A Comprehensive Perspective on Pilot-Job Systems

Matteo Turilli
RADICAL Laboratory, ECE
Rutgers University
New Brunswick, NJ, USA
matteo.turilli@rutgers.edu

Mark Santcroos
RADICAL Laboratory, ECE
Rutgers University
New Brunswick, NJ, USA
mark.santcroos@rutgers.edu

Shantenu Jha*
RADICAL Laboratory, ECE
Rutgers University
New Brunswick, NJ, USA
shantenu.jha@rutgers.edu

ABSTRACT

Pilot-Job systems play an important role in supporting distributed scientific computing. They are used to consume more than 700 million CPU hours a year by the Open Science Grid communities, and by processing up to 1 million jobs a day for the ATLAS experiment on the Worldwide LHC Computing Grid. With the increasing importance of task-level parallelism in high-performance computing, Pilot-Job systems are also witnessing an adoption beyond traditional domains. Notwithstanding the growing impact on scientific research, there is no agreement upon a definition of Pilot-Job system and no clear understanding of the underlying abstraction and paradigm. Pilot-Job implementations have proliferated with no shared best practices or open interfaces and little interoperability. Ultimately, this is hindering the realization of the full impact of Pilot-Jobs by limiting their robustness, portability, and maintainability. This paper offers a comprehensive analysis of Pilot-Job systems critically assessing their motivations, evolution, properties, and implementation. The three main contributions of this paper are: (i) an analysis of the motivations and evolution of Pilot-Job systems; (ii) an outline of the Pilot abstraction, its distinguishing logical components and functionalities, its terminology, and its architecture pattern; and (iii) the description of core and auxiliary properties of Pilot-Jobs systems and the analysis of seven exemplar Pilot-Job implementations. Together, these contributions illustrate the Pilot paradigm, its generality, and how it helps to address some challenges in distributed scientific computing.

Keywords

Pilot-Job, distributed applications, distributed systems.

1. INTRODUCTION

*Corresponding author.

Pilot-Jobs provide a multi-stage mechanism to execute workloads. Resources are acquired via a placeholder job and subsequently assigned to workloads. Pilot-Jobs are having a high impact on scientific and distributed computing [1]. They are used to consume more than 700 million CPU hours a year [2] by the Open Science Grid (OSG) [3, 4] communities, and process up to 1 million jobs a day [5] for the ATLAS experiment [6] on theLarge Hadron Collider (LHC) [7] Computing Grid (WLCG) [8, 9]. A variety of Pilot-Job systems are used on distributed computing infrastructures (DCI): Glidein/GlideinWMS [10, 11], the Coaster System [12], DIANE [13], DIRAC [14], PanDA [15], GWPilot [16], Nimrod/G [17], Falkon [18], MyCluster [19] to name a few.

A reason for the success and proliferation of Pilot-Job systems is that they provide a simple solution to the rigid resource management model historically found in high-performance and distributed computing. Pilot-Jobs break free of this model in two ways: (i) by using late binding to make the selection of resources easier and more effective [20–22]; and (ii) by decoupling the workload specification from the management of its execution. Late binding results in the ability to utilize resources dynamically, i.e., the workload is distributed onto resources only when they are effectively available. Decoupling workload specification and execution simplifies the scheduling of workloads on those resources.

In spite of the success and impact of Pilot-Jobs, we perceive a problem: the development of Pilot-Job systems has not been grounded on an analytical understanding of underpinning abstractions, architectural patterns, or computational paradigms. The properties and functionalities of Pilot-Jobs have been understood mostly, if not exclusively, in relation to the needs of the containing software systems or on use cases justifying their immediate development.

These limitations have also resulted in a fragmented software landscape, where many Pilot-Job systems lack generality, interoperability, and robust implementations. This has led to a proliferation of functionally equivalent systems motivated by similar objectives that often serve particular use cases and target particular resources.

Addressing the limitations of Pilot systems while improving our general understanding of Pilot-Job systems is a priority due to the role they will play in the next generation of high-performance computing. Most existing high-performance system software and middleware are designed to support the execution and optimization of single tasks. Based on their current utilization, Pilot-Jobs have the potential to sup-

port the growing need for scalable task-level parallelism and dynamic resource management in high-performance computing [12, 18, 23].

The causes of the current status quo of Pilot-Job systems are social, economic, and technical. While social and economic considerations may play a determining role in promoting fragmented solutions, this paper focuses on the technical aspects of Pilot-Jobs. We contribute a critical analysis of the current state of the art describing the technical motivations and evolution of Pilot-Job systems, their characterizing abstraction (the Pilot abstraction), and the properties of their most representative and prominent implementations. Our analysis will yield the Pilot paradigm, i.e., the way in which Pilot-Jobs are used to support and perform distributed computing.

The remainder of this paper is divided into four sections. §2 offers a description of the technical motivations of Pilot-Job systems and of their evolution.

In §3, the logical components and functionalities constituting the Pilot abstraction are discussed. We offer a terminology consistent across Pilot-Job implementations, and an architecture pattern for Pilot-Jobs systems is derived and described.

In §4, the focus moves to Pilot-Job implementations and to their core and auxiliary properties. These properties are described and then used alongside the Pilot abstraction and the pilot architecture pattern to describe and compare exemplar Pilot-Job implementations.

In §5, we outline the Pilot paradigm, arguing for its generality, and elaborating on how it impacts and relates to both other middleware and applications. Insight is offered about the future directions and challenges faced by the Pilot paradigm and its Pilot-Job systems.

2. EVOLUTION OF PILOT-JOB SYSTEMS

Three aspects of Pilot-Jobs are investigated in this paper: the Pilot-Job system, the Pilot-Job abstraction, and the Pilot-Job paradigm. A Pilot-Job system is a type of software, the Pilot-Job abstraction is the set of properties of that type of software, and the Pilot-Job paradigm is the way in which Pilot-Job systems enable the execution of workloads on resources. For example, DIANE is an implementation of a Pilot-Job system; its components and functionalities are elements of the Pilot-Job abstraction; and the type of workloads, the type of resources, and the way in which DIANE executes the former on the latter are features of the Pilot-Job paradigm.

This section introduces Pilot-Job systems by investigating their technical origins and motivations alongside the chronology of their development.

2.1 Technical Origins and Motivations

Five features need elucidation to understand the technical origins and motivations of Pilot-Job systems: task-level distribution and parallelism, master-worker pattern, multi-tenancy, multi-level scheduling, and resource placeholding. Pilot-Job systems coherently integrate resource placeholders, multi-level scheduling, and coordination patterns to enable task-level distribution and parallelism on multi-tenant resources. The analysis of each feature clarifies how Pilot-Job systems support the execution of workloads comprised of multiple tasks on one or more distributed machine.

Task-level distribution and parallelism on multiple resources can be traced back to 1922 as a way to reduce the time to solution of differential equations [24]. In his Weather Forecast Factory [25], Lewis Fry Richardson imagined distributing computing tasks across 64,000 "human computers" to be processed in parallel. Richardson's goal was exploiting the parallelism of multiple processors to reduce the time needed for the computation. Today, task-level parallelism is commonly adopted in weather forecasting on modern high performance machines[1] as computers. Task-level parallelism is also pervasive in computational science [26] (see Ref. [27] and references therein).

Master-worker is a coordination pattern commonly used for distributed computations [28–32]. Submitting tasks to multiple computers at the same time requires coordinating the process of sending and receiving tasks; of executing them; and of retrieving and aggregating their outputs [33]. In the master-worker pattern, a "master" has a global view of the overall computation and of its progress towards a solution. The master distributes tasks to multiple "workers", and retrieves and aggregates the results of each worker's computation. Alternative coordination patterns have been devised, depending on the characteristics of the computed tasks but also on how the system implementing task-level distribution and parallelism has been designed [34].

Multi-tenancy defines how high-performance machines are exposed to their users. Job schedulers, often called "batch queuing systems" [35] and first used in the time of punched cards [36, 37], adopt the batch processing concept to promote efficient and fair resource sharing. Job schedulers enable users to submit computational tasks called "jobs" to a queue. The execution of these jobs is delayed waiting for the required amount of the machine's resources to be available. The extent of delay depends on the number, size, and duration of the submitted jobs, resource availability, and policies (e.g., fair usage).

The resource provisioning of high-performance machines is limited, irregular, and largely unpredictable [38–41]. By definition, the resources accessible and available at any given time can be fewer than those demanded by all the active users. The resource usage patterns are also not stable over time and alternating phases of resource availability and starvation are common [42, 43]. This landscape has promoted continuous optimization of the resource management and the development of alternative strategies to expose and serve resources to the users.

Multi-level scheduling is one of the strategies used to improve resource access across high-performance machines. In multi-level scheduling, a global scheduling decision results from a set of local scheduling decisions [44, 45]. For example, an application submits tasks to a scheduler that schedules those tasks on the schedulers of individual high-performance machines. While this approach can increase the scale of applications, it also introduces complexities across resources, middleware, and applications.

Several approaches have been devised to manage these complexities [46–54] but one of the persistent issues is the increase of the implementation burden imposed on applications. For example, in spite of progress made by grid computing [55, 56]

[1] A high-performance machine indicates a cluster of computers delivering higher performances than single workstations or desktop computers, or a resource with adequate performance to support multiple science and engineering applications concurrently.

to transparently integrate diverse resources, most of the requirements involving the coordination of task execution still reside with the applications [57–59]. This translates into single-point solutions, extensive redesign and redevelopment of existing applications when adapted to new use cases or new high-performance machines, and lack of portability and interoperability.

Resource placeholders are used as a pragmatic solution to better manage the complexity of executing applications. A resource placeholder decouples the acquisition of compute resources from their use to execute the tasks of an application. For example, resources are acquired by scheduling a job onto a high-performance machine which, when executed, is capable of retrieving and executing application tasks itself.

Resource placeholders bring together multi-level scheduling and task-level distribution and parallelism. Placeholders are scheduled on one or more machines and then multiple tasks are scheduled at the same time on those placeholders. Tasks can then be executed concurrently and in parallel when the placeholders covers multiple compute resources. The master-worker pattern is often an effective choice to manage the coordination of tasks execution.

It should be noted that resource placeholders also mitigate the side-effects of multi-tenancy. A placeholder still spends a variable amount of time waiting to be executed on a high-performance machine, but, once executed, the application exerts total control over the placeholder resources. In this way, tasks are directly scheduled on the placeholder without competing with other users for the same resources.

Resource placeholders are programs with specific queuing and scheduling capabilities. They rely on jobs submitted to a high-performance machine to execute a program with diverse capabilities. For example, jobs usually execute non interactive programs, but users can submit jobs that execute terminals, debuggers, or other interactive software.

2.2 Chronological Evolution

Figure 1 shows the introduction of Pilot-Job systems over time alongside some of the defining milestones of their evolution.[2] This is an approximated chronology based on the date of the first publication, or when publications are not available, on the date of the systems' code repository.

The evolution of Pilot-Job systems began with the implementation of resource placeholders to explore application-side task scheduling and high-throughput task execution. Prototypes of Pilot-Job systems followed, eventually evolving into production-grade systems supporting specific types of applications and high-performance machines. Recently, Pilot systems have been employed to support a wide range of workloads and applications (e.g., MPI, data-driven workflows, tightly and loosely coupled ensembles), and more diverse high-performance machines (e.g., MPI, data-driven workflows, tightly and loosely coupled ensembles).

AppLeS (Application Level Schedulers) [61] offered an early implementation of resource placeholders. Developed around 1997, AppLeS provided an agent that could be embedded into an application to acquire resources and to schedule tasks onto them. AppLeS provided application-level scheduling

[2]To the best of the authors' knowledge, the term "pilot" was first coined in 2004 in the context of the WLCG Data Challenge [8,9], and then introduced in writing as "pilot-agent" in a 2005 LHCb report [60].

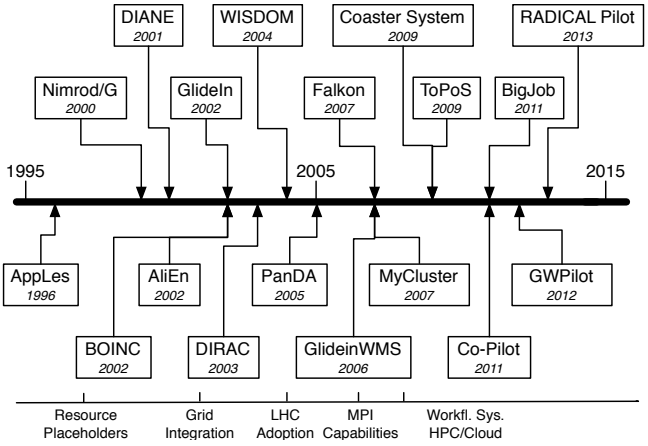

Figure 1: Introduction of Pilot-Job systems over time alongside some exemplar milestones of their evolution. When available, the date of first mention in a publication or otherwise the release date of software implementation is used.

but did not isolate the application from resource acquisition. Any change in the agent directly translated into a change of the application code. AppLeS Templates [62] was developed to address this issue, each template representing a class of applications (e.g., parameter sweep [63]) that could be adapted to the requirements of a specific realization.

Volunteer computing projects started around the same time as AppLeS was introduced. In 1997, the Great Internet Mersenne Prime Search effort [64], shortly followed by distributed.net [65] competed in the RC5-56 secret-key challenge [66]. In 1999, the SETI@Home project [67] was released to the public to analyze radio telescope data. The Berkeley Open Infrastructure for Network Computing (BOINC) framework [68] grew out of SETI@Home in 2002 [69], becoming the *de facto* standard framework for volunteer computing.

Volunteer computing implements a client-server architecture to achieve high-throughput task execution. Users install a client on their own workstation and then the client pulls tasks from the Server when CPU cycles are available. Each client behaves as a sort of resource placeholder, one of the core features of a Pilot-Job system as seen in §2.1.

HTCondor (formerly known as Condor) is a distributed computing framework [70] with a resource model similar to that of volunteer computing. Developed around 1988, Condor enabled users to execute tasks on a resource pool made of departmental Unix workstations. In 1996 Flocking [71] implemented task scheduling over multiple Condor resource pools and, in 2002, "Glidein" [72] added grid resources to Condor pools via resource placeholders.

Several Pilot-Job systems were developed alongside Glidein to benefit from the high-throughput and scale promised by grid resources. Around 2000, Nimrod/G [17] extended the parameterization engine of Nimrod [73] with resource placeholders. Four years later, the WISDOM (wide in silico docking on malaria) [74] project developed a workload manager that used resource placeholders on the EGEE (Enabling Grids for E-Science in Europe) grid [75] to compute the docking of multiple compounds, i.e. the molecules.

The success of grid-based Pilot-Job systems and especially of Glidein reinforced the relevance of resource placeholders to

enable scientific computation but their implementation also highlighted two main challenges: user/system layer isolation, and application development model. For example, Glidein allowed for the user to manage resource placeholders directly but machine administrators had to manage the software required to create the resource pools. Application-wise, Glidein enabled integration with application frameworks but did not programmatically support the development of applications by means of dedicated APIs and libraries.

Concomitant and correlated with the development of LHC [76] there was a "Cambrian Explosion" of Pilot-Job systems. Approximately between 2001 and 2006, five major Pilot systems were developed: DIstributed ANalysis Environment (DIANE) [77, 78], ALIce ENvironmen (AliEn) [79, 80], Distributed Infrastructure with Remote Agent Control (DIRAC) [81,82], Production and Distributed Analysis (PanDA) [83], and Glidein Workload Management System (GlideinWMS) [84,85]. These Pilot-Job systems were developed to serve user communities and experiments at the LHC: DIRAC is being developed and maintained by the LHCb experiment [86]; AliEn by ALICE [87]; PanDA by ATLAS; and GlideinWMS by the US national group [88] of the CMS experiment [89].

