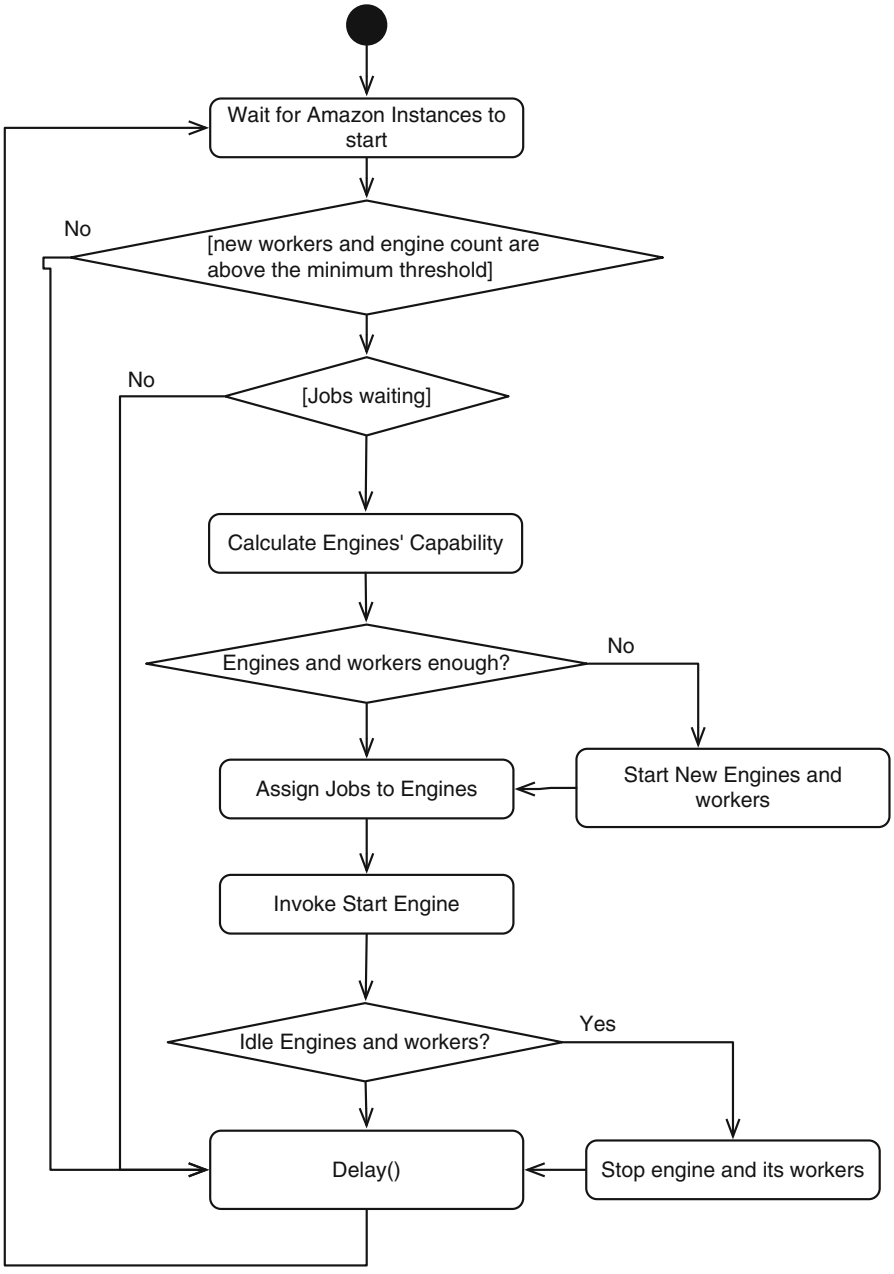

Fig. 2 Auto scaling at the PaaS layer of our proposed middleware

engines by provisioning the right amount of virtual machines (both workers and workflow engines). It does so by increasing or decreasing the number of running virtual machines based on the number of tasks to run and the capability of each VM (see Algorithms 1 and 2). Therefore, if all the submitted applications finish execution and no further application are submitted, running engines and workers are turned off automatically by the load balancer, as detailed in the next section.

Algorithm 1: PaaS load balancing algorithm

Input: W_{pE}: Application-dependent worker-per-engine rate.
while *There are incomplete Tasks in L in Q* **do**
  Update $R_{workers}$;
  Apply the CCE Algorithm to divide newly added instances between $E_{engines}$ and $R_{workers}$;
  Associate up to W_{pE} workers in $\{R_{workers}\}$ to each engine $e_i \in E_{engines}$;
  if *(* $(|R_{workers}| \geq 0)$ *OR all waiting compute resources available) AND* $(|Q| > 0)$ **then**
    Number of tasks remaining to be submitted for execution
    $nPending = CCE(sizeof(Q))$;
  if $nPending > 0$ **then**
    Workers to run $wr = nPending/N_{tW}$;
    $wPending = wr$;
    foreach *Engine* $e_i \in E_{engines}$ **do**
      Free slots for engine $e_i : es = W_{pE} - (current\_number\_of\_workers\_in\_e_i)$;
      $wPending = wPending - es$;
      if $wPending > 0$ **then**
        Engines to run $er = \lfloor wPending/W_{pE} \rfloor$;
      Provision *er* engines in the Cloud;
      Provision *wr* workers in the Cloud;
  foreach *Engine* $e_i \in E_{engines}$ **do**
    repeat
      Assign tasks in Q to e_i;
    until *current_engine reaches it maximum load*;
    Start execution of tasks assigned to engine e_i;

3.1 Load Balancing

Load balancing in our proposed architecture is managed by the *Load Balancer/ Distributor* component. This component acts both at task level, in order to balance the load of workers, and also at the middleware level, by controlling the number of running Workload Engine instances and balancing the number of jobs submitted to each engine. The general operation of the Load balancer is detailed in Algorithm 1.

When jobs are submitted to the system, their corresponding tasks are queued at the Load Balancer (LB) in a queue Q. The LB groups the running resources in

two sets: workflow engines $E_{engines}$ and computing workers $R_{workers}$. Application-dependent, user-defined W_{pE} workers are assigned to each engine running in the system. If new virtual machines were started since the last execution of the algorithm, each new VM is assigned to an already running workflow engine in a round-robin basis.

After all the provisioned VMs are ready to accept requests, the Load balancer checks for jobs waiting. Definition of number of engines to be added to the platform layer and the number of workers to be added to these engines is based on several factors. One such factor is the estimated capacity of available resources to handle extra tasks, which is determined using the method presented in Algorithm 2. The Load balancer computes the average tasks completion time observed in a configurable timespan and uses this value to estimate resource availability for the next time span. The availability estimation and number of waiting tasks are then compared in order to determine whether existing engines are enough to handle all the tasks or not.

Algorithm 2: CCE: engine capacity/load calculating algorithm

Set the capability of each free worker to N_{tW};
Set the threshold of completion time of a single task to *MAXCompTime*;
foreach $e_i \in E_{engines}$ **do**
    Get average task completion time ct_{e_i} of e_i during the last n minutes;
    Compute the availability a_i of e_i,
    $a_i = (ct_{ei} - MAXCompTime)/MAXCompTime$;
    Compute the capability c_i of e_i, $c_i = a_i \times$(number of workers of e_i)$\times$ (max tasks per worker);
Refresh compute resource status;
if *New workers are ready* **then**
    Increase the capability of its engine by N_{tW};
return *list of unassigned tasks*;

If the algorithm determines that available resources are not enough, the number of extra workers and engines is computed and the corresponding number of resources is started in the public Cloud provider. Each new engine, once ready, receives waiting tasks belonging to the same job. The assignment step is repeated until each engine reaches its maximum load or until no more waiting jobs exist. The engines then start applications that have been assigned for execution, and availability of resources is recomputed. The load balancing process is repeated until all tasks are finalized (either completed or canceled after a maximum number of failures).

The algorithm initially sets the capability of each free worker to N_{tW}, which is the maximum number of tasks that can be allocated to a worker while ensuring that the tasks can be completed in a reasonable time. The threshold of completion time of a single task is set *MAXCompTime*, which is the default threshold for determination if the engine is overloaded. If it is overloaded, it stops having tasks assigned to it. Afterwards, for each engine of $E_{engines}$, the algorithm computes the average

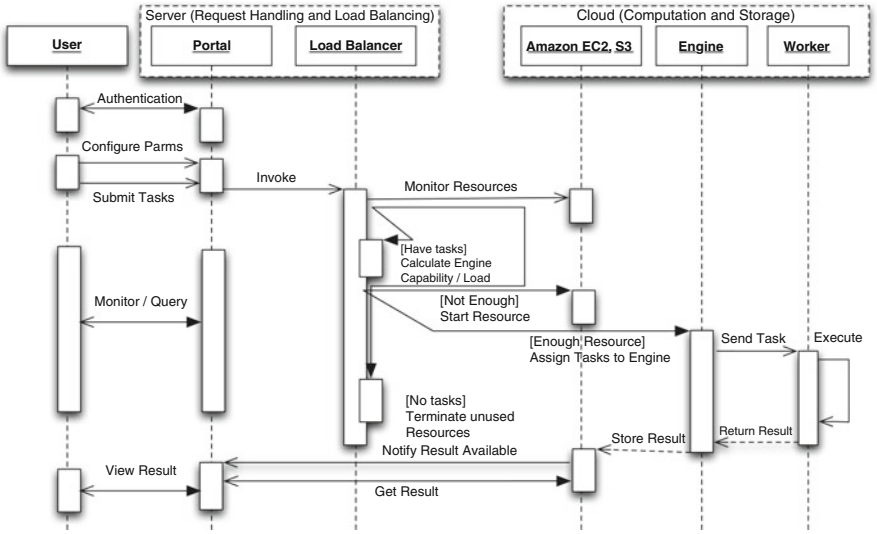

Fig. 3 Sequence diagram showing the interaction between entities involved in scaling PaaS services

completion time observed in the last time interval, so the estimated availability and capability can be computed. Based on such values, waiting tasks are assigned to running engines up to the maximum calculated capacity of each engine. Remaining tasks are taken into account when deciding to extend the number of engines (Fig. 3).

4 LIGO Data Analysis and the Search for Gravitational Waves

The system described in the previous sections was implemented and used in an application scenario: a data analysis pipeline in a gravitational wave search. The Laser Interferometer Gravitational Wave Observatory (LIGO) is one of the world's largest physics projects [2]. It will inaugurate a new era in astronomy by detecting Einstein's elusive gravitational waves, vibrations in space–time emitted by various cosmic sources. LIGO is currently the most sensitive element of an international detector network including facilities like Virgo, spaced widely around the globe to take advantage of the dramatically improved angular resolution afforded by intercontinental baselines. Sophisticated computing is the backbone of LIGO: the sheer scale of the data flows and the difficulty of detecting minuscule signals make gravitational wave searches one of the great computing challenges of our time.