The LHC Pilot-Job systems have been designed to be functionally very similar, work on almost the same underlying infrastructure, and serve applications with very similar characteristics. Around 2011, these similarities enabled CoPilot [90,91] to support the execution of resource placeholders on cloud and volunteer computing [92] resource pools for all the LHC experiments.

Pilot-Job systems development continued to support research, resources, middleware, and frameworks independent from the LHC experiments. ToPoS (Token Pool Server) [93] was developed around 2009 by SARA (Stichting Academisch Rekencentrum Amsterdam) [94]. ToPoS mapped tasks to tokens and distributed tokens to resource placeholders. A REST interface was used to store task definitions avoiding the complexities of the middleware of high-performance machines [95].

Developed around 2011, BigJob [96] (now re-implemented as RADICAL-Pilot [23]) supported task-level parallelism on HPC machines. BigJob extended pilots to also hold data resources exploring the notion of "pilot-data" [97] and uses an interoperability library called "SAGA" (Simple API for Grid Applications) to work on a variety of computing infrastructures [96, 98, 99]. BigJob also offered application-level programmability of distributed applications and their execution.

GWPilot [100] built upon the GridWay meta-scheduler [101] to implement efficient and reliable scheduling algorithms. Developed around 2012, GWPilot was specifically aimed at grid resources and enabled customization of scheduling at the application level, independent from the resource placeholder implementation.

Pilot-Job systems have also been used to support science workflows. For example, Corral [102] was developed as a frontend to Glidein and to optimize the placement of glideins (i.e., resource placeholders) for the Pegasus workflow system [103]. Corral was later extended to also serve as one of the frontends of GlideinWMS. BOSCO [104], also a workflow management system, was developed to offer a unified job submission interface to diverse middleware, including the Glidein and GlideinWMS Pilot-Job systems. The Coaster [12, 105] and Falkon [18] Pilot-Job systems were both tailored to support the execution of workflows specified in the Swift language [106].

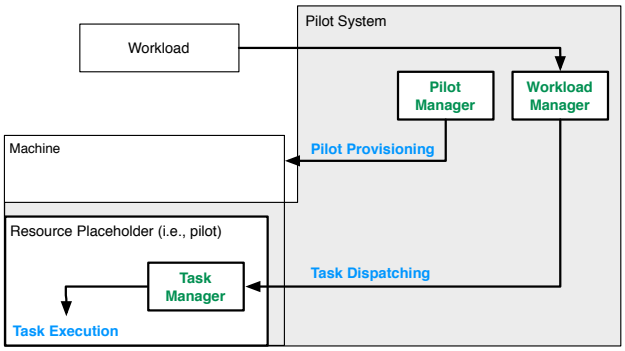

Figure 2: Diagrammatic representation of the logical components and functionalities of Pilot systems. The logical components are highlighted in green, and the functionalities in blue.

3. THE PILOT ABSTRACTION

The overview presented in §2 shows a degree of heterogeneity among Pilot-Job systems. These systems are implemented to support specific use cases by executing certain types of workload on machines with particular middleware. Implementation details hide the commonalities and differences among Pilot-Job systems. Consequently, in this section we describe the components, functionalities, and architecture pattern shared by Pilot-Job systems. Together, these elements comprise what we call the "pilot abstraction".

Pilot-Job systems are developed by independent projects and described with inconsistent terminologies. Often, the same term refers to multiple concepts or the same concept is named in different ways. We address this source of confusion by defining a terminology that can be used consistently across Pilot-Job systems, including the workloads they execute and the resources they use.

3.1 Logical Components and Functionalities

Pilot-Job systems employ three separate but coordinated logical components: a **Pilot Manager**, a **Workload Manager**, and a **Task Manager** (Figure 2). The Pilot Manager handles the provisioning of one or more resource placeholders (i.e., pilots) on single or multiple machines. The Workload Manager handles the dispatching of one or more workloads on the available resource placeholders. The Task Manager handles the execution of the tasks of each workload on the resource placeholders.

The implementation of these three logical components vary across Pilot-Job systems (see §4). For example, two or more logical components may be implemented by a single software element or additional functionalities may be integrated into the three management components.

The three logical components support the common functionalities of Pilot-Job systems: **Pilot Provisioning**, **Task Dispatching**, and **Task Execution** (Figure 2). Pilot-Job systems have to provision resource placeholders on the target machines, dispatch tasks on the available placeholders,

and use these placeholders to execute the tasks of the given workload. More functionalities may be needed to implement a production-grade Pilot-Job system as, for example, authentication, authorization, accounting, data management, fault-tolerance, or load-balancing. However, these functionalities depend on the type of use cases, workloads, or resources and, as such, are not necessary to every Pilot-Job system.

As seen in §2, resource placeholders enable tasks to utilize resources without directly depending on the capabilities exposed by the target machines. Resource placeholders are scheduled onto target machines by means of dedicated capabilities, but once scheduled and then executed, these placeholders make their resources directly available for the execution of the tasks of a workload.

The provisioning of resource placeholders depends on the capabilities exposed by the middleware of the targeted machine and on the implementation of each Pilot-Job system. Provisioning a placeholder on middleware with queues, batch systems and schedulers, typically involves the placeholder being submitted as a job. For such middleware, a job is a type of logical container that includes configuration and execution parameters alongside information on the application to be executed on the machine's compute resources. Conversely, for machines without a job-based middleware, a resource placeholder might be executed by means of other types of logical container as, for example, a virtual machine [107, 108].

Once placeholders control a portion of a machine resources, tasks need to be dispatched to those placeholders for execution. Task dispatching is controlled by the Pilot-Job system, not by the targeted machine's middleware. This is a defining characteristic of Pilot-Job systems because it decouples the execution of a workload from the need to submit its tasks via the machine's scheduler. Execution patterns involving task and/or data dependences can thus be implemented independent of the constraints of the target machine's middleware. Ultimately, this is how Pilot-Job systems can improve workload execution compared to direct submission.

The three logical components of a Pilot-Job system – Workload Manager, Pilot Manager, and Task Manager – need to communicate and coordinate in order to execute the given workload. Any suitable communication and coordination pattern [109, 110] can be used and this pattern may be implemented by any suitable technology. In a distributed context, different network architectures and protocols may also be used to achieve effective communication and coordination.

As seen in §2, master-worker is a common coordination pattern among Pilot-Job systems. Workload and task Managers are implemented as separated modules, one acting as master and the other as worker. The master dispatches tasks while the workers execute them independent of each other. Alternative coordination patterns can be used where, for example, Workload and Task Managers are implemented as a single module sharing dispatching and execution responsibilities.

Data management can play an important role within a Pilot-Job system as most of workloads require reading input and writing output data. The mechanisms used to make input data available and to store and share output data depend on use cases, workloads, and resources. Accordingly, data capabilities other than reading and writing files like, for example, data replication, (concurrent) data transfers, non file-based data abstractions, or data placeholders should be considered special-purpose capabilities, not characteristic of every Pilot-Job system.

3.2 Terms and Definitions

In this subsection, we define a minimal set of terms related to the logical components and capabilities of Pilot-Job systems. The terms "pilot" and "job" need to be understood in the context of machines and middleware used by Pilot-Job systems. These machines offer compute, storage, and network resources and pilots allow for the utilization of those resources to execute the tasks of one or more workloads.

Task. A set of operations to be performed on a computing platform, alongside a description of the properties and dependences of those operations, and indications on how they should be executed and satisfied. Implementations of a task may include wrappers, scripts, or applications.

Workload. A set of tasks, possibly related by a set of arbitrarily complex relations. For example, relations may involve tasks, data, or runtime communication requirements.

The tasks of a workload can be homogeneous, heterogeneous, or one-of-a-kind. An established taxonomy for workload description is not available. We propose a taxonomy based upon the orthogonal properties of coupling, dependency, and similarity of tasks.

Workloads comprised of tasks that are independent and indistinguishable from each other are commonly referred to as a Bag-of-Tasks (BoT) [111, 112]. Ensembles are workloads where the collective outcome of the tasks is relevant (e.g., computing an average property) [113]. The tasks that comprise the workload in turn can have varying degrees and types of coupling; coupled tasks might have global (synchronous) or local (asynchronous) exchanges, and regular or irregular communication. We categorize such workloads as coupled ensembles independent of the specific details of the coupling between the tasks. A workflow represents a workload with arbitrarily complex relationships among the tasks, ranging from dependencies (e.g., sequential or data) to coupling between the tasks (e.g., frequency or volume of exchange) [52].

Resource. A description of a finite, typed, and physical entity utilized when executing the tasks of a workload. Compute cores, data storage space, or network bandwidth between a source and a destination are examples of resources commonly utilized when executing workloads.

Distributed Computing Resource (DCR). A system characterized by: a set of possibly heterogeneous resources, a middleware, and an administrative domain. A cluster is an example of a DCR: it offers sets of compute, data, and network resources; it deploys a middleware as, for example, the Torque batch system, the Globus grid middleware, or the OpenStack cloud platform; and enforces policies of an administrative domain like XSEDE, OSG, CERN, NERSC, or a University. So called supercomputers or workstations can be other examples of DCR, where the term "distributed" refers to (correlated) sets of independent types of resources.

Distributed Computing Infrastructure (DCI). A set of DCRs federated with a common administrative, project, or policy domain, also shared at the software level. The federation and thus the resulting DCI can be

dynamic, for example, a DCR that is part of XSEDE can be federated with a DCR that is part of OSG without having to integrate entirely the two administrative domains.

Our definitions of resource and DCR might seem restrictive or inconsistent with how the term "resource" is sometimes used in the field of distributed computing. This is because the terms "DCR" and "resource" as defined here refer to the types of machine and to the types of computing resource they expose to the user. In its common use, the term "resource" conflates these two elements because it is used to indicate specific machines like, for example, `Stampede`, but also a specific computing resource as, for example, compute cores.

The term "DCR" also offers a more precise definition of the generic term "machine". DCR indicates a type of machine in terms of its resources, middleware, and administrative domain. These three elements are required to characterize Pilot-Job systems as they determine the type of resources that can be held by a pilot, the pilot properties and capabilities, and the administrative constraints on its instantiation.

The use of the term "distributed" in DCR makes explicit that the aggregation of diverse types of resources may happen at a physical or logical level, and at an arbitrary scale. This is relevant because the set of resources of a DCR can belong to a physical or virtual machine as much as to a set of these entities [114–116], either co-located on a single site or distributed across multiple sites. Both a physical cluster of compute nodes and a logical cluster of virtual machines are DCRs as they have a set of resources, a middleware, and an administrative domain.

The term "DCI", commonly used to indicate a distributed computing infrastructure, is consistent with both "resource" and "DCR" as defined here. Diverse types of resource are collected into one or more DCR, and aggregates of DCRs that share some common administrative aspects or policy form a DCI.

As seen in §2, most of the DCRs used by Pilot-Job systems utilize "queues", "batch systems", and "schedulers". In these DCRs, jobs are scheduled and then executed by a batch system.

Job. A type of container used to acquire resources on a DCR.

When considering Pilot-Job systems, jobs and tasks are functionally analogous but qualitatively different. Functionally, both jobs and tasks are containers, i.e. metadata wrappers around one or more executables often called "application" or "script". Qualitatively, tasks are the functional units of a workload, while jobs are what is scheduled on a DCR. Given their functional equivalence, the two terms can be adopted interchangeably when considered outside the context of Pilot-Job systems.

As described in §3.1, a resource placeholder needs to be submitted to a DCR in order to acquire resources for the Pilot-Job. The placeholder needs to be wrapped in a container, e.g., a job, and that container needs to be supported by the middleware of the target DCR. For this reason, the capabilities exposed by the middleware of the target DCR determine the submission process of resource placeholders and its specifics.

Pilot. A container (e.g., a "job") that functions as a resource placeholder on a given infrastructure and is capable of executing tasks of a workload on that resource.

A pilot is a resource placeholder that holds portion of a DCR's resources. A Pilot-Job system is software capable of creating pilots so as to gain exclusive control over a set of resources on one or more DCRs, and then to execute the tasks of one or more workloads on those pilots.

The term "pilot" as defined here is named differently across Pilot-Job systems. In addition to the term "placeholder", pilots have also been named "job agent", "job proxy", "coaster", and "glidein" [11, 12, 20, 117]. These terms are used as synonyms, often without distinguishing between the type of container and the type of executable that compose a pilot.

Until now, the term "Pilot-Job system" has been used to indicate those systems capable of executing workloads on pilots. For the remainder of this paper, the term "Pilot system" will be used instead, as the term "job" in "Pilot-Job" identifies just the way in which a pilot is provisioned on a DCR exposing specific middleware. The use of the term "Pilot-Job system" should be regarded as a historical artifact, indicating the use of middleware in which the term "job" was, and still is, meaningful.

We have now defined resources, DCRs, and pilots. We have established that a pilot is a placeholder for a set of DCR's resources. When combined, the resources of multiple pilots form a resource overlay. The pilots of a resource overlay can potentially be distributed over distinct DCRs.

Resource Overlay. The aggregated set of resources of multiple pilots possibly instantiated on multiple DCRs.

As seen in §2.1, three more terms associated with Pilot systems need to be explicitly defined: "early binding", "late binding", and "multi-level scheduling".

The terms "binding" and "scheduling" are often used interchangeably but here we use "binding" to indicate the association of a task to a pilot and "scheduling" to indicate the enactment of that association. Binding and scheduling may happen at distinct points in time and this helps to expose the difference between early and late binding, and multi-level scheduling.

The type of binding of tasks to pilots depends on the state of the pilot. A pilot is inactive until it is executed on a DCR, is active thereafter, until it completes or fails. Early binding indicates the binding of a task to an inactive pilot; late binding the binding of a task to an active one.

Early binding is useful because by knowing in advance the properties of the tasks that are bound to a pilot, specific deployment decisions can be made for that pilot. For example, a pilot can be scheduled onto a specific DCR, because of the capabilities of the DCR or because the data required by the tasks are already available on that DCR. Late binding is instead critical to assure high throughput by enabling sustained task execution without additional queuing time or pilot instantiation time.

Once tasks have been bound to pilots, Pilot systems are said to implement multi-level scheduling [5, 16, 54] because they include scheduling onto the DCR as well as scheduling onto the pilots. Unfortunately, the term "level" in multi-level is left unspecified making unclear what is scheduled and when. Assuming the term "entity" indicates what is scheduled, and the term "stage" the point in time at which the scheduling happens, "multi-entity" and "multi-stage" are better terms to

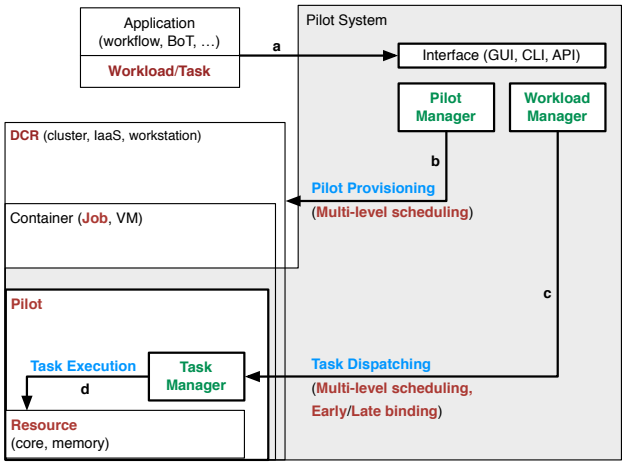

Figure 3: Diagrammatic representation of the logical components, functionalities, and core terminology of a Pilot system. The core terminology is highlighted in red, the logical components of a Pilot system in green, and the functionalities in blue. The components of a Pilot system are represented by boxes with a thicker border.

describe the scheduling properties of Pilot systems. "Multi-entity" indicates that (at least) two entities are scheduled and "multi-stage" that such scheduling happens at separate moments in time. Pilot systems schedule pilots on DCR and tasks on pilots at different point in time.

Early binding. Binding one or more tasks to an inactive pilot.

Late binding. Binding one or more tasks to an active pilot.

Multi-entity and Multi-stage scheduling. Scheduling pilots onto resources, and scheduling tasks onto (active or inactive) pilots.

Figure 3 offers a diagrammatic overview of the logical components of Pilot systems (green) alongside their functionalities (blue) and the defined terminology (red). The figure is composed of three main blocks: the one on the top-left corner represents the workload originator. The one starting at the top-right and shaded in gray represents the Pilot system, while the four boxes one inside the other on the left side of the figure represent a DCR. Of the four boxes, the outmost denotes the DCR boundaries, e.g., a cluster. The second box the container used to schedule a pilot on the DCR, e.g., a job or a virtual machine. The third box represents the pilot once it has been instantiated on the DCR, and the fourth box represents the resources held by the pilot. The boxes representing the components of a Pilot system have been highlighted with a thicker border.

Figure 3 shows the separation between the DCR and the Pilot system, and how the resources on which tasks are executed are contained in the DCR within different logical and physical components. Appreciating the characteristics and functionalities of a Pilot system depends upon understanding the levels at which each of its component exposes capabilities. An application submits one or more workloads composed of tasks to the Pilot system via an interface (tag a). The Pilot Manager is responsible for pilot provisioning (tag b), the Workload Manager to dispatch tasks to the Task Manager (tag c), the Task Manager to execute those tasks once the pilot has become available (tag d).

Note how in Figure 3 scheduling happens at the DCR (tag b), for example, by means of a cluster scheduler, and then at the pilot (tag c). This illustrates what here has been called "multi-entity" and "multi-stage" scheduling, replacing the more common but less precise term multi-level scheduling. The separation between scheduling at the pilot and scheduling at the Workload Manager highlights the four entities involved in the two-stage scheduling: jobs on DCR middleware, and tasks on pilots. This helps to appreciate the critical distinction between the container of a pilot and the pilot itself. A container is used by the Pilot Manager to provision the pilot. Once the pilot has been provisioned, it is the pilot and not the container that is responsible of both holding a set of resources and offering the functionalities of the Task Manager.

Figure 3 should not be confused with an architectural diagram. No indications are given about the interfaces that should be used, how the logical component should be mapped into software modules, or what type of communication and coordination protocols should be adopted among such components. This is why no distinction is made diagrammatically between, for example, early and late binding.

Figure 3 is instead an architectural pattern [118] for systems that execute workloads on multiple DCRs via pilot-based multi-entity, many-stage scheduling of tasks. This pattern can be realized into an architectural description and then implemented into a specific Pilot system. Several architectural models, frameworks, languages, supporting platforms, and standards are available to produce architectural descriptions [119, 120]. Common examples are 4+1 architectural view [121], Open Distributed Processing (ODP) [122], Zachman [123], The Open Group Architecture Framework (TOGAF) [124], and the Attribute-Driven Design (ADD) [125].

4. PILOT SYSTEMS

In this section we examine multiple implementations of Pilot systems. Initially, we derive core and auxiliary properties of Pilot system implementations from the components and functionalities described in §3.1. Subsequently, we describe a selection of Pilot system implementations showing how the architecture of each system maps to the architectural pattern presented in §3.2. Finally, we offer insight about the commonalities and differences among the described Pilot system implementations discussing also their most relevant auxiliary properties.

4.1 Core properties

Core properties are specific to Pilot systems and necessary for their implementation. These properties characterize Pilot systems because they relate to pilots and how they are used to execute tasks. Without core properties Pilot Managers, Workload Managers, and Task Managers would not be capable to provide pilots, and to dispatch and execute tasks. We list the core properties of Pilot systems in Table 1.

The first three core properties – Pilot Scheduling, Pilot Bootstrapping, and Pilot Resources – relate to the procedures used to provision pilots and to the resources they hold. Pilots can be deployed by a Pilot Manager using a suitable wrapper

that can be scheduled on the targeted DCR middleware. Pilots become available only with a correct bootstrapping procedure, and they can be used for task execution only if they acquire at least one type of resource, e.g., compute cores or data storage.

The Workload Binding and Workload Scheduling core properties relate to how Pilot systems bind tasks to pilots, and then how these tasks are scheduled once pilots become available. A Workload Manager can early or late bind tasks to pilots depending on the DCR's resources and workload's requirements. Scheduling decisions may depend on the number and capabilities of the available pilots or on the status of workload execution. Workload Binding and Workload Scheduling enable Pilot systems to control the coupling between tasks requirements and pilot capabilities.

The Workload Environment core property relates to the features and configuration of the environment provided by the pilot in which tasks are executed on the DCR. A Task Manager requires information about the environment to successfully manage the execution of tasks. For example, the Task Manager may have to make available supporting software or choose suitable parameters for the task executable. The following describes each core property. Note that these properties refer to Pilot systems and not to individual pilots instantiated on a DCR.

- **Pilot Scheduling**. Modalities for scheduling pilots on DCRs. Pilot scheduling may be: fully automated (i.e., implicit) or directly controlled by applications or users (i.e., explicit); performed on a single DCR (i.e., local) or coordinated across multiple DCRs (i.e., global); tailored to the execution of the workload (i.e., adaptive) or predefined on the basis of policies and heuristics (i.e, static).

- **Pilot Bootstrapping**. Modalities for pilot bootstrapping on DCRs. Pilots can be bootstrapped from code downloaded at every instantiation or from code that is bundled by the DCR. The design of pilot bootstrapping depends on the DCR environment and on whether single or multiple types of DCRs are targeted. For example, a design based on connectors can be used with multiple DCRs to get information about container type (e.g., job, virtual machine), scheduler type (e.g., PBS, HTCondor, Globus), amount of cores, walltime, or available filesystems.

- **Pilot Resources**. Types and characteristics of the resources exposed by a Pilot system. Resource types are, for example, compute, data, or networking while some of the their typical characteristics are: size (e.g., number of cores or storage capacity), lifespan, intercommunication (e.g., low-latency or inter-domain), computing platforms (e.g., x86 or GPU), file systems (e.g., local or distributed). The resource held by a pilot varies depending on the system architecture of the DCR in which the pilot is instantiated. For example, a pilot may hold multiple compute nodes, single nodes, or portion of the cores of each node. The same applies to file systems and their partitions or to physical and software-defined networks.

- **Workload Binding**. Time of workload assignment to pilots. Executing a workload requires its tasks to be bound to one or more pilots before or after they are instantiated on a DCR. As seen in §3, Pilot systems may allow for two modalities of binding between tasks and pilots: early binding and late binding. Pilot system implementations differ in whether and how they support these two types of binding.

- **Workload Scheduling**. Enactment of a binding. Pilot systems can support (prioritized) application-level or multi-stage scheduling decisions. Coupled tasks may have to be scheduled on a single pilot, loosely coupled or uncoupled tasks to multiple pilots; tasks may be scheduled to a pilot and then to a specific pool of resources on a single compute node; or task scheduling may be prioritized depending on task size and duration.

- **Workload Environment**. Type, dependences, and characteristics of the environment in which workload's tasks are executed. Once scheduled to a pilot, a task needs an environment that satisfies its execution requirements. The execution environment depends on the type of task (e.g., single or multi-threaded, MPI), task code dependences (e.g., compilers, libraries, interpreters, or modules), and task communication, coordination and data requirements (e.g., interprocess, inter-node communication, data staging, sharing, and replication).

4.2 Auxiliary properties

Auxiliary properties are not specific to Pilot systems and may be optional for their implementation. Pilot systems share auxiliary properties with other types of system and Pilot system implementations may have different subsets of these properties. For example, authentication and authorization are properties shared by many systems and Pilot systems may have to implement them only for some DCRs. Analogously, communication and coordination is not a core property of Pilot systems because, at some level, all software systems require communication and coordination.

We list a representative subset of auxiliary properties for Pilot systems in Table 2. The following describes these auxiliary properties and, also in this case, these properties refer to Pilot systems and not to individual pilots instantiated on a DCR.

- **Architecture**. Pilot systems may be implemented by means of different architectures, e.g., service-oriented, client-server, or peer-to-peer. Architectural choices may depend on multiple factors, including application use cases, deployment strategies, or interoperability requirements.

- **Communication and Coordination**. As discussed in §3.1, Pilot system implementations are not defined by any specific communication and coordination protocol or pattern. Communication and coordination among the Pilot system components are determined by its design, the chosen architecture, and the deployment scenarios.

- **Workload Semantics**. Pilot-Job systems may support workloads with different compute and data requirements, and inter-task dependences. Pilot systems may assume that only workloads with a specific semantics are given or may allow the user to specify, for example, BoT, ensemble, or workflow.

Property	Description	Component	Functionality
Pilot Scheduling	Modalities for pilot scheduling on DCRs	Pilot Manager	Pilot Provisioning
Pilot Bootstrapping	Modalities for pilot bootstrapping on DCRs		
Pilot Resources	Types and characteristics of pilot resources		
Workload Binding	Modalities and policies for binding tasks to pilots	Workload Manager	Task Dispatching
Workload Scheduling	Modalities and policies for scheduling tasks to pilots		
Workload Environment	Type and features of the task execution environment	Task Manager	Task Execution

Table 1: Mapping of the core properties of Pilot system implementations onto the components and functionalities described in §3.1. Core properties are specific to Pilot systems and necessary for their implementation.

Property	Description
Architecture	Structures and components of the Pilot system
Coordination and Communication	Interaction protocols and patterns among the components of the system
Interface	Interaction mechanisms both among components and exposed to the user
Interoperability	Qualitative and functional features shared among Pilots systems
Multitenancy	Simultaneous use of the Pilot system components by multiple users
Resource Overlay	The aggregation of resources from multiple pilots into overlays
Robustness	Resilience and reliability of pilot and workload executions
Security	Authentication, authorization, and accounting framework
Files and Data	Mechanisms for data staging and management
Performance	Measure of the scalability, throughput, latency, or memory usage
Development Model	Practices and policies for code production and management
DCR Interaction	Modalities and protocols for pilot system/DCR interaction coordination

Table 2: Sample of Auxiliary Properties and their descriptions. Auxiliary properties are not specific to Pilot systems and may be optional for their implementation.

- **Interface**. Pilot systems may implement several private and public interfaces: among the components of the Pilot system; among the Pilot system, the applications, and the DCRs; or between the Pilot system and the users via one or more application programming interfaces.

- **Interoperability**. Pilot system may implement at least two types of interoperability: among Pilot system implementations, and among DCRs with heterogeneous middleware. For example, two Pilot systems may execute tasks on each others' pilots, or a Pilot system may be able to provide pilots on LSF, Slurm, Torque, or OpenStack middleware.

- **Multitenancy**. Pilot systems may offer multitenancy at both system and local level. When offered at system level, multiple users can utilize the same instance of a Pilot system; when available at local level, multiple users can share the same pilot. Executing multiple pilots on the same DCR indicates the multitenancy of the DCR, not of the Pilot system.

- **Robustness**. Indicates the features of a Pilot system that contribute to its resilience and reliability. Usually, fault-tolerance, high-availability, and state persistence are indicators of the maturity of the Pilot system implementation and its use cases support.

- **Security**. The deployment and usability of Pilot systems are influenced by security protocols and policies. Authentication and authorization can be based on diverse protocols and vary across Pilot systems.

- **Data Management**. As discussed in §3.1, only basic data reading/writing functionalities are mandated by a Pilot system. Nonetheless, most real-life use cases require more advanced data management functionalities that can be implemented within the Pilot system or delegated to third-party tools.

- **Performance and scalability**. Pilot systems can be optimized for one or more performance metrics, depending on the target use cases. For example, Pilot systems vary in terms of overheads they add to the execution of a given workload, size and duration of the workloads a user can expect to be supported, and type and number of supported DCRs and DCIs.

- **Development Model**. The model used to develop Pilot systems may have an impact on the life span of the Pilot system, its maintainability and, possibly its evolution path. This is especially relevant when considering whether the development is supported by an open community or by a single research project.

4.3 Implementations

We analyze seven Pilot systems based on their availability, design, intended use, and uptake. We describe systems that: (i) implement diverse design; (ii) target specific or general-purpose use cases and DCR; and (iii) are currently available, actively maintained, and used by scientific communities. Space constraints prevented consideration of additional Pilot systems, as well as necessitated limiting the analysis to the core properties of Pilot systems.

We compare Pilot systems using the architectural pattern and common terminology defined in §3. Table 3 shows how

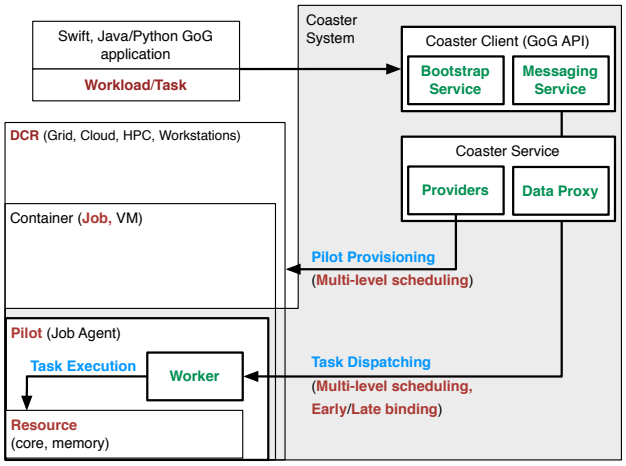

Figure 4: Diagrammatic representation of the Coaster System components, functionalities, and core terminology mapped on Figure 3.

the components of the architectural pattern are named differently across implementations. Table 4 offers instead a summary of how the core properties are implemented for each Pilot system we compared.[3]

4.3.1 *Coaster System*

The Coaster System (also referred to in literature as Coasters) was developed by the Distributed Systems Laboratory at the University of Chicago [129] and it is currently maintained by the Swift project [130]. Initially developed within the CoG project [131] and maintained in a separate, standalone repository, today the Coaster System provides pilot functionalities to Swift by means of an abstract task interface [132, 133].

The Coaster System is composed of three main components [12]: a Coaster Client, a Coaster Service, and a set of Workers. The Coaster Client implements both a Bootstrap and a Messaging Service while the Coaster Service implements a data proxy service and a set of job providers for diverse DCRs middleware. Workers are executed on the DCR compute nodes to bind compute resources and execute the tasks submitted by the users to the Coaster System.

Figure 4 illustrates how the Coaster System components map to the components and functionalities of a Pilot system as described in in §3: the Coaster Client is a Workload Manager, the Coaster Service a Pilot Manager, and each Worker a Task Manager. The Coaster Service implements the Pilot Provisioning functionality by submitting adequate numbers of Workers on suitable DCRs. The Coaster Client implements Task Dispatching while the Workers implement Task Execution.

The execution model of the Coaster System can be summarized in seven steps [105]: 1. a set of tasks is submitted by a user via the Coaster Client API; 2. when not already active, the Bootstrap Service and the Message Service are started within the Coaster Client; 3. when not already active, a Coaster Service is instantiated for the DCR(s) indicated in the task descriptions; 4. the Coaster Service gets the task descriptions and analyzes their requirements; 5. the Coaster Service submits one or more Workers to the target DCR taking also into account whether any other Worker is already active; 6. when a Worker becomes active it pulls a task and, if any, its data dependences from the Coaster Client via the Coaster Service; 7. the task is executed.

Each Worker holds compute resources in the form of compute cores. Data can be staged from a shared file-system, directly from the client to the Worker, or via the Coaster Service acting as a proxy. Data are not a type of resource held by the pilots and pilots are not used to expose data to the user. Networking capabilities are assumed to be available among the components of the Coaster System, but a dedicated communication protocol is implemented and also used for data staging as required.