To illustrate the application scenario and its requirements, we present a search for periodic GW signals from neutron starts in binary orbits [10, 17, 22]. Of this class of source, low mass X-ray binaries (LMXBs) are prime candidates due to numerous

accurate observations across the electromagnetic spectrum [22, 24]. Sco X-1, the brightest X-ray source in the sky, located in the constellation Scorpius, is likely to be the LMXB that emits GW most strongly [24]. It is targeted for the development of the search application investigated in this study.

In working towards the detection of GWs, the LIGO Data Analysis Software Working Group has built several analysis tools. The sideband search is part of the LIGO Algorithm Library (LAL).[5] We use tools in LAL and its application suite (LALapps) to generate and analyze synthetic test data. Using the LAL tools we create LIGO-like data with an injected signal and synthetic noise and run the sideband search to retrieve the injected signal. The LAL tools ensure that the synthetic data resembles the data generated by actual detectors. Real data is available, but remains proprietary for now; its analysis lies outside the scope of this paper.

The sideband search has two stages. The first stage is a matched filter known as the $\mathscr{F}$-statistic [10]. It requires knowledge of the source sky position and searches over the unknown source frequency, by comparing against a signal template via a maximum likelihood approach. It is computationally intensive. If the source is in a binary system, the $\mathscr{F}$-statistic power is smeared out over many frequency bins (sidebands), spaced by integer multiples of the orbital frequency, i.e., a frequency comb. Hence, the second stage of the sideband search involves summing up semi-coherently the output of the $\mathscr{F}$-statistic at the frequency of each sideband in the comb. This requires knowledge of the orbital period and semi-major axis but is not computationally expensive. It produces a result called $\mathscr{C}$-statistic. The frequency parameter space can be split to allow parallel distribution. However the $\mathscr{F}$-statistic and $\mathscr{C}$-statistic steps must be performed sequentially. A search pipeline for this procedure is shown in Fig. 4. Coherent follow-up (Step 3) proceeds in the event of a positive provisional detection at the end of Step 2 and may leverage other signal processing algorithms, whose details lie outside the scope of this paper.

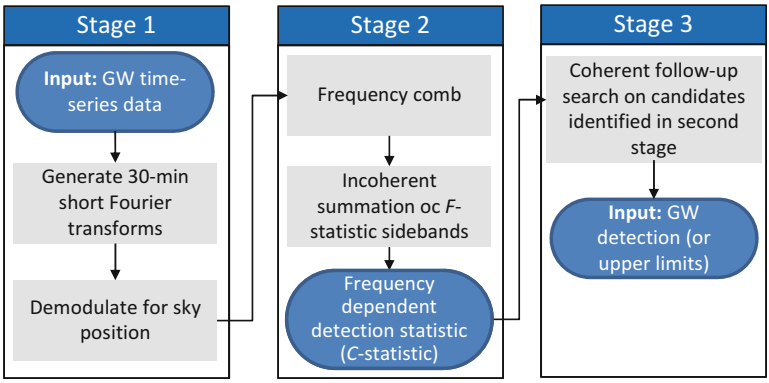

Fig. 4 The frequency comb search algorithm for periodic sources in binary systems

[5] http://www.lsc-group.phys.uwm.edu/daswg/.

Sample plots of $\mathscr{F}$- and $\mathscr{C}$-statistic output obtained from a simulated Sco X-1 search are presented in Fig. 5. The signal is injected at 505 Hz and is clearly recovered by the $\mathscr{C}$-statistic in Fig. 5d from the two-horned frequency comb structure in the $\mathscr{F}$-statistic output in Fig. 5b. Figure 5a, c shows null results from the same experiment but in a region of frequency space away from the injected frequency of the signal, where the data should contain just noise. Only a few Hz (out of a total search band of 1 kHz) are plotted for clarity.

A simple workflow for this procedure is depicted in Fig. 6. The various parameters that form the input are represented at the top of the figure. For each frequency range, one workflow task (circles in the figure) is created. For each $\mathscr{F}$-statistic computing task, a comb search is performed. Once the comb search is complete, the mean value is calculated and submitted to the last task, which provides the visualization in the form of a plot. The activity described above can be triggered multiple times by a user simultaneously considering multiple GW sources.