The Coaster Service automates the deployment of pilots (i.e., Workers) by taking into account several parameters: total number of jobs that the DCR batch system accepts; number of cores for each DCR compute node; DCR policy for compute nodes allocation; walltime of the pilots compared to the total walltime of the tasks submitted by the users. These parameters are evaluated by a custom pilot deployment algorithm that performs a walltime overallocation estimated against user-defined parameters, and chooses the number and sizing of pilots on the base of the target DCR capabilities.

The Coaster System serves as a Pilot backend for the Swift System and, together, they can execute workflows composed of loosely coupled tasks with data dependences. Natively, the Coaster Client implements a Java CoG Job Submission Provider [131, 133, 134] for which Java API are available to submit tasks and to develop distributed applications. While tasks are assumed to be single-core by default, multi-core tasks can be executed by configuring the Coaster System to submit Workers holding multiple cores [135]. It should also be possible to execute MPI tasks by having Workers to span multiple compute nodes of a DCR.

The Coaster Service uses providers from the Java CoG Kit Abstraction Library to submit Workers to DCR with grid, HPC, and cloud middleware. The late binding of tasks to pilots is implemented by Workers pulling tasks to be executed as soon as free resources are available. It should be noted that tasks are bound to the pilots instantiated on a specific DCR specified as part of the task description. Experiments have been made with late binding to pilots instantiated on arbitrary DCRs but no documentation is currently available about the results obtained.[4]

4.3.2 *DIANE*

DIANE (DIstributed ANalysis Environment) [13] has been developed at CERN [136] to support the execution of workloads on the DCRs federated to be part of European Grid Infrastructure (EGI) [137] and worldwide LHC Computing Grid (WLCG). DIANE has also been used in the Life Sciences [138–140] and in few other scientific domains [141, 142].

DIANE is an application task coordination framework that executes distributed applications using the master-worker pattern [13]. DIANE consists of four logical components: a TaskScheduler, an ApplicationManager, a SubmitterScript, and a set of ApplicationWorkers [143]. The first two components – TaskScheduler and the ApplicationManager – are implemented as a RunMaster service, while the Application-

[3]Pilot systems are ordered alphabetically in the table and in the text.

[4]Based on private communication with the Coaster System development team.

Pilot System	Pilot Manager	Workload Manager	Task Manager	Pilot
Coaster System	Coaster Service	Coaster Client	Worker	Job Agent
DIANE	Submitter script	RunMaster	ApplicationWorker	WorkerAgent
DIRAC	WMS (Directors)	WMS (Match Maker)	Job Wrapper	Job Agent
GlideinWMS	Glidein Factory	Schedd	Startd	Glidein
MyCluster	Cluster Builder Agent	Virtual Login Session	Task Manager	Job Proxy
PanDA	Grid Scheduler	PANDA Server	RunJob	Pilot
RADICAL-Pilot	Pilot Manager	CU Manager	Agent	Pilot

Table 3: Mapping of the names given to the components of the pilot architectural pattern defined in §3.2, Figure 3 and the names given to the components of pilot system implementations.

Pilot System	Pilot Resources	Pilot Deployment	Workload Semantics	Workload Binding	Workload Execution
Coaster System	Compute	Implicit	WF (Swift [126])	Late	Serial, MPI
DIANE	Compute	Explicit	WF (MOTOUR [126])	Late	Serial
DIRAC	Compute	Implicit	WF (TMS)	Late	Serial, MPI
GlideinWMS	Compute	Implicit	WF (Pegaus, DAGMan [127])	Late	Serial, MPI
MyCluster	Compute	Implicit	job descriptions	Late	Serial, MPI
PanDA	Compute	Implicit	BoT	Late	Serial, MPI
RADICAL-Pilot	Compute, data	Explicit	ENS (EnsembleMD Toolkit [128])	Early, Late	Serial, MPI

Table 4: Overview of Pilot systems and a summary of the values of their core properties. Based on the tooling currently available for each Pilot system, the types of workload supported as defined in §3.2 are: BoT = Bag of Tasks; ENS = Ensembles; WF = workflows.

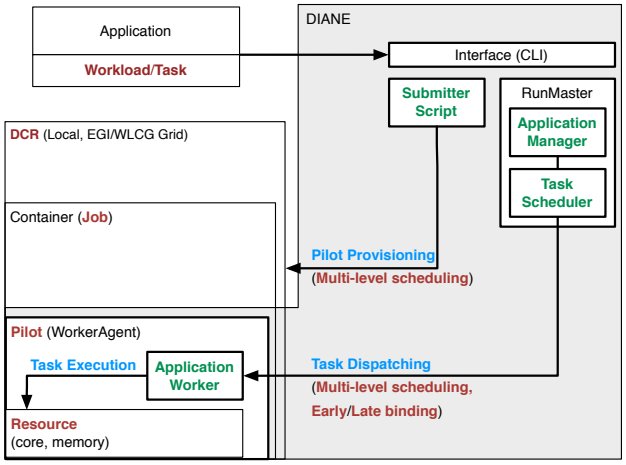

Figure 5: Diagrammatic representation of DIANE components, functionalities, and core terminology mapped on Figure 3.

Workers as a WorkerAgent service. Submitter Scripts deploy ApplicationWorkers on DCRs.

Figure 5 shows how DIANE implements the components and functionalities of a pilot system as described in §3: the RunMaster service is a Workload Manager, the SubmitterScript is a Pilot Manager, and the ApplicationWorker of each WorkerAgent service is a Task Manager. Accordingly, the Pilot provisioning functionality is implemented by the SubmitterScript, Task Dispatching by the RunMaster, and Task Execution by the WorkerAgent. In DIANE, Pilots are called "WorkerAgents".

The execution model of DIANE can be summarized in four steps [144]: 1. the user submits one or more jobs to DCR by means of SubmitScript(s) to bootstrap one or more WorkerAgent; 2. When ready, the WorkAgent(s) reports back to the ApplicationManager; 3. tasks are scheduled by the TaskScheduler on the available WorkerAgent(s); 4. after execution, WorkerAgents send the output of the computation back to the ApplicationManager.

The pilots used by DIANE (i.e., WorkerAgents) hold compute resources on the target DCRs. WorkerAgents are executed by the DCR middleware as jobs with mostly one core but possibly more. DIANE also offers a data service with a dedicated API and CLI that allows for staging files in and out of WorkerAgents. This service represents an abstraction of the data resources and capabilities offered by the DCR, and it is designed to handle data only in the form of files stored into a file system. Network resources are assumed to be available among DIANE components.

DIANE requires a user to develop pilot deployment mechanisms tailored to specific resources. The RunMaster service assumes the availability of pilots to schedule the tasks of the workload. Deployment mechanisms can range from direct manual execution of jobs on remote resources, to deployment scripts, or full-fledged factory systems to support the sustained provisioning of pilots over extended periods of time.

A tool called "GANGA" [145, 146] is available to support the development of SubmitterScripts. GANGA facilitates the submission of pilots to diverse DCRs by means of a uniform interface and abstraction. GANGA offers interfaces for job submission to DCRs with Globus, HTCondor, UNICORE, or gLite middleware.

DIANE has been designed to execute workloads that can be partitioned into ensembles of parametric tasks on multiple pilots. Each task can consist of an executable invocation but also of a set of instructions, OpenMP threads, or MPI

processes [144]. Relations among tasks and group of tasks can be specified before or during runtime enabling DIANE to execute articulated workflows. Plugins have been written to manage DAGs [147] and data-oriented workflows [148].

DIANE is primarily designed for HTC and Grid environments and to execute pilots with a single core. Nonetheless, the notion of "capacity" is exposed to the user to allow for the specification of pilots with multiple cores. Although the workload binding is controllable by the user-programmable TaskScheduler, the general architecture is consistent with a pull model. The pull model naturally implements the late binding paradigm where every ApplicationAgent of each available pilot pulls a new task.

4.3.3 DIRAC

DIRAC (Distributed Infrastructure with Remote Agent Control) [149] is a software product developed by the CERN LHCb project. DIRAC implements a Workload Management System (WMS) to manage the processing of detector data, Monte Carlo simulations, and end-user analyses. DIRAC primarily serves as the LHCb workload management interface to WLCG executing workloads on DCRs deploying Grid, Cloud, and HPC middleware.

DIRAC has four main logical components: a set of TaskQueues, a set of TaskQueueDirectors, a set of JobWrappers, and a MatchMaker. TaskQueues, TaskQueueDirectors, and the MatchMaker are implemented within a monolithic WMS. Each TaskQueue collects tasks submitted by users, multiple TaskQeue being created depending on the requirements and ownership of the tasks. JobWrappers are executed on the DCR to bind compute resources and execute tasks submitted by the users. Each TaskQueueDirector submits JobWrappers to target DCRs. The MatchMaker matches requests from JobWrappers to suitable tasks into TaskQueues.

DIRAC was the first pilot-based WMS designed to serve a LHC main experiment [14]. Figure 6 shows how the DIRAC WMS implements a Workload, a Pilot, and a Task Manager as they have been described in §3. TaskQueues and the MatchMaker implement the Workload Manager and the related Task Dispatching functionality. Each TaskQueueDirector implements a Pilot Manager and its Pilot Provisioning functionality, while each JobWrapper implements a Task Manager and Pilot Execution.

The DIRAC execution model can be summarized in five steps: 1. a user submits one or more tasks by means of a CLI, Web portal, or API to the WMS Job Manager; 2. submitted tasks are validated and added to a new or an existing TaskQueue, depending on the task properties; 3. one or more TaskQueues are evaluated by a TaskQueueDirector and a suitable number of JobWrappers are submitted to available DCRs; 4. JobWrappers, once instantiated on the DCRs, pull the MatchMaker asking for tasks to be executed; 5. tasks are executed by the JobWrappers under the supervision of each JobWrapper's Watchdog.

JobWrappers, the DIRAC pilots, hold compute resources in the form of single or multiple cores, spanning portions, whole, or multiple compute nodes. A dedicated subsystem is offered to manage data staging and replication but data capabilities are not exposed via pilots. Network resources are assumed to be available to allow pilots to communicate with the WMS.

Pilots are deployed by TaskQueueDirectors. Three main

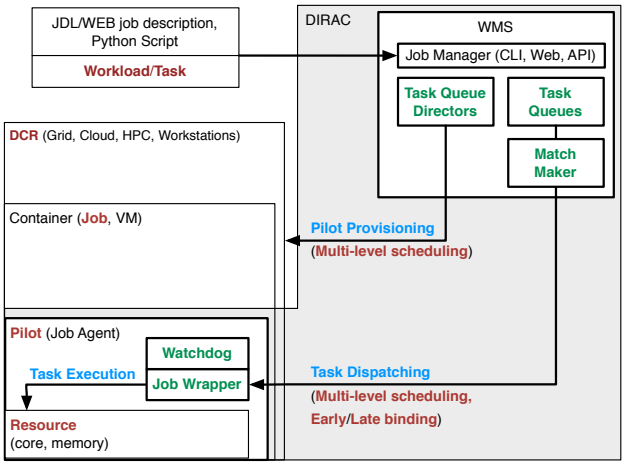

Figure 6: Diagrammatic representation of DIRAC components, functionalities, and core terminology mapped on Figure 3.

operations are iterated: 1. getting a list of TaskQueues; 2. calculating the number of pilots to submit depending on the user-specified priority of each task, and the number and properties of the available or scheduled pilots; and 3. submitting the calculated number of pilots.

Natively, DIRAC can execute tasks described by means of the Job Description Language (JDL) [150]. As such, single-core, multi-core, MPI, parametric, and collection tasks can be described and submitted. Users can specify a priority index for each submitted task and one or more specific DCR that should be targeted for execution. Tasks with complex data dependences can be described by means of a DIRAC system called "Transformation Management System" (TMS) [151]. In this way, user-specified, data-driven workflows can be automatically submitted and managed by the DIRAC WMS.

Similar to DIANE and the Coaster System, DIRAC features a task pull model that naturally implements late binding of tasks to pilots. Each JobWrapper pulls a new task once it is available and has free resources. No early binding of tasks on pilots is offered.

4.3.4 HTCondor Glidein and GlideinWMS

The HTCondor Glidein system [152] was developed by the Center for High Throughput Computing at the University of Wisconsin-Madison (UW-Madison) [153] as part of the HTCondor [154] software ecosystem. The HTCondor Glidein system implements pilots to aggregate DCRs with heterogeneous middleware into HTCondor resource pools.

The logical components of HTCondor relevant to the Glidein system are: a set of Schedd and Startd daemons, a Collector, and a Negotiator [10]. Schedd is a queuing system that holds workload tasks and Startd handles the DCR resources. The Collector holds references to all the active Schedd/Startd daemons, and the Negotiator matches tasks queued in a Schedd to resources handled by a Startd.

HTCondor Glidein has been complemented by GlideinWMS, a Glidein-based workload management system that automates deployment and management of Glideins on multiple types of DCR middleware. GlideinWMS builds upon the HTCondor Glidein system by adding the following logical

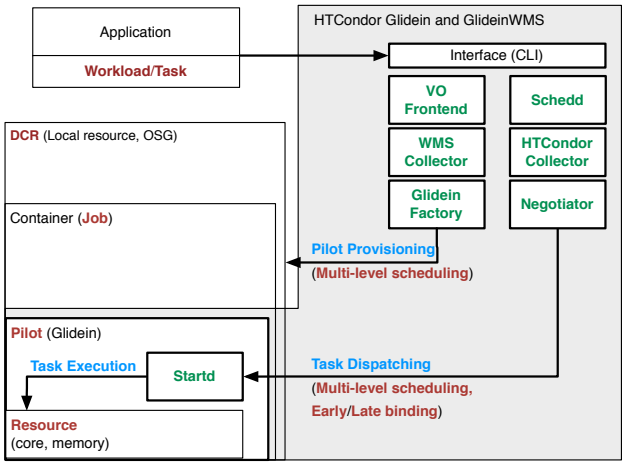

Figure 7: Diagrammatic representation of Glidein components, functionalities, and core terminology mapped on Figure 3.

components: a set of Glidein Factory daemons, a set of Frontend daemons for Virtual Organization (VO) [155, 156], and a Collector dedicated to the WMS [157]. Glidein Factories submit tasks to the DCRs middleware, each VO Frontend matches the tasks on one or more Schedd to the resource attributes advertised by a specific Glidein Factory, and the WMS Collector holds references to all the active Glidein Factories and VO Frontend daemons.

Figure 7 shows the mapping of the HTCondor Glidein Service and GlideinWMS elements to the components and functionalities of a Pilot system as described in §3. The set of VO Frontends and Glidein Factories alongside the WMS collector implement a Pilot Manager and its pilot provisioning functionality. The set of Schedd, the Collector, and the Negotiator implement a Workload Manager and its task dispatching functionality. The Startd daemon implements a Task Manager alongside its task execution functionality. A Glidein is a job submitted to a DCR middleware that, once instantiated, configures and executes a Startd daemon. Glidein is therefore a pilot.

The execution model of the HTCondor Glidein system can be summarized in nine steps: 1. the user submits a Glidein (i.e., a job) to a DCR batch scheduler; 2. once executed, this Glidein bootstraps a Startd daemon; 3. the Startd daemon advertises itself to the Collector; 4. the user submits the tasks of the workload to the Schedd daemon; 5. the Schedd advertises these tasks to the Collector; 6. the Negotiator matches the requirements of the tasks to the properties of one of the available Startd daemon (i.e., a Glidein); 7. the Negotiator communicates the match to the Schedd; 8. the Schedd submits the tasks to the Startd daemon indicated by the Negotiator; 9. the task is executed.