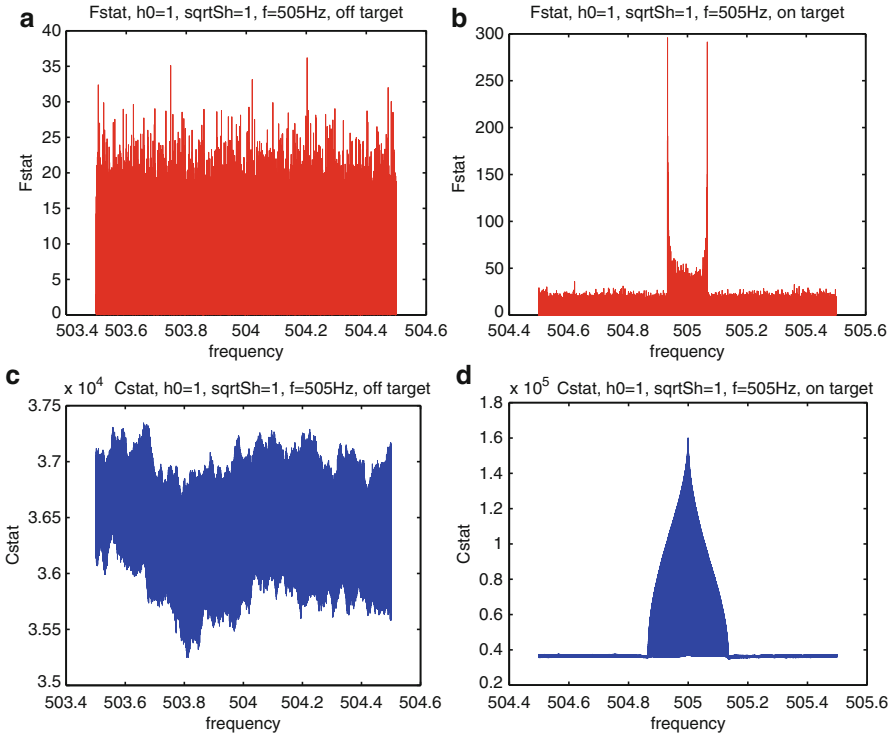

Fig. 5 $\mathscr{F}$-statistic (*top*) and $\mathscr{C}$-statistic (*bottom*) versus frequency plots obtained after processing the workflow in Fig. 4, demonstrating the noise (left) and signal (*right*) cases. The $\mathscr{F}$-statistic (**a** and **b**) calculation is the first stage of the CombSearch workflow, and also the input to the second stage, $\mathscr{C}$-statistic (**c** and **d**) calculation (Color figure online)

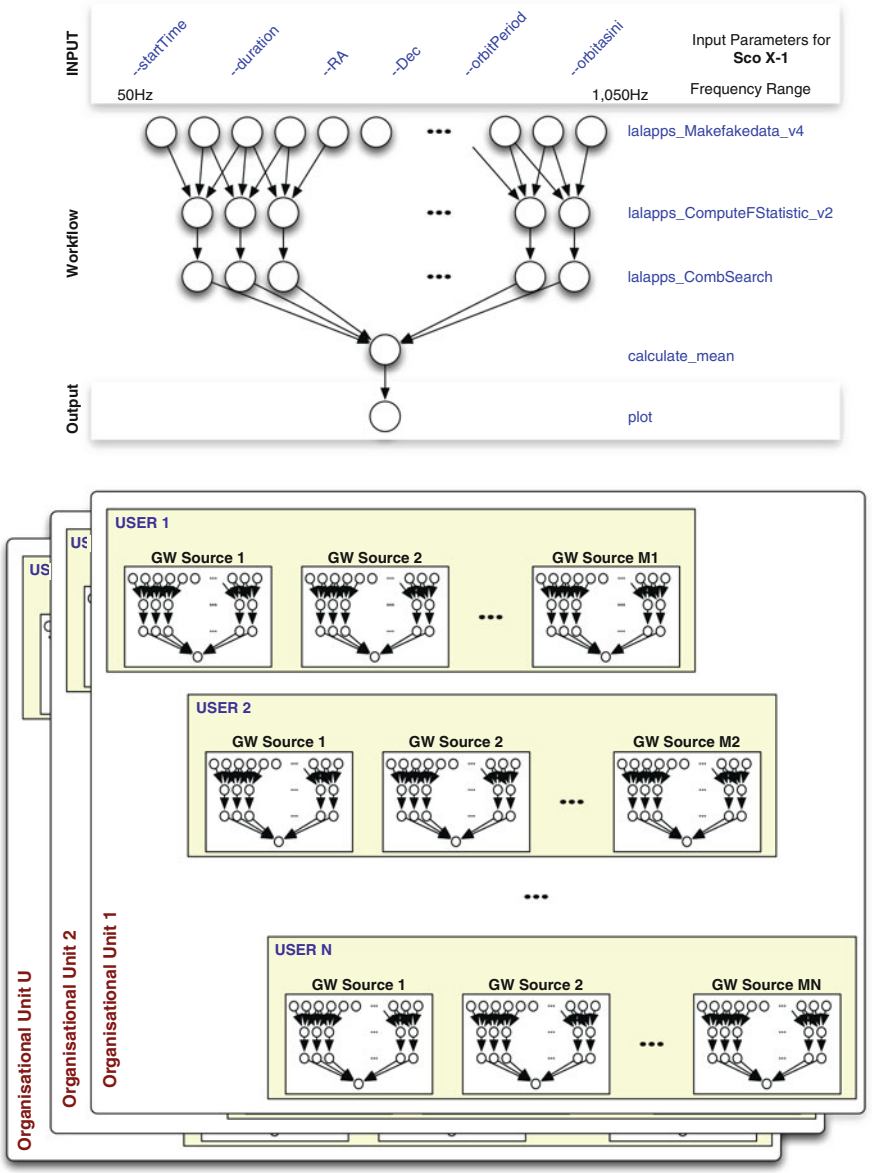

Fig. 6 A data analysis application workflow for Sco X-1 search over 1,000 Hz

The figure also depicts the fact that multiple Organizational Units (for example, research labs belonging to different universities) can have various users requesting execution of such workflow at the same time.