GlideinWMS extends the execution model of the HTCondor Glidein system by automating the provision of Glideins. The user does not have to submit Glidein directly but only tasks to Schedd. From there: 1. every Schedd advertises its tasks with the VO Frontend; 2. the VO Frontend matches the tasks' requirements to the resource properties advertised by the WMS Connector; 3. the VO Frontend places requests for Glideins instantiation to the WMS Collector; 4. the WMS Collector contacts the appropriate Glidein Factory to execute the requested Glideins; 5. the requested Glideins become active on the DCRs; and 6. the Glideins advertise their availability to the (HTCondor) Collector. From there on the execution model is the same as described for the HTCondor Glidein Service.

The resources managed by a single Glidein (i.e., pilot) are limited to compute resources. Glideins may bind one or more cores, depending on the target DCRs. For example, heterogeneous HTCondor pools with resources for desktops, workstations, small campus clusters, and some larger clusters will run mostly single core Glideins. More specialized pools that hold, for example, only DCRs with HTC, Grid, or Cloud middleware may instantiate Glideins with a larger number of cores. Both HTCondor Glidein and GlideinWMS provide abstractions for file staging but pilots are not used to hold data or network resources.

The process of pilot deployment is the main difference between HTCondor Glidein and GlideinWMS. While the HTCondor Glidein system requires users to submit the pilots to the DCRs, GlideinWMS automates and optimizes pilot provisioning. GlideinWMS attempts to maximize the throughput of task execution by continuously instantiating Glideins until the queues of the available Schedd are emptied. Once all the tasks have been executed, the remaining Glideins are terminated.

HTCondor Glidein and GlideWMS expose the interfaces of HTCondor to the application layer and no theoretical limitation is posed on the type and complexity of the workloads that can be executed. For example, DAGMan (Directed Acyclic Graph Manager) [158] has been designed to execute workflows by submitting tasks to Schedd, and a tool is available to design applications based on the master-worker coordination pattern.

HTCondor was originally designed for resource scavenging and opportunistic computing. Thus, in practice, independent and single (or few-core) tasks are more commonly executed than many-core tasks, as is the case for OSG, the largest HTCondor and GlideinWMS deployment. Nonetheless, in principle projects may use dedicated installation and resources to execute tasks with larger core requirements both for distributed and parallel applications, including MPI applications.

Both HTCondor Glidein and GlideWMS rely on one or more HTCondor Collectors to match task requirements and resource properties, represented as ClassAds [159]. This matching can be evaluated right before the scheduling of the task. In this way, late binding is achieved but early binding remains unsupported.

4.3.5 *MyCluster*

MyCluster [160, 161] is not maintained but is included in the comparison because it presents some distinctive features. Its user/Pilot system interface and task submission system based on the notion of virtual cluster highlight the flexibility of Pilot systems implementations. Moreover, MyCluyster was one of the first Pilot system to be aimed specifically at HPC DCRs.

MyCluster was originally developed at the Texas Advanced Computing Center (TACC) [162], sponsored by NSF to enable execution of workloads on TeraGrid [163], a set of DCRs deploying Grid middleware. MyCluster provides users with

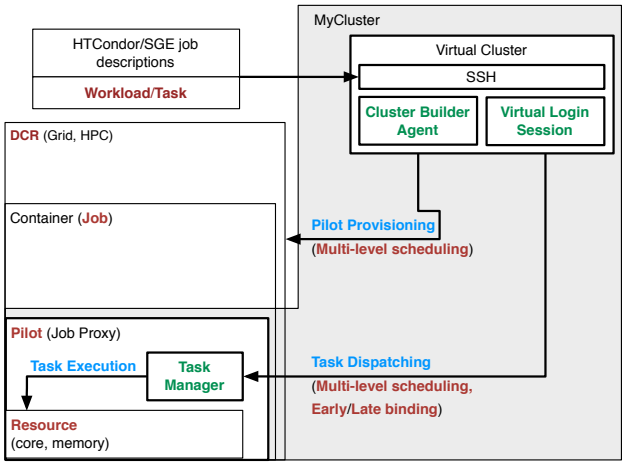

Figure 8: Diagrammatic representation of MyCluster components, functionalities, and core terminology mapped on Figure 3.

virtual clusters: aggregates of homogeneous resources dynamically acquired on multiple and diverse DCRs. Each virtual cluster exposes HTCondor, SGE [164], or OpenPBS [165] job-submission systems, depending on the user and use case requirements.

MyCluster is designed around three main components: a Cluster Builder Agent, a system where users create Virtual Login Sessions, and a set of Task Managers. The Cluster Builder Agent acquires the resources from diverse DCRs by means of multiple Task Managers, while the Virtual Login Session presents these resources as a virtual cluster to the user. A virtual login session can be dedicated to a single user, or customized and shared by all the users of a project. Upon login on the virtual cluster, a user is presented with a shell-like environment used to submit tasks for execution.

Figure 8 shows how the components of MyCluster map to the components and functionalities of a Pilot system as described in §3.1: The Cluster Builder Agent implements a Pilot Manager and a Virtual Login Session implements a Workload Manager. The Task Manager shares its name and functionality with the homonymous component defined in §3.1. The Cluster Builder Agent provides Task Managers by submitting Job Proxies to diverse DCRs, and a Virtual Login Session uses the Task Managers to submit and execute tasks. As such, Job Proxies are pilots.

The execution model of MyCluster can be summarized in five steps: 1. a user logs into a dedicated virtual cluster via, for example, ssh to access a dedicated Virtual Login Session; 2. the user writes a job wrapper script using the HTCondor, SGI, or OpenPBS job specification language; 3. the user submits the job to the job submission system on the virtual cluster; 4. the Cluster Builder Agent submits a suitable number of Job Proxies on one or more DCR; 5. when the Job Proxies become active, the user-submitted job is executed on the resources they hold.

Job Proxies hold compute resources in the form of compute cores. MyCluster does not offer any dedicated data subsystem and Job Proxies (i.e. pilots) are not used to expose data resources to the user. Users are assumed to stage the data required by the compute tasks directly, or by means of the data capabilities exposed by the job submission system of the virtual cluster. Networking is assumed to be available among the MyCluster components.

The Cluster Builder Agent submits Job Proxies to each DCR by using the GridShell framework [166]. GridShell wraps the Job Proxies description into the job description language supported by the target DCR. Thanks to GridShell, MyCluster can submit jobs to DCR with diverse middleware.

MyCluster exposes a virtual cluster with a predefined job submission system to the user. Pilots can have a user-defined amount of cores inter or cross-compute node. As such, every application built to utilize HTCondor, SGE, or OpenPBS can be executed transparently on MyCluster. This includes single and multi-core tasks, MPI tasks, and data-driven workflows.

The jobs specified by a user are bound to the DCR resources as soon as Job Proxies become active. The user does not have to specify on which Job Proxies or DCR each task has to be executed. In this way, MyCluster implements late binding.

4.3.6 PANDA

PanDA (Production and Distributed Analysis) [167] was developed to provide a workload management system (WMS) for ATLAS. ATLAS is a particle detector at the LHC that requires a WMS to handle large numbers of tasks for their data-driven processing workloads. In addition to the logistics of handling large-scale task execution, ATLAS also needs integrated monitoring for the analysis of system state, and a high degree of automation to reduce user and administrative intervention.

PanDA has been initially deployed as a HTC-oriented, multi-user WMS system consisting of 100 heterogeneous computing sites [168]. Recent improvements to PanDA have extended the range of deployment scenarios to HPC and cloud-based DCRs making PanDA a general-purpose Pilot system [169].

PanDA architecture consists of a Grid Scheduler and a PanDA Server [170,171]. The Grid Scheduler is implemented by a component called "AutoPilot" that submits jobs to diverse DCRs. The PanDA server is implemented by four main components: a Task Buffer, a Broker, a Job Dispatcher, and a Data Service. The Task Buffer collects all the submitted tasks into a global queue and the Broker prioritizes and binds those tasks to DCRs on the basis of multiple criteria. The Data Service stages the input file(s) of the tasks to the DCR to which the tasks have been bound using the data transfer technologies exposed by the DCR middleware (e.g., uberftp, gridftp, or lcg-cp). The Job Dispatcher delivers the tasks to the RunJobs run by each Pilot bound to a DCR.

Figure 9 shows how PANDA implements the components and functionalities of a Pilot system as described in §3: the Grid Scheduler is a Pilot Manager implementing Pilot Provisioning while the PanDA Server is a Workload Manager implementing Task Dispatching. The jobs submitted by the Grid Scheduler are called "Pilots" and act as pilots once instantiated on the DCR by running RunJob, i.e., the Task Manager. RunJob contacts the Job Dispatcher component to request for tasks to be executed.

The execution model of PANDA can be summarized in eight steps [172,173]: 1. the user submits tasks to the PanDA server; 2. the tasks are queued within the Task Buffer; 3. the tasks requirements are evaluated by the Broker and bound

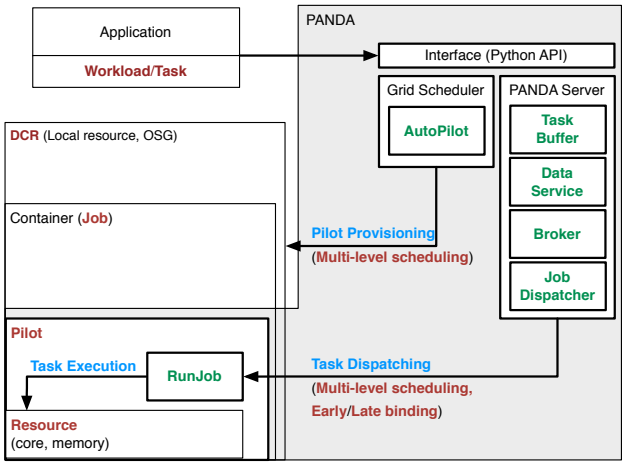

Figure 9: Diagrammatic representation of PANDA components, functionalities, and core terminology mapped on Figure 3.

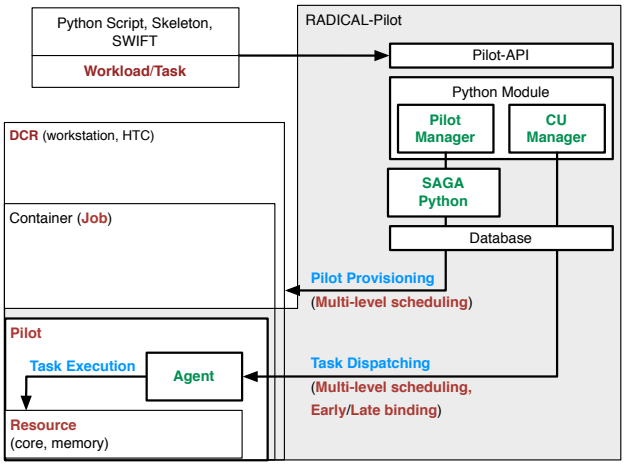

Figure 10: Diagrammatic representation of RADICAL Pilot components, functionalities, and core terminology mapped on Figure 3.

to a DCR; 4. the input files of the tasks are staged to the bound DCR by the Data Service; 5. the required pilot(s) are submitted as jobs to the target DCR; 6. the submitted pilot(s) becomes available and reports back to the Job Dispatcher; 7. tasks are dispatched to the available pilots for execution; 8. tasks are executed.

PanDA pilots expose mainly single cores, but extensions have been developed to instantiate pilots with multiple cores [174]. The Data Service of PanDA allows the integration and automation of data staging within the task execution process, but no pilots are offered for data [168]. Network resources are assumed to be available among PanDA components, but no network-specific abstraction is made available.

The AutoPilot component of PanDA's Grid Scheduler has been designed to use multiple methods to submit pilots to DCRs. The PanDA installations of the US ATLAS infrastructure uses the HTCondor-G [72] system to submit pilots to the US production sites. Other schedulers enable AutoPilot to submit to local and remote batch systems and to the GlideinWMS frontend. Submissions via the canonical tools offered by HTCondor have also been used to submit tasks to cloud resources.

PanDA was initially designed to serve specifically the ATLAS use case, executing mostly single-core tasks with input and output files. Since its initial design, the ATLAS analysis and simulation tools have started to investigate multi-core task execution with AthenaMP [174] and PanDA has been evolving towards a more general purpose workload manager [175–177]. Currently, PanDA offers experimental support for multi-core pilots and tasks with or without data dependences. PanDA is being generalized to support applications from a variety of science domains. [178].

PanDA offers late binding but not early binding capabilities. Workload jobs are assigned to activated and validated pilots via the PanDA server based on brokerage criteria like data locality and resource characteristics.

4.3.7 *RADICAL-Pilot*

The authors of this paper have been engaged in theoretical and practical aspects of Pilot systems. In addition to formulating the P* Model [179], the RADICAL group [180] is responsible for developing and maintaining the RADICAL-Pilot Pilot system [181]. RADICAL-Pilot is built upon the experience gained from developing BigJob, and integrating it with many applications [182–184] on different DCRs.

RADICAL-Pilot consists of five main logical components: a Pilot Manager, a Compute Unit (CU) Manager, a set of Agents, the SAGA-Python DCR interface, and a database. The Pilot Manager describes pilots and submits them via SAGA-Python to DCR(s), while the CU manager describes tasks (i.e. CU) and schedules them to one or more pilots. Agents are instantiated on DCRs and execute the CUs pushed by the CU manager. The database is used for the communication and coordination of the other four components.

RADICAL-Pilot closely resembles the description offered in §3 (see Figure 10). The Pilot Manager and SAGA-Python implement the logical component also called "Pilot Manager" in §3.1. The Workload Manager is implemented by the CU Manager. The Agent is deployed on the DCR to expose its resources and execute the tasks pushed by the CU Manager. As such, the Agent is a pilot.

RADICAL-pilot is implemented as two Python modules to support the development of distributed applications. The execution model of RADICAL-Pilot can be summarized in six steps: 1. the user describes tasks in Python as a set of CUs with or without data and DCR dependences; 2. the user also describes one or more pilots choosing the DCR(s) they should be submitted to; 3. upon execution of the user's application, the Pilot Manager submits each pilot that has been described to the indicated DCR utilizing the SAGA interface; 4. The CU Manager schedules each CU either to the pilot indicated in the CU or on the first pilot with free and available resources. Scheduling is done by storing the CU description into the database; 5. when required, the CU Manager also stages the CU's input file(s) to the target DCR; and 6. the Agent pulls its CU from the database and executes it.

The Agent component of RADICAL-Pilot offers abstractions for both compute and data resources. Every Agent can

expose between one and all the cores of the compute node where it is executed; it can also expose a data handle that abstracts away specific storage properties and capabilities. In this way, the CUs running on an Agent can benefit from unified interfaces to both core and data resources. Networking is assumed to be available between the RADICAL-Pilot components.

The Pilot Manager deploys the Agents of RADICAL-Pilot by means of the SAGA-python API [98]. SAGA provides access to diverse DCR middleware via a unified and coherent API, and thus RADICAL-Pilot can submit pilots to resources exposed by XSEDE and NERSC [185], by the OSG HTCondor pools, and many "leadership" class systems like those managed by OLCF [186] or NCSA [187].

The resulting separation of agent deployment from DCR architecture reduces the overheads of adding support for a new DCR [23]. This is illustrated by the relative ease with which RADICAL-Pilot is extended to support (i) a new type of DCR such as IaaS, and (ii) DCRs that have essentially similar architecture but different middleware, for example the Cray supercomputers operated in the US and Europe.

RADICAL-Pilot can execute tasks with varying coupling and communication requirements. Tasks can be completely independent, single or multi-threaded; they may be loosely coupled requiring input and output files dependencies, or they might require low-latency runtime communication. As such, RADICAL-Pilot supports MPI applications, workflows, and diverse execution patterns such as pipelines.

CU descriptions may or may not contain a reference to the pilot to which the user wants to bind the CU. When a reference is present, the scheduler of the CU Manager waits for a slot to be available on the indicated pilot. When a target pilot is not specified, the CU Manager binds and schedules the CU on the first pilot available. As such, RADICAL-Pilot supports both early and late binding, depending on the use case and the user specifications.

4.4 Comparison

The previous subsection shows how diverse Pilot system implementations conform to the architecture pattern we described in §3.2. This confirms the generality of the pattern at capturing the components and functionalities required to implement a Pilot system. The described Pilot systems also show implementation differences, especially concerning the following auxiliary properties: Architecture, Communication and Coordination, Interoperability, Interface, Security, and Performance and Scalability.

The Pilot systems described in §4.3 implement different architectures. DIANE, DIRAC, and, to some extent, both PANDA and the Coaster System are monolithic (Figures 5, 6, 9, and 4). Most of their functionalities are aggregated into a single component implemented "as a service" [188]. A dedicated hardware infrastructure is assumed for a production-grade deployment of DIRAC and PANDA. Consistent with a Globus-oriented design, the Coaster Service is instead assumed to be run on the DCR acting as a proxy for both the pilot and workload functionalities.

MyCluster and RADICAL-Pilot also are mostly monolithic (Figures 10 and 8) but not implemented as a service. MyCluster resembles the architecture of a HPC middleware while Radical-Pilot is implemented as two Python modules. MyCluster requires dedicated hardware analogously to the head-node of a traditional HPC cluster. RADICAL-Pilot users are instead free to decide where to deploy their applications, either locally on workstations or remotely on dedicated machines. In production-grade deployment, RADICAL-Pilot requires a dedicated database to support its communication and coordination protocols.

GlideinWMS requires integration within the HTCondor ecosystem and therefore also a service oriented architecture but it departs from a monolithic design. GlideinWMS implements a set of separate, mostly autonomous services (Figure 7) that can be deployed depending on the available resources and on the motivating use case.

Architecture frameworks and description languages [189, 190] can be used to further specify and refine the component architectures in Figures 4-10. For example, the 4+1 framework alongside a UML-based notation [191, 192] could be used to describe multiple "views" of each Pilot system architecture, offering more details and better documentation about the implementation of their components, the functionalities provided to the user, the behavior of the system, and its deployment scenarios.

The Pilot systems described in the previous subsection also display differences in their communication and coordination models. While all the Pilot systems assume preexisting networking functionalities, the Coaster System implements a dedicated communication protocol used both for coordination and data staging. The Coaster System and RADICAL-Pilot can both work as communication proxies among the Pilot system's components when the DCR compute nodes do not expose a public network interface. All the Pilot systems implement the master-worker coordination pattern, but the Task and the Workload Managers in DIRAC, PANDA, MyCluster, and the Coaster System can also coordinate to recover task failures and isolate under-performing or failing DCR compute nodes.

Figures 4-10 also shows different interfaces between Pilot systems and DCRs, and between Pilot systems and users or applications. Most of the described Pilot systems interoperate across diverse DCR middleware, including HPC, grid, and cloud batch systems. Implementations of this interoperability diverge, ranging from the dedicated SAGA API used by RADICAL-Pilot, to special-purpose connectors used by DIANE, DIRAC and PANDA, to the installation of specialized components on the DCR middleware used by Coaster System, Glidein, and MyCluster. These interfaces are functionally analogous; reducing their heterogeneity would limit effort duplication and promote interoperability across Pilot systems.

The interfaces exposed to give users access to pilot capabilities differ both in types and implementations. DIANE, DIRAC, GlideinWMS, MyCluster, and PANDA offer command line tools. These are often tailored to specific use cases, applications, and DCRs, requiring to be installed on the users' workstations or on dedicated machines. The Coaster System and RADICAL-Pilot expose an API, and the command line tools of DIANE, DIRAC, and PANDA are built on APIs that users may directly access to develop distributed applications.

Differences in the user interfaces stem from assumptions about distributed applications and their use cases. Interfaces based on command line tools assume applications that can be "submitted" to the Pilot system for execution. APIs assume instead applications that need to be coded by the user, depending on the specific requirements of the use case.

These assumptions justify multiple aspects of the design of Pilot systems, determining many characteristics of their implementations.

The described Pilot systems also implement different types of authentication and authorization (AA). The AA required by the user to submit tasks to their own pilots varies depending on the pilot's tenancy. With single tenancy, AA can be based on inherited privileges as the pilot can be accessed only by the user that submitted it. With multitenancy, the Pilot system has to evaluate whether a user requesting access to a pilot is part of the group of allowed users. This requires abstractions like virtual organizations and certificate authorities [193], implemented, for example, by GlideinWMS and the Coaster Systems.

The credential used for pilot deployment depends on the target DCR. The AA requirements of DCRs are a diverse and often inconsistent array of mechanisms and policies. Pilot systems are gregarious in the face of such a diversity as they need to present the credentials provided by the application layer (or directly by the user) to the DCR. As such, the AA requirements specific to Pilot systems are minimal but the implementation required to present suitable credentials may be complex, especially when considering Pilot systems offering interoperability among diverse DCRs.

Finally, the differences among Pilot system implementations underline the difficulties in defining and correlating performance metrics. The performance of each Pilot system can be evaluated under multiple metrics that are affected by the workload, the Pilot system behavior, and the DCR. For example, the commonly used metrics of system overhead and workload's time to completion depend on the design of the Pilot system; on the data, compute and network requirements of the workload executed; and on the capabilities of the target resources. These parameters vary at every execution and require dedicated instrumentation built into the Pilot system to be measured. Without consistent performance models and set of probes, performance comparison among Pilot systems appears unfeasible.

5. DISCUSSION AND CONCLUSION

We introduced the Pilot abstraction in §3 describing the capabilities, components, and architecture pattern of Pilot systems. We also defined a terminology consistent across Pilot systems clarifying the meaning of "pilot", "job", and their cognate concepts. In §4 we offered a classification of the core and auxiliary properties of Pilot system implementations, and we analyzed a set of exemplars. Considered altogether, these contributions outline a paradigm for the execution of heterogeneous, multi-task workloads via multi-entity and multi-stage scheduling on DCR resource placeholders. This computing paradigm is here referred to as "Pilot paradigm".

5.1 The Pilot Paradigm

The generality of the Pilot paradigm may come as a surprise when considering that, traditionally, Pilot systems have been implemented to optimize the throughput of single-core (or at least single-node), short-lived, uncoupled tasks execution [3, 194, 195]. For example DIANE, DIRAC, MyCluster, PanDA, or HTCondor Glidein and GlideinWMS were initially developed to focus on either a type of workload, a specific infrastructure, or the optimization of a single performance metric.

The Pilot paradigm is general because the execution of a workload via multi-entity and multi-stage scheduling on DCR resource placeholders does not have to depend on a single type of workload, DCR, or resource. In principle, systems implementing the Pilot paradigm can execute workloads composed of an arbitrary number of tasks with diverse data, compute, and networking requirements. The same generality applies to the types of DCR and of resource on which a Pilot system executes workloads.[5]

The analysis presented in §4, shows how Pilot systems have progressed to implement the generality of the Pilot paradigm. Pilot systems are now engineered to execute homogeneous or heterogeneous workloads; these workloads can be comprised of independent or intercommunicating tasks of arbitrary duration or data and computation requirements. These workloads can also be executed on an increasingly diverse pool of DCRs. Pilot systems were originally designed for DCR with HTC grid middleware; Pilot systems have emerged that are capable of also operating on DCRs with HPC and cloud middleware.

As seen in §3, the Pilot paradigm demands resource placeholders but does not specify the type of resource that the placeholder should expose. In principle, pilots can also be placeholders for data or network resources, either exclusively or in conjunction with compute resources. For example, in Ref. [97] the concept of Pilot-Data was conceived to be fundamental to dynamic data placement and scheduling as Pilot is to computational tasks. The concept of "Pilot networks" was introduced in Ref. [196] in reference to Software-Defined Networking [197] and User-Schedulable Network paths. [198]

The generality of the Pilot paradigm also promotes the adoption of Pilot functionalities and systems by other middleware and tools. For example, Pilot systems have been successfully integrated within workflow systems to support optimal execution of workloads with articulated data and single or multi-core task dependencies [103, 132, 199]. As such, not only can throughput be optimized for multi-core, long-lived, coupled tasks executions, but also for optimal data/compute placement, and dynamic resource sizing.

The Pilot paradigm is not limited to academic projects and scientific experiments. Hadoop [200] introduced the YARN [201] resource manager for heterogeneous workloads. YARN supports multi-entity and multi-stage scheduling: applications initialize an "Application-Master" via YARN; the Application Master allocates resources in "containers" for the applications; and YARN then can execute tasks in these containers (i.e., resource placeholders). TEZ [202], a DAG processing engine primarily designed to support the Hive SQL engine [203], enables applications to hold containers across the DAG execution without de/reallocating resources. Independent of the Hadoop developments, Google's Kubernetes [204] is emerging as a leading container management approach. Not completely coincidently, Kubernetes is the Greek term for the English "Pilot".

5.2 Future Directions and Challenges

The Pilot landscape is currently fragmented with duplicated effort and capabilities. The reasons for this balka-

[5]The generality of the pilot paradigm across workload, DCR, and resource types was first discussed in Ref. [179], wherein an initial conceptual model for Pilot systems was proposed. The introduction of the pilot architecture pattern and the discussion in §3 and §4 enhances and extends the preliminary analysis of Ref. [179].

nization can be traced back mainly to two factors: (i) the relatively recent discovery of the generality and relevance of the Pilot paradigm; and (ii) the development model fostered within academic institutions.

As seen in §2 and §4, Pilot systems were developed as a pragmatic solution to improve the throughput of distributed applications, and were designed as local and point solutions. Pilot systems were not thought from their inception as an independent system, but, at best, as a module within a framework. Inheriting the development model of the scientific projects within which they were initially developed, Pilot systems were not engineered to promote (re)usability, modularity, open interfaces, or long-term sustainability. Collectively, this resulted in duplication of development effort across frameworks and projects, and hindered the appreciation for the generality of the Pilot abstraction, the theoretical framework underlying the Pilot systems, and the paradigm for application execution they enable.

Consistent with this analysis, many of the Pilot systems described in §4.3 offer a set of overlapping functionalities. This duplication may have to be reduced in the future to promote maintainability, robustness, interoperability, extensibility, and overall capabilities of existing Pilot systems. As seen in §4.4, Pilot systems are already progressively supporting diverse DCRs and types of workload. This trend might lead to consolidation and to increased adoption of multi-purpose Pilot systems. The scope of the consolidation process will depend on the diversity of used programming languages, deployment models, interaction with existing applications, and how they will be addressed.

The analysis proposed in this paper suggests critical commonalities across Pilot systems stemming from a shared architectural pattern, abstraction, and computing paradigm. Models of pilot functionality can be grounded on these commonalities, as well as be reflected in the definition of unified and open interfaces for the users, applications, and DCRs. End-users, developers, and DCR administrators could rely upon these interfaces, which would promote better integration of Pilot systems into application and resource-facing middlware.

There is evidence of ongoing integration and consolidation processes, such as the adoption of extensible workload management capabilities or utilization of similar resource interoperability layers. For example, PanDA is iterating its development cycle and the resulting system, called "Big PanDA" is now capable of opportunistically submitting pilots to the Titan supercomputer [205] at the Oak Ridge Leadership Computing Facility (OLCF) [186, 206]. Further, Big PanDA has adopted SAGA, an open and standardized DCR interoperability library developed independent of Pilot systems but now adopted both by Big Panda and RADICAL-Pilot.

5.3 Summary and Contributions

This paper contributes to the understanding, design, and adoption of Pilot systems by characterizing the Pilot abstraction, the Pilot paradigm, and exemplar implementations.

We provided an analysis of the technical origins and motivations of Pilot systems in §2 and we summarized their chronological development in Figure 1. We described the logical components and functionalities that constitute the Pilot abstraction in §3, and we outlined them in Figure 2. We then defined a consistent terminology to clarify the heterogeneity of the Pilot systems landscape, and we used this terminology together with the logical components and functionalities of the Pilot systems to specify the pilot architecture pattern in Figure 3.

We defined the core and auxiliary properties of Pilot system implementations in §4 (Tables 1 and 2). We then used these properties alongside the contributions offered in §3 to describe seven exemplar Pilot system implementations. We gave details about their architecture and execution model showing how they conformed to the pilot architecture paradigm we defined in §3.2. We summarized this analysis in Figures 4–10.

We used the Pilot abstraction and insight about Pilot systems, their motivations and diverse implementations to highlight the properties of the Pilot paradigm in §5. We argued for the generality of the Pilot paradigm on the basis of demonstrated generality of the type of workload and use cases Pilot systems can execute, as well as a lack of constraints on the type of DCR that can be used or on the type of resource exposed by the pilots. Finally, we reviewed the benefits that a more structured approach to the conceptualization and design of Pilot systems may offer.

With this paper, we also contributed a methodology to evaluate software systems that have developed organically and without an established theoretical framework. This methodology is composed of five steps: (i) analysis of the abstraction(s) underlying the observed software system implementations; (ii) the definition of a consistent terminology to reason about abstractions; (iii) the evaluation of the components and functionalities that may constitute a specific architectural pattern for the implementation of that abstraction; (iv) the definition of core and auxiliary implementation properties; (v) the evaluation of implementations.

The application of this methodology offers the opportunity to uncover the theoretical framework underlying the observed software systems, and to understand whether such systems are implementations of a well-defined and independent abstraction. This theoretical framework can be used to inform or understand the development and engineering of software systems without mandating specific design, representation, or development methodologies or tools.

Workflow systems are amenable to be studied with the methodology proposed and used in this paper. Multiple workflow systems have been developed independently to serve diverse use cases and be executed on heterogeneous DCRs. In spite of broad surveys [52, 207–209] about workflow systems and their usage scenarios, an encompassing theoretical framework for the underlying abstraction, or set of abstractions if any, is not yet available. This is evident in the state of workflow systems which shows a significant duplication of effort, limited extensibility and interoperability, and proprietary solutions for interfaces to both the resource and application layers.

Acknowledgements and Author Contributions

This work is funded by the Department of Energy Award (ASCR) DE-FG02-12ER26115, NSF CAREER ACI-1253644 and NSF ACI-1440677 "RADICAL-Cybertools". We thank former and current members of the RADICAL group for helpful discussions, comments, and criticisms. We also thank members of the AIMES project for helpful discussions, in particular Daniel S. Katz for comments. We thank Rajiv Ramnath for useful discussions that helped to establish connections between this work and software architecture patterns. MT wrote the paper. MT and SJ co-organized and

edited the paper. MS contributed to an early draft of parts of Section 2 and 4.

6. REFERENCES

[1] D. S. Katz, S. Jha, M. Parashar, O. Rana, and J. Weissman, "Survey and analysis of production distributed computing infrastructures," *arXiv preprint arXiv:1208.2649*, 2012.

[2] C. Sehgal, "Opportunistic eco-system & OSG-XD update," Presentation at OSG Council Meeting, April 11, 2014, https://twiki.opensciencegrid.org/twiki/bin/viewfile/Council/April-2014/OSG-XD_Report_to_Council_11apr2014.pdf, accessed: 2015-11-6.

[3] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein *et al.*, "The open science grid," in *Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) conference, Journal of Physics: Conference Series*, vol. 78(1). IOP Publishing, 2007, p. 012057.

[4] Open Science Grid (OSG), http://www.opensciencegrid.org/, accessed: 2015-11-5.

[5] K. De, A. Klimentov, T. Wenaus, T. Maeno, and P. Nilsson, "PanDA: A new paradigm for distributed computing in HEP through the lens of ATLAS and other experiments," ATL-COM-SOFT-2014-027, Tech. Rep., 2014.