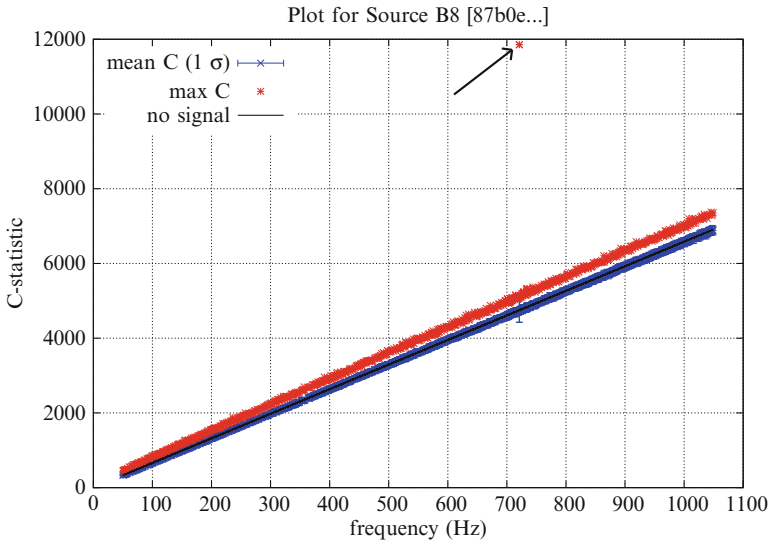

Fig. 7 $\mathscr{C}$-statistic output versus frequency after processing the workflow in Fig. 6. The *cross with error bars* represents the mean $\mathscr{C}$-statistic $+/-1$ standard deviation for each Hz band. The maximum $\mathscr{C}$-statistic from each band is indicated with *stars*. The "no signal" *black curve* refers to the theoretically expected value of the $\mathscr{C}$-statistic in the case of pure noise. The *black arrow* indicates the outlier from the $\mathscr{C}$-statistic results in the Hz band containing the signal, which was injected at 721.27 Hz (Color figure online)

As an example, a search for Sco X-1 over a 1,000 Hz band can be divided into 10^3 jobs of 1 Hz each. The result of each job is a list of $\mathscr{C}$-statistic values for every frequency bin in the 1 Hz band. The number of frequency bins is determined by the frequency resolution of the $\mathscr{F}$-statistic, $(2T)^{-1}$, which is a function of the observation time T. In our example, for T =10 days, each 1 Hz band contains $\sim 10^6$ $\mathscr{C}$-statistic values.

To test for a detection, we calculate the $\mathscr{C}$-statistic for each bin in a 1 Hz band. For each band, the mean, standard deviation, minimum, and maximum values are collected and used for plotting the output. Figure 7 shows the output from a simulated search for a signal injected at 721.27 Hz with strength $h_0 = 1.6 \times 10^{-23}$ and noise $\sqrt{S_h} = 6 \times 10^{-23}$ across the 50–1,050 Hz band. Mean $\mathscr{C}$-statistic values are shown as crosses and the solid black line indicates the expected signal-free result. The plot also shows the maximum $\mathscr{C}$-statistic value in each band (stars). Since the injected signal is narrow band, it only appears in $\preceq 10$ bins (out of 10^6 per Hz), so the maximum $\mathscr{C}$ is a better diagnostic than the mean. The maximum $\mathscr{C}$-statistic values shows a clear outlier in the region of the injected signal around 721 Hz, highlighted by the black arrow.

4.1 Application Requirements

Data Requirements The size and quantity of data produced by the workflow depicted in Fig. 6 are substantial. In our test example, which is deliberately chosen to be small, if 10 days of synthetic data is generated by LAL, the workflow must handle 480 files of size 142 KB each, each file is a 1,800-second Short-time Fourier Transform (SFT). The total volume of data generated depends on the search duration chosen by LIGO scientists. The *ComputeFStatistic* and *CombSearch* scripts each produce 77 MB of data after processing the synthetic data. Depending on the input parameters, the result obtained after plotting the points (106 points in a single file; points are FStat-frequency and CStat-frequency, as depicted in Fig. 5) may need further processing to produce an image file (e.g. png, eps, etc.) for visualization.

It is vital to emphasize that these data volumes are small because we restrict ourselves to a small test problem in this paper. In general, full LIGO searches involve petabytes of data. The LIGO detectors sample the gravitational-wave signal channel at 16 kHz continuously for several years and generate another $\sim 10^4$ environmental channels sampled at similar rates. A typical compact binary coalescence search must process all the environmental channels, as must the pre-processing scripts that generate the SFTs for Sco X-1-like searches. The LIGO data storage requirements are determined by the rate at which data is produced by the LIGO interferometers. Each advanced LIGO interferometer is expected to produce a total data rate of $\sim$10 MB/s. This corresponds to an annual data volume of $\sim$315 TB or $\sim$200 TB with best current compression. The Advanced LIGO computing plan calls for each interferometer to maintain an archive of its own raw data as well as copies of the raw data generated by the other two interferometers. Additionally, a separate redundant archive of the raw data is to be maintained at each Tier 1 data center [2, 3].

Computational Requirements The processing time taken by *ComputeFStatistic* and *CombSearch* is around 9 min on an Intel dual core 2 GHz CPU with 7 GB memory when executed for a single source across the 10-day stretch of data with a band of 1 Hz.

Multiple GW Sources and Multi-User Environment As noted in the introduction, GWs can be detected from multiple sources. Users may elect to search for multiple sources or single sources across different sections of the data. Each source has different input parameters even though the underlying workflow is the same. This scenario is depicted in Fig. 6 as GW Source 1–GW Source N. In a LIGO organizational unit, it is expected that many users conduct searches simultaneously on different sources. In an organizational unit, e.g., a university research group, there are many users conducting different search procedures on the same and/or different GW sources. These users are depicted as User 1–User N in Fig. 6.

Execution Time in Clouds In order to evaluate the expected execution time of the application in public Clouds, we executed the search procedure on Amazon EC2, an IaaS service enabling users to buy virtual machines (instances) with specific characteristics in terms of CPU, memory, and storage (instance types). We repeated

the experiment with different instance types (Small, Large, and Extra Large). The synthetic input data was generated using *lalapps_Makefakedata_v4*. The execution times of the applications and their input/output file sizes are reported in Table 2. The values listed in Table 2 show that the application's runtime is CPU-intensive, and therefore dependent on the machine processing capacity and directly affected by the computing power of the resource in use. Furthermore, the size of data produced will help us in identifying techniques to manage the transfer and storage of such large data sets.