[6] G. Aad, E. Abat, J. Abdallah, A. Abdelalim, A. Abdesselam, O. Abdinov, B. Abi, M. Abolins, H. Abramowicz, E. Acerbi *et al.*, "The ATLAS experiment at the CERN large hadron collider," *Journal of Instrumentation*, vol. 3, no. 08, p. S08003, 2008.

[7] LHC Study Group, "The large hadron collider, conceptual design," CERN/AC/95-05 (LHC) Geneva, Tech. Rep., 1995.

[8] D. Bonacorsi, T. Ferrari *et al.*, "WLCG service challenges and tiered architecture in the LHC era," in *IFAE 2006*. Springer, 2007, pp. 365–368.

[9] Worldwide LHC Computing Grid (WLCG), The Apache software foundation, http://wlcg.web.cern.ch/, accessed: 2015-11-5.

[10] G. Juve, "The Glidein service," Presentation, http://www.slideserve.com/embed/5100433, accessed: 2015-11-5.

[11] I. Sfiligoi, "glideinWMS—a generic pilot-based workload management system," in *Proceedings of the international conference on computing in high energy and nuclear physics (CHEP2007), Journal of Physics: Conference Series*, vol. 119(6). IOP Publishing, 2008, p. 062044.

[12] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for clouds and grids," in *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2011, pp. 114–121.

[13] J. T. Mościcki, "DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, vol. 3. IEEE, 2003, pp. 1617–1620.

[14] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev *et al.*, "DIRAC pilot framework and the DIRAC Workload Management System," in *Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series*, vol. 219(6). IOP Publishing, 2010, p. 062049.

[15] P.-H. Chiu and M. Potekhin, "Pilot factory – a Condor-based system for scalable Pilot Job generation in the Panda WMS framework," in *Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series*, vol. 219(6). IOP Publishing, 2010, p. 062041.

[16] A. Rubio-Montero, E. Huedo, F. Castejón, and R. Mayo-García, "GWpilot: Enabling multi-level scheduling in distributed infrastructures with gridway and pilot jobs," *Future Generation Computer Systems*, vol. 45, pp. 25–52, 2015.

[17] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid," in *Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, vol. 1. IEEE, 2000, pp. 283–289.

[18] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 8th ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 43.

[19] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment," in *Proceedings of the IEEE Challenges of Large Applications in Distributed Environments (CLADE) workshop*. IEEE, 2006, pp. 95–103.

[20] J. Mościcki, M. Lamanna, M. Bubak, and P. M. Sloot, "Processing moldable tasks on the grid: Late job binding with lightweight user-level overlay," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 725–736, 2011.

[21] T. Glatard and S. Camarasu-Pop, "Modelling pilot-job applications on production grids," in *Proceedings of Euro-Par 2009 – Parallel Processing Workshops*. Springer, 2010, pp. 140–149.

[22] A. Delgado Peris, J. M. Hernandez, and E. Huedo, "Distributed scheduling and data sharing in late-binding overlays," in *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2014, pp. 129–136.

[23] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers," 2015, (under review) http://arxiv.org/abs/1512.08194.

[24] L. F. Richardson, *Weather prediction by numerical process*. Cambridge University Press, 1922, reprinted in 2006.

[25] P. Lynch, "Richardson's marvelous forecast," in *The life cycles of extratropical cyclones*. Springer, 1999,

pp. 61–73.

[26] P. Wegner, "Why interaction is more powerful than algorithms," *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, 1997.

[27] V. Balasubramanian, A. Trekalis, O. Weidner, and S. Jha, "Ensemble-MD Toolkit: Scalable and Flexible Execution of Ensembles of Molecular Simulations," 2016 (Submitted, Under Review), http://arxiv.org/abs/1602.00678.

[28] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "An enabling framework for master-worker applications on the computational grid," in *Proceedings of the 9th International Symposium on High-Performance Distributed Computing*. IEEE, 2000, pp. 43–50.

[29] E. Heymann, M. A. Senar, E. Luque, and M. Livny, "Adaptive scheduling for master-worker applications on the computational grid," in *Grid Computing—GRID 2000*. Springer, 2000, pp. 214–227.

[30] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive parallelism and piranha," *Computer*, vol. 28, no. 1, pp. 40–49, 1995.

[31] L. F. Sarmenta and S. Hirano, "Bayanihan: Building and studying web-based volunteer computing systems using java," *Future Generation Computer Systems*, vol. 15, no. 5, pp. 675–686, 1999.

[32] H. A. Schmidt, K. Strimmer, M. Vingron, and A. von Haeseler, "TREE-PUZZLE: maximum likelihood phylogenetic analysis using quartets and parallel computing," *Bioinformatics*, vol. 18, no. 3, pp. 502–504, 2002.

[33] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. Pearson Education, 2005.

[34] B. Freisleben and T. Kielmann, "Coordination patterns for parallel computing," in *Coordination Languages and Models*. Springer, 1997, pp. 414–417.

[35] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," in *Job Scheduling Strategies for Parallel Processing*. Springer, 1998, pp. 62–82.

[36] J. H. Katz, "Simulation of a multiprocessor computer system," in *Proceedings of the Spring joint computer conference*. ACM, 1966, pp. 127–139.

[37] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Wiley, 9^{th} Edition, 2012.

[38] A. B. Downey, "Predicting queue times on space-sharing parallel computers," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997, pp. 209–218.

[39] R. Wolski, "Experiences with predicting resource performance on-line in computational grid settings," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 4, pp. 41–49, 2003.

[40] H. Li, D. Groep, J. Templon, and L. Wolters, "Predicting job start times on clusters," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004, pp. 301–308.

[41] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.

[42] T. R. Furlani, B. L. Schneider, M. D. Jones, J. Towns, D. L. Hart, S. M. Gallo, R. L. DeLeon, C.-D. Lu, A. Ghadersohi, R. J. Gentner, A. K. Patra, G. von Laszewski, F. Wang, J. T. Palmer, and N. Simakov, "Using XDMoD to facilitate XSEDE operations, planning and analysis," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, ser. XSEDE '13. ACM, 2013, pp. 46:1–46:8.

[43] C.-D. Lu, J. Browne, R. L. DeLeon, J. Hammond, W. Barth, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra, "Comprehensive job level resource usage measurement and analysis for XSEDE HPC systems," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, ser. XSEDE '13. ACM, 2013, pp. 50:1–50:8.

[44] C. Li and L. Li, "Multi-level scheduling for global optimization in grid computing," *Computers & Electrical Engineering*, vol. 34, no. 3, pp. 202–221, 2008.

[45] G. Weikum, "A theoretical foundation of multi-level concurrency control," in *Proceedings of the 5th ACM SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 1985, pp. 31–43.

[46] G. Singh, C. Kesselman, and E. Deelman, "Optimizing grid-based workflow execution," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 201–219, 2005.

[47] L. Ramakrishnan, D. Irwin, L. Grit, A. Yumerefendi, A. Iamnitchi, and J. Chase, "Toward a doctrine of containment: grid hosting with adaptive resource control," in *Proceedings of the ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 101.

[48] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Proceedings of the Grid Computing Environments Workshop*. Ieee, 2008, pp. 1–10.

[49] G. Juve and E. Deelman, "Resource provisioning options for large-scale scientific workflows," in *Proceedings of the IEEE 4th International Conference on e-Science*. IEEE, 2008, pp. 608–613.

[50] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2012, pp. 612–619.

[51] Y. Song, H. Wang, Y. Li, B. Feng, and Y. Sun, "Multi-tiered on-demand resource scheduling for VM-based data center," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE Computer Society, 2009, pp. 148–155.

[52] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds., *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., 2007.

[53] V. Curcin and M. Ghanem, "Scientific workflow systems-can one size fit all?" in *Proceedings of the Biomedical Engineering Conference (CIBEC)*. IEEE,

2008, pp. 1–9.

[54] J. R. Balderrama, T. T. Huu, and J. Montagnat, "Scalable and resilient workflow executions on production distributed computing infrastructures," in *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2012, pp. 119–126.

[55] F. Berman, G. Fox, and A. J. Hey, *Grid computing: making the global infrastructure a reality*. Wiley, 2003, vol. 2.

[56] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[57] A. Legrand, L. Marchal, and H. Casanova, "Scheduling distributed applications: the simgrid simulation framework," in *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 2003, pp. 138–145.

[58] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software: Practice and Experience*, vol. 32, no. 2, pp. 135–64, 2002.

[59] F. Darema, "Grid computing and beyond: The context of dynamic data driven applications systems," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 692–697, 2005.

[60] R. A. Nobrega, A. F. Barbosa, I. Bediaga, G. Cernicchiaro, E. C. De Oliveira, J. Magnin, L. M. De Andrade Filho, J. M. De Miranda, H. P. L. Junior, A. Reis *et al.*, "LHCb computing technical design report," CERN, Tech. Rep. CERN-LHCC-2005-019 ; LHCb-TDR-11, 06 2005.

[61] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-level scheduling on distributed heterogeneous networks," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE, 1996, pp. 39–39.

[62] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf *et al.*, "Adaptive computing on the grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, 2003.

[63] H. Casanova, F. Berman, G. Obertelli, and R. Wolski, "The AppLeS parameter sweep template: User-level middleware for the grid," in *Proceedings of the ACM/IEEE 2000 Supercomputing Conference*. IEEE, 2000, pp. 60–60.

[64] G. Woltman, S. Kurowski *et al.*, The great Internet Mersenne prime search, http://www.mersenne.org, accessed: 2015-11-5.

[65] G. Lawton, "Distributed net applications create virtual supercomputers," *Computer*, no. 6, pp. 16–20, 2000.

[66] The RSA Laboratories Secret-Key Challenge, RSA Laboratories, http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-laboratories-secret-key-challenge.htm, accessed: 2015-11-5.

[67] SETI@home, Berkeley University, http://setiathome.ssl.berkeley.edu/, accessed: 2015-11-5.

[68] Berkeley Open Infrastructure for Network Computing (BOINC), Berkeley University, https://boinc.berkeley.edu/, accessed: 2015-11-5.

[69] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE, 2004, pp. 4–10.

[70] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[71] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, "A worldwide flock of Condors: Load sharing among workstation clusters," *Future Generation Computer Systems*, vol. 12, no. 1, pp. 53–65, 1996.

[72] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[73] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: a tool for performing parametrised simulations using distributed workstations," in *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 1995, pp. 112–121.

[74] N. Jacq, J. Salzemann, F. Jacq, Y. Legré, E. Medernach, J. Montagnat, A. Maaß, M. Reichstadt, H. Schwichtenberg, M. Sridhar *et al.*, "Grid-enabled virtual screening against malaria," *Journal of Grid Computing*, vol. 6, no. 1, pp. 29–43, 2008.

[75] E. Laure and B. Jones, "Enabling grids for e-Science: The EGEE project," *Grid computing: infrastructure, service, and applications*, p. 55, 2009.

[76] Large Hadron Collider (LHC), CERN, http://home.web.cern.ch/topics/large-hadron-collider, accessed: 2015-11-5.

[77] J. Moscicki, "The DIANE user-scheduler provides quality of service," CERN Computer Newletter, http://cerncourier.com/cws/article/cnl/25828, 2006, accessed: 2015-11-8.

[78] DIANE, CERN, http://it-proj-diane.web.cern.ch/, accessed: 2015-11-5.

[79] P. Saiz, L. Aphecetche, P. Bunčić, R. Piskač, J.-E. Revsbech, V. Šego, A. Collaboration *et al.*, "AliEn: ALICE environment on the GRID," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2, pp. 437–440, 2003.

[80] AliEn, CERN, http://alien2.cern.ch/, accessed: 2015-11-5.

[81] A. Tsaregorodtsev, V. Garonne, J. Closier, M. Frank, C. Gaspar, E. van Herwijnen, F. Loverre, S. Ponce, R. G. Diaz, D. Galli *et al.*, "Dirac–distributed infrastructure with remote agent control," in *Proceedings of the Conference for Computing in High Energy and Nuclear Physics*, 2003.

[82] DIRAC, LHCb, http://diracgrid.org/, accessed: 2015-11-5.

[83] PanDA, CERN, https://twiki.cern.ch/twiki/bin/view/PanDA/PanDA, accessed: 2015-11-5.

[84] GlideinWMS starting date, US CMS, http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/doc.prd/history.html#old, accessed: 2015-11-8.

[85] GlideinWMS, US CMS, http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/doc.prd/index.html, accessed: 2015-11-5.

[86] The LHCb experiment, CERN, http://lhcb-public.web.cern.ch/lhcb-public/, accessed: 2015-11-5.

[87] A. Collaboration, K. Aamodt *et al.*, "The ALICE experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, no. 420, p. S08002, 2008.

[88] U.S. CMS, http://www.uscms.org/index.shtml, accessed: 2015-11-12.

[89] S. Chatrchyan, V. Khachatryan, A. M. Sirunyan, A. Tumasyan, W. Adam, E. Aguilo, T. Bergauer, M. Dragicevic, J. Erö, C. Fabjan *et al.*, "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC," *Physics Letters B*, vol. 716, no. 1, pp. 30–61, 2012.

[90] P. Buncic and A. Harutyunyan, "Co-Pilot: The distributed job execution framework," CERN, Tech. Rep., 03 2011.

[91] A. Harutyunyan, J. Blomer, P. Buncic, I. Charalampidis, F. Grey, A. Karneyeu, D. Larsen, D. L. González, J. Lisec, B. Segal *et al.*, "CernVM Co-Pilot: an extensible framework for building scalable computing infrastructures on the cloud," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2012), Journal of Physics: Conference Series*, vol. 396(3). IOP Publishing, 2012, p. 032054.

[92] CERN public computing challenge 2015, CERN, https://test4theory.cern.ch/challenge/, accessed: 2015-11-12.

[93] Token Pool Server (ToPoS), SARA, https://grid.surfsara.nl/wiki/index.php/Using_the_Grid/ToPoS, accessed: 2015-11-12.

[94] Stichting Academisch Rekencentrum Amsterdam (SARA), SURF, https://grid.surfsara.nl/wiki/index.php/Main_Page, accessed: 2015-11-12.

[95] ToPoS manual, SARA, https://docs.google.com/document/d/10VLX9ko75gwdDkISKTBpCZ8wxw6on_bW6WyHAZqpzfY/, accessed: 2015-11-12.

[96] A. Luckow, L. Lacinski, and S. Jha, "SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2010, pp. 135–144.

[97] A. Luckow, M. Santcroos, A. Zebrowski, and S. Jha, "Pilot-Data: An Abstraction for Distributed Data," *Journal Parallel and Distributed Computing*, October 2014, http://dx.doi.org/10.1016/j.jpdc.2014.09.009.

[98] A. Merzky, O. Weidner, and S. Jha, "SAGA: A standardized access layer to heterogeneous distributed computing infrastructure" *Software-X*, 2015, dOI: 10.1016/j.softx.2015.03.001. [Online]. Available: http://dx.doi.org/10.1016/j.softx.2015.03.001

[99] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf, "SAGA: A Simple API for Grid Applications. high-level application programming on the grid," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 7–20, 2006.

[100] A. J. Rubio-Montero, F. Castejón, E. Huedo, M. Rodríguez-Pascual, and R. Mayo-García, "Performance improvements for the neoclassical transport calculation on grid by means of pilot jobs," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2012, pp. 609–615.

[101] E. Huedo, R. S. Montero, and I. M. Llorente, "A modular meta-scheduling architecture for interfacing with pre-ws and ws grid resource management services," *Future Generation Computer Systems*, vol. 23, no. 2, pp. 252–261, 2007.