Table 2 Execution characteristics on Amazon AWS

Task names	Execution time (s)			Input size (KB)	Output size
	Small	Large	Extra large		
Generate data	37	17	12	–	138 kB
Compute F-statistic	2,611	1202	883	138 KB	480 MB
Comb search	184	85	N/A	480 KB	76 MB
Plot results	32	15	N/A	138 KB	100 KB

5 Performance Evaluation

In this section, we present the experiments conducted for evaluating the performance of the system design and the load balancing algorithms. We divide the experiments into two groups, namely *Platform scalability* and *Dynamic provisioning and instantiation of compute resources*.

The PaaS system and the workflow application was demonstrated at the Fourth IEEE International Scalable Computing Challenge (SCALE 2011), in California, USA, during May 23–26, 2011. The experimental results presented in this section are a result of the data collected from the executions during and after the challenge.

5.1 Platform Scalability

As discussed earlier, the PaaS middleware (the Workflow Engine) has limitations on the number of workers and concurrent workflow applications it can manage. In an environment such as that of the LIGO project, where it is expected that multiple users from multiple organizations will be simultaneously performing GW searches, a single engine can become a bottleneck for the scalability of the solution. To tackle such a limitation, our proposed system is able to dynamically scale the PaaS layer and also the worker pools by deploying multiple Engines when the demand is high.

In order to evaluate the dynamic scaling of PaaS services work, a first series of experiments was conducted. The experiments consist in a series of execution of the application described in the previous section with a fixed number of compute sources and different combination of engines and workers numbers. The application conducts the search for GW signals between the frequencies of 50 and 1,050 Hz, performing both the full-range (1,000 Hz) and proximity (within 200 Hz intervals) searches.

The maximum number of tasks executed was 40. Moreover, in order to enable us to acquire a better understanding of the practical environment, we use different sources for each experiment. Experiments were performed with a maximum of one, two, and four engines. For each number of engines, experiments were executed with 4, 8 and 16 workers per engine.

Figures 8, 9, and 10 show respectively results when the system was allowed to scale up to one, two, and four engines. The topmost plot on each figures shows the execution time of individual tasks, while the bottom plots show start and finish time of each task with different scaling levels related to number of workers.

Figure 8 shows that, when a single workflow engine is available, execution time of tasks when only four workers are deployed is smaller than when 16 workers are deployed. Also, there are bigger variation in the execution time when 16 workers are in use. This demonstrates the limitation of a single workflow engine in managing too many concurrent workers, caused by overheads related to the management of

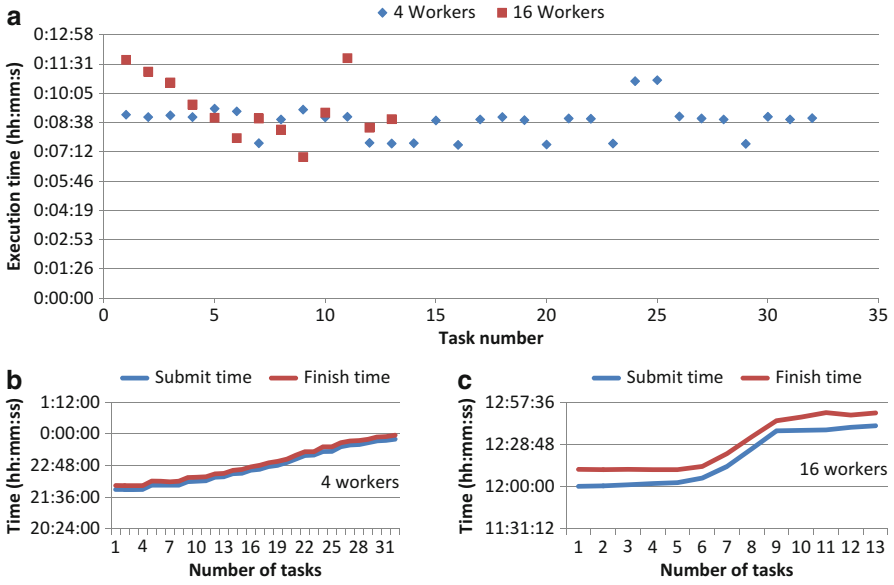

Fig. 8 Performance of a single workflow engine. (**a**) Execution time of tasks. (**b** and **c**) Submission and finish time of tasks for 4 and 16 workers, respectively

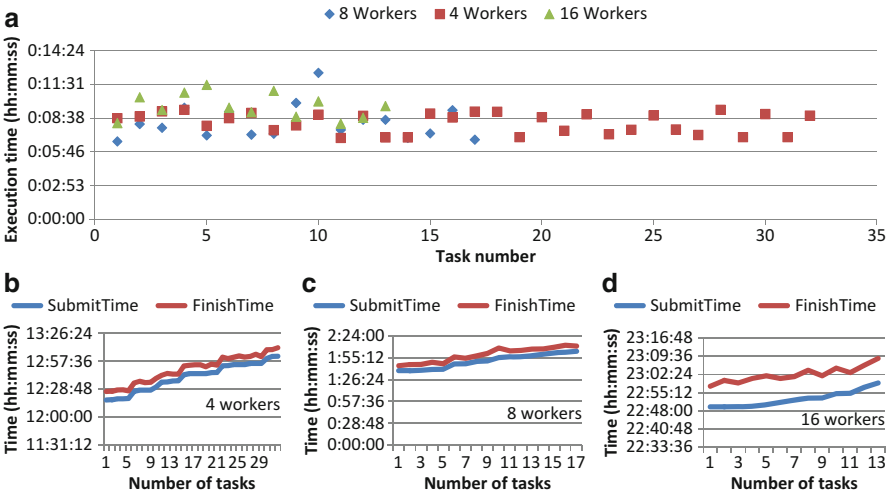

Fig. 9 Performance of two concurrent workflow engines. (**a**) Execution time of tasks. (**b–d**) Submission and finish time of tasks for 4, 8, and 16 workers, respectively

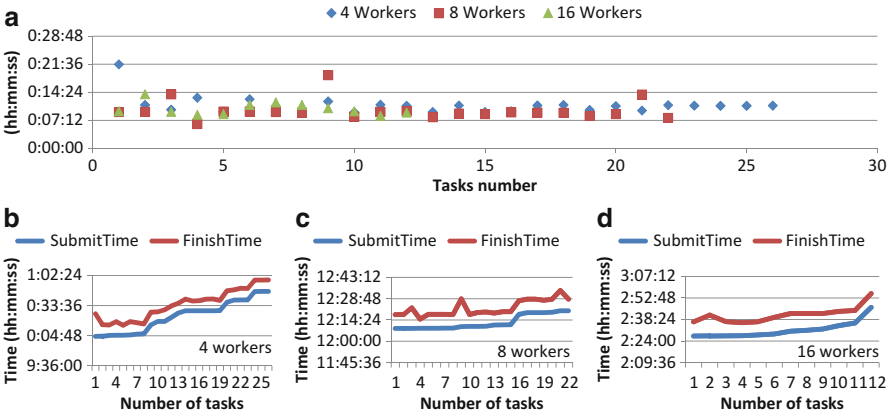

Fig. 10 Performance of four concurrent workflow engines. (**a**) Execution time of tasks. (**b–d**) Submission and finish time of tasks for 4, 8, and 16 workers, respectively

multiple workers. for the scenario with one workflow engine, utilization of only four workers makes execution time of tasks more homogeneous. The same trend is observed with two and four simultaneous workflow engines, as shown in Figs. 9 and 10, respectively. When the ratio of workers per engine is low, increase in the number of concurrent engines reduces tasks runtime.

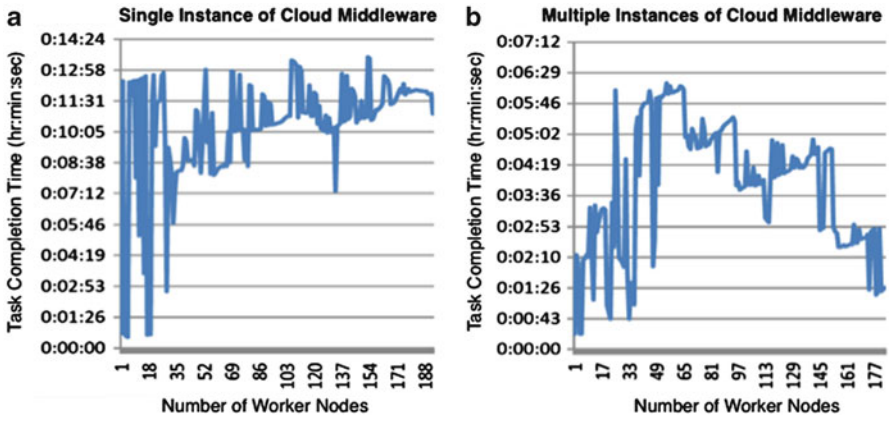

Fig. 11 (**a**) A single workflow engine handles all the user requests. (**b**) Multiple workflow engines are "dynamically" provisioned based on user requests, thus forming resource pools

5.2 Dynamic Provisioning of Workers

The next set of experiments aimed at evaluating the performance of the mechanism for dynamic provisioning of workers. The experiment consisted in the execution of the same application used in the previous experiments with an increasing number of tasks in order to stress the system, triggering the dynamic provisioning process.

The graphs presented in Fig. 11 shows tasks completion time as a function of number of resources and the number of workflow engines running. Due to overheads cause by monitoring and management of workers, as the number of workflow tasks increases until a maximum value of 40, the task completion time increases, which in turn triggers the instantiation of new compute resources (workers). Figure 11a shows the completion time of tasks when the number of workers increases linearly, with only one workflow middleware handling all the requests. It can be noticed that, in this case, the completion time steeply falls when more compute resources are added. However, we also observed that efficiency is constantly decreasing. For instance, the completion time when there are 170 workers is around 11 min, as compared to around 5 min when there are 35 resources. Although the ratio of tasks to workers is the same, a single PaaS middleware introduces higher overheads for a large number of tasks and workers, which affects the completion time. The same trend is not observed when multiple workflow engines are deployed.

In contrast, in Fig. 11b, we can observe that the completion time decreases for the same task to worker ratio as in Fig. 11a. As the completion time starts to climb, we instantiate a new workflow engine, which has an immediate impact on the completion times of new tasks. This effect is visible as a ladder-like curve in Fig. 11b. When 170 workers are instantiated, the completion time is around 2 min 20 s, nearly five times less than in Fig. 11a. Multiple workflow engines (PaaS middleware) divide the overheads of scheduling and execution.

Conclusions and Future Work

Cloud computing is a promising technology for transparently managing the challenges brought by large-scale research applications that are characterized by large volumes of data, computationally demanding applications, and execution concurrency. However, such scientific applications demand specialized platform tools capable of coordinating the different stages of execution of tasks while optimizing user-defined deadlines and Cloud usage cost. Because platforms for scientific applications supporting such features are not readily available, we designed and implemented an automated PaaS middleware that uses public IaaS providers to host and support scalable execution of scientific application workflows.

The proposed middleware is able to independently manage and scale the platform layer composed of workflow engines and the infrastructure layer composed of worker units able to execute application tasks. The scalable PaaS middleware architecture was described, algorithms for load balance and scaling were presented, and an application case study in the area of particle physics was presented. The application is a search for gravitational waves from the LIGO project, and workflows of such project were executed in a prototype of the discussed architecture in order to validate our approach and enable us to evaluate the systems performance. Results show that our goals of independent and automated scaling of different layers is achievable and enable reduction in execution time of applications even with variable pattern and size of user requests.

As future work, we intend to evaluate the impact of different types of applications in the performance of our proposed architecture. We also plan to enhance the load balance algorithm and extend them to support execution of workflows in multi-cloud scenarios, where resources from different Cloud providers, both public and private, are used at the same time in a federated environment.

Acknowledgements This project is partially supported by project grants from the University of Melbourne (Sustainable Research Excellence Implementation Fund and Melbourne School of Engineering) and the Australian Research Council (ARC). We thank Amazon for providing access to their Cloud infrastructure, the Australian and international LIGO communities for their guidance and support, and Dong Leng for his contribution towards extending the Workflow Engine for the LIGO experiment.

References

1. Large scale computing and storage requirements for basic energy sciences research. Workshop Report LBNL-4809E, Lawrence Berkeley National Laboratory, USA, Jun. 2011.
2. B. P. Abbott et al. LIGO: the laser interferometer gravitational-wave observatory. *Reports on Progress in Physics*, 72(7):076901, Jul. 2009.

3. Advanced LIGO Team. Advanced ligo reference design. Technical Report LIGO M060056-08-M, LIGO Laboratory, USA, May 2007.
4. Lars Bildsten. Gravitational radiation and rotation of accreting neutron stars. *The Astrophysical Journal Letters*, 501(1):L89–L93, Jul. 1998.
5. E. Casalicchio and L. Silvestri. Architectures for autonomic service management in cloud-based systems. In *Proceedings of the 2011 IEEE Symposium on Computers and Communications (ISCC'11)*, 2011.
6. Wei Chen, Junwei Cao, and Ziyang Li. Customized virtual machines for software provisioning in scientific clouds. In *Proceedings of the 2nd International Conference on Networking and Distributed Computing (ICNDC'11)*, 2011.
7. Ewa Deelman et al. GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, 2002.
8. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, Jul. 2005.
9. B. Dougherty, J. White, and D. C. Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2):371–378, Feb. 2012.
10. Piotr Jaranowski, Andrzej Królak, and Bernard F. Schutz. Data analysis of gravitational-wave signals from spinning neutron stars: The signal and its detection. *Physics Review D*, 58(6), Aug. 1998.
11. Hyunjoo Kim, Yaakoub el Khamra, Ivan Rodero, Shantenu Jha, and Manish Parashar. Autonomic management of application workflows on hybrid computing infrastructure. *Scientific Programming*, 19(2–3):75–89, Jun. 2011.
12. Sifei Lu, Reuben Mingguang Li, William Chandra Tjhi, Long Wang, Xiaorong Li, Terence Hung, and Di Ma. A framework for cloud-based large-scale data analytics and visualization: Case study on multiscale climate data. In *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science (CloudCom'11)*, 2011.
13. 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, Aug. 2006.
14. M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011.
15. Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011.
16. A. Melatos and D. J. B. Payne. Gravitational radiation from an accreting millisecond pulsar with a magnetically confined mountain. *The Astrophysical Journal*, 623(2):1044–1050, Apr. 2005.
17. C. Messenger and G. Woan. A fast search strategy for gravitational waves from low-mass x-ray binaries. *Classical and Quantum Gravity*, 24(19):S469–S480, 2007.
18. Ashish Nagavaram, Gagan Agrawal, Michael A. Freitas, and Kelly H. Telu. A cloud-based dynamic workflow for mass spectrometry data analysis. In *Proceedings of the 7th IEEE International Conference on eScience (eScience'11)*, 2011.
19. Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, Nov. 2004.
20. Simon Ostermann, Radu Prodan, and Thomas Fahringer. Extending grids with cloud resource management for scientific computing. In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID'09)*, 2009.

21. S. Pandey, D. Karunamoorthy, and R. Buyya. Workflow engine for clouds. In R. Buyya, J. Broberg, and A.Goscinski, editors, *Cloud Computing: Principles and Paradigms*, chapter 12, pages 321–344. Wiley, 2011.
22. Stuart L. Shapiro and Saul A. Teukolsky. *Black holes, white dwarfs, and neutron stars: The physics of compact objects*. Wiley-Interscience, New York, USA, 1983.
23. Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Computer Communication Review*, 41(1):45–52, Jan. 2011.
24. Anna L. Watts, Badri Krishnan, Lars Bildsten, and Bernard F. Schutz. Detecting gravitational wave emission from the known accreting neutron stars. *Monthly Notices of the Royal Astronomical Society*, 389(2):839–868, 2008.
25. Fan Zhang, Junwei Cao, Kai Hwang, and Cheng Wu. Ordinal optimized scheduling of scientific workflows in elastic compute clouds. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'11)*, 2011.