[102] M. Rynge, G. Juve, G. Mehta, E. Deelman, G. B. Berriman, K. Larson, B. Holzman, S. Callaghan, I. Sfiligoi, and F. Würthwein, "Experiences using glideinWMS and the corral frontend across cyberinfrastructures," in *Proceedings of the IEEE 7th International Conference on e-Science*. IEEE, 2011, pp. 311–318.

[103] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[104] D. Weitzel, D. Fraser, B. Bockelman, and D. Swanson, "Campus grids: Bringing additional computational resources to HEP researchers," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2012), Journal of Physics: Conference Series*, vol. 396(3). IOP Publishing, 2012, p. 032116.

[105] Coasters, ANL, http://wiki.cogkit.org/wiki/Coasters, accessed: 2015-11-5.

[106] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[107] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[108] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," IBM Research Division, Tech. Rep. RC25482 (AUS1407-001), 2014.

[109] G. Agha, "Abstracting interaction patterns: A programming paradigm for open distributed systems," *Formal Methods for Open Object-based Distributed Systems*, vol. 1, 1997.

[110] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, *Coordination of Internet agents: Models, technologies, and applications*. Springer, 2013.

[111] D. P. Da Silva, W. Cirne, and F. V. Brasileiro, "Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational

grids," in *Euro-Par 2003 Parallel Processing*. Springer, 2003, pp. 169–180.

[112] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. Silva, C. O. Barros, and C. Silveira, "Running bag-of-tasks applications on computational grids: The mygrid approach," in *Proceedings of the International Conference on Parallel Processing*. IEEE, 2003, pp. 407–416.

[113] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proceedings of the Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. IEEE, 2008, pp. 1–11.

[114] T. L. Sterling, *Beowulf cluster computing with Linux*. MIT press, 2002.

[115] AWS and High Performance Computing, Amazon.com, https://aws.amazon.com/hpc/, accessed: 2015-11-5.

[116] Nomad, HashiCorp, https://www.hashicorp.com/blog/nomad.html, accessed: 2015-11-5.

[117] C. Pinchak, P. Lu, and M. Goldenberg, "Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences," in *Proceedings of the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2002, pp. 205–228.

[118] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. Wiley, 2007, vol. 5.

[119] P. Kruchten and J. Stafford, "The past, present, and future for software architecture," *IEEE Software*, vol. 23, no. 2, pp. 22–30, 2006.

[120] R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2014.

[121] P. B. Kruchten, "Architectural blueprints—the "4+1" view model of software architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[122] K. Raymond, "Reference model of open distributed processing (rm-odp): Introduction," in *Open Distributed Processing*. Springer, 1995, pp. 3–14.

[123] J. Zachman, *The zachman framework for enterprise architecture*. Zachman Framework Associates, 2006.

[124] A. Josey, *TOGAF® Version 9.1 A Pocket Guide*. Van Haren, 2011.

[125] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood, "Attribute-driven design (ADD), version 2.0," DTIC Document, Tech. Rep., 2006.

[126] V. V. Korkhov, J. T. Moscicki, and V. V. Krzhizhanovskaya, "Dynamic workload balancing of parallel applications with user-level scheduling on the grid," *Future Generation Computer Systems*, vol. 25, no. 1, pp. 28–34, 2009.

[127] DAGMan, UW-Madison, https://research.cs.wisc.edu/htcondor/dagman/dagman.html, accessed: 2015-11-5.

[128] Ensemble MD Toolkit, RADICAL CyberTools, http://radical-cybertools.github.io/ensemble-md/index.html, accessed: 2015-11-5.

[129] Distributed System Laboratory (DSL), University of Chicago, http://dsl-wiki.cs.uchicago.edu/index.php/Main_Page, accessed: 2015-11-5.

[130] Swift Project, University of Chicago, http://swift-lang.org/main/index.php, accessed: 2015-11-5.

[131] Java CoG Kit, Globus Alliance, https://wiki.cogkit.org/wiki/Main_Page, accessed: 2015-11-5.

[132] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proceedings of the IEEE Congress on Services*. IEEE, 2007, pp. 199–206.

[133] G. Von Laszewski, I. Foster, and J. Gawor, "CoG kits: a bridge between commodity distributed computing and high-performance grids," in *Proceedings of the ACM conference on Java Grande*. ACM, 2000, pp. 97–106.

[134] G. von Laszewski and M. Hategan, "Workflow concepts of the java cog kit," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 239–258, 2005.

[135] Swift user guide, University of Chicago, http://swift-lang.org/guides/release-0.96/userguide/userguide.html, accessed: 2015-11-5.

[136] European Organization for Nuclear Research (CERN), http://home.web.cern.ch/, accessed: 2015-11-5.

[137] European Grid Infrastructure (EGI), http://www.egi.eu/, accessed: 2015-11-5.

[138] J. Mościcki, H. Lee, S. Guatelli, S. Lin, and M. Pia, "Biomedical applications on the grid: Efficient management of parallel jobs," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, vol. 4. IEEE, 2004, pp. 2143–2147.

[139] N. Jacq, V. Breton, H.-Y. Chen, L.-Y. Ho, M. Hofmann, V. Kasam, H.-C. Lee, Y. Legré, S. C. Lin, A. Maaß *et al.*, "Virtual screening on large scale grids," *Parallel Computing*, vol. 33, no. 4, pp. 289–301, 2007.

[140] J. Mościcki, "Distributed analysis environment for HEP and interdisciplinary applications," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2, pp. 426–429, 2003.

[141] V. Bacu, D. Mihon, D. Rodila, T. Stefanut, and D. Gorgan, "gSWAT platform for grid based hydrological model calibration and execution," in *Proceedings of the 10th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2011, pp. 288–291.

[142] A. Mantero, B. Bavdaz, A. Owens, T. Peacock, and M. Pia, "Simulation of X-ray fluorescence and application to planetary astrophysics," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, vol. 3. IEEE, 2003, pp. 1527–1529.

[143] DIANE API and reference documentation, CERN, http://it-proj-diane.web.cern.ch/it-proj-diane/install/2.4/doc/reference/html/index.html, accessed: 2015-11-5.

[144] J. Mościcki, "Understanding and mastering dynamics in computing grids: processing moldable tasks with user-level overlay," Ph.D. dissertation, University of

Amsterdam, 2011.

[145] J. Mościcki, F. Brochu, J. Ebke, U. Egede, J. Elmsheuser, K. Harrison, R. Jones, H. Lee, D. Liko, A. Maier *et al.*, "GANGA: a tool for computational-task management and easy access to grid resources," *Computer Physics Communications*, vol. 180, no. 11, pp. 2303–2316, 2009.

[146] GANGA, CERN, https://ganga.web.cern.ch/, accessed: 2015-11-5.

[147] G. Grzeslo, T. Szepieniec, and M. Bubak, "DAG4DIANE-enabling dag-based applications on DIANE framework," in *Proceedings of the Cracow Grid Workshop*. Cyfronet, 2009, pp. 310–313.

[148] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR," *International Journal of High Performance Computing Applications*, vol. 22, no. 3, pp. 347–360, 2008.

[149] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees, "DIRAC: A scalable lightweight architecture for high throughput computing," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004, pp. 19–25.

[150] F. Pacini and A. Maraschini, "Job description language (JDL) attributes specification," EGEE Consortium, Tech. Rep. 590869, 2006.

[151] A. Tsaregorodtsev, M. Sanchez, P. Charpentier, G. Kuznetsov, J. Closier, R. Graciani, I. Stokes-Rees, A. C. Smith, N. Brook, J. Saborido Silva *et al.*, "DIRAC, the lhcb data production and distributed analysis system," Presented at the International Conference on Computing in High Energy and Nuclear Physics (CHEP), Tech. Rep., 2006.

[152] HTCondor Manual v7.6.10, Glidein, UW-Madison, http://research.cs.wisc.edu/htcondor/manual/v7.6.10/5_4Glidein.html, accessed: 2015-11-5.

[153] The Center for High Throughput Computing (CHTC), UW-Madison, http://chtc.cs.wisc.edu/, accessed: 2015-11-5.

[154] HTCondor, UW-Madison, https://research.cs.wisc.edu/htcondor/, accessed: 2015-11-5.

[155] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International journal of high performance computing applications*, vol. 15, no. 3, pp. 200–222, 2001.

[156] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro, "VOMS, an authorization system for virtual organizations," in *Proceedings of the 1st European Across Grids Conference*. Springer, 2004, pp. 33–40.

[157] GlideinWMS manual, US CMS, http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/doc.v1_0/manual/index.html, accessed: 2015-11-5.

[158] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in Condor," in *Workflows for e-Science*. Springer, 2007, pp. 357–375.

[159] HTCondor ClassAd, UW-Madison, http://research.cs.wisc.edu/htcondor/manual/v7.8/4_1HTCondor_s_ClassAd.html, accessed: 2015-11-5.

[160] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Personal adaptive clusters as containers for scientific jobs," *Cluster Computing*, vol. 10, no. 3, pp. 339–350, 2007.

[161] E. Walker, MyCluster archive, https://sites.google.com/site/ewalker544/research-2/mycluster, accessed: 2015-11-5.

[162] Texas Advanced Computing Center (TACC), University of Texas, https://www.tacc.utexas.edu/, accessed: 2015-11-5.

[163] TeraGrid archive, XSEDE, https://www.xsede.org/tg-archives, accessed: 2015-11-5.

[164] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual clusters in a grid site manager," in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 2003, pp. 90–100.

[165] OpenPBS homepage, ANL, http://www.mcs.anl.gov/research/projects/openpbs/, accessed: 2015-11-5.

[166] E. Walker, T. Minyard, and J. Boisseau, "GridShell: A login shell for orchestrating and coordinating applications in a grid enabled environment," in *Proceedings of the International Conference on Computing, Communications and Control Technologies*, 2004, pp. 182–187.

[167] X. Zhao, J. Hover, T. Wlodek, T. Wenaus, J. Frey, T. Tannenbaum, M. Livny, A. Collaboration *et al.*, "PanDA pilot submission using Condor-G: experience and improvements," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2010), Journal of Physics: Conference Series*, vol. 331(7). IOP Publishing, 2011, p. 072069.

[168] T. Maeno, K. De, and S. Panitkin, "PD2P: PanDA dynamic data placement for ATLAS," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2012), Journal of Physics: Conference Series*, vol. 396(3). IOP Publishing, 2012, p. 032070.

[169] P. Nilsson, J. C. Bejar, G. Compostella, C. Contreras, K. De, T. Dos Santos, T. Maeno, M. Potekhin, and T. Wenaus, "Recent improvements in the ATLAS PanDA pilot," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2012), Journal of Physics: Conference Series*, vol. 396(3). IOP Publishing, 2012, p. 032080.

[170] PanDA Architecture, CERN, https://twiki.cern.ch/twiki/bin/view/PanDA/PanDA#Architecture_and_workflow, accessed: 2015-11-5.

[171] T. Maeno, K. De, T. Wenaus, P. Nilsson, G. Stewart, R. Walker, A. Stradling, J. Caballero, M. Potekhin, D. Smith *et al.*, "Overview of atlas panda workload management," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2011), Journal of Physics: Conference Series*, vol. 331(7). IOP Publishing, 2011, p. 072024.

[172] P. Nilsson, J. Caballero, K. De, T. Maeno, A. Stradling, T. Wenaus *et al.*, "The ATLAS PanDA

pilot in operation," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2010), Journal of Physics: Conference Series*, vol. 331(6). IOP Publishing, 2011, p. 062040.

[173] PandaRun, CERN, https://twiki.cern.ch/twiki/bin/view/PanDA/PandaRun, accessed: 2015-11-5.

[174] D. Crooks, P. Calafiura, R. Harrington, M. Jha, T. Maeno, S. Purdie, H. Severini, S. Skipsey, V. Tsulaia, R. Walker *et al.*, "Multi-core job submission and grid resource scheduling for ATLAS AthenaMP," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2012), Journal of Physics: Conference Series*, vol. 396(3). IOP Publishing, 2012, p. 032115.

[175] J. Schovancova, M. Potekhin, K. De, A. Klimentov, P. Love, and T. Wenaus, "The next generation of the ATLAS PanDA monitoring system," ATL-COM-SOFT-2014-011, Tech. Rep., 2014.

[176] J. Schovancova, K. De, S. Panitkin, D. Yu, T. Maeno, D. Oleynik, A. Petrosyan, A. Klimentov, T. Wenaus, P. Nilsson *et al.*, "PanDA beyond ATLAS: Workload management for data intensive science," ATL-COM-SOFT-2013-122, Tech. Rep., 2013.

[177] M. Borodin, J. E. García Navarro, T. Maeno, D. Golubkov, A. Vaniachine, K. De, and A. Klimentov, "Scaling up ATLAS production system for the LHC run 2 and beyond: project ProdSys2," ATL-COM-SOFT-2015-059, Tech. Rep., 2015.

[178] T. Maeno, K. De, A. Klimentov, P. Nilsson, D. Oleynik, S. Panitkin, A. Petrosyan, J. Schovancova, A. Vaniachine, T. Wenaus *et al.*, "Evolution of the ATLAS PanDA workload management system for exascale computational science," in *Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013), Journal of Physics: Conference Series*, vol. 513(3). IOP Publishing, 2014, p. 032062.

[179] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, "P*: A model of pilot-abstractions," *IEEE 8th International Conference on e-Science*, pp. 1–10, 2012, http://dx.doi.org/10.1109/eScience.2012.6404423.

[180] "RADICAL: Research in Advanced Distributed Cyberinfrastructure and Applications Laboratory, Rutgers University," accessed: 2015-11-5. [Online]. Available: http://radical.rutgers.edu

[181] RADICAL-Pilot, RADICAL CyberTools, http://radical-cybertools.github.io/radical-pilot/index.html, accessed: 2015-11-5.

[182] S.-H. Ko, N. Kim, J. Kim, A. Thota, and S. Jha, "Efficient Runtime Environment for Coupled Multi-physics Simulations: Dynamic Resource Allocation and Load-Balancing," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 349–358. [Online]. Available: http://www.cct.lsu.edu/~sjha/select_publications/coupled_simulations_framework.pdf

[183] J. Kim, W. Huang, S. Maddineni, F. Aboul-Ela, and S. Jha, "Exploring the RNA folding energy landscape using scalable distributed cyberinfrastructure," in *Emerging Computational Methods in the Life Sciences, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 477–488.

[184] B. K. Radak, M. Romanus, T.-S. Lee, H. Chen, M. Huang, A. Treikalis, V. Balasubramanian, S. Jha, and D. M. York, "Characterization of the Three-Dimensional Free Energy Manifold for the Uracil Ribonucleoside from Asynchronous Replica Exchange Simulations," *Journal of Chemical Theory and Computation*, vol. 11, no. 2, pp. 373–377, 2015, http://dx.doi.org/10.1021/ct500776j. [Online]. Available: http://dx.doi.org/10.1021/ct500776j

[185] National Energy Research Scientific Computing Center (NERSC), https://www.nersc.gov/, accessed: 2015-11-5.

[186] Oak Ridge National Laboratory (OLCF), https://www.olcf.ornl.gov/, accessed: 2015-11-5.

[187] National Centre for Supercompunting Applications (NCSA), http://www.ncsa.illinois.edu/, accessed: 2015-11-5.

[188] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[189] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[190] Architectural Languages Today, University of L'Aquila, http://www.di.univaq.it/malavolta/al/, accessed: 2015-11-5.

[191] Unified Modeling Language (UML), OMG, http://www.uml.org/, accessed: 2015-11-5.

[192] N. Medvidovic and D. S. Rosenblum, "Assessing the suitability of a standard design method for modeling software architectures," in *Software Architecture*. Springer, 1999, pp. 161–182.

[193] J. Horwitz and B. Lynn, "Toward hierarchical identity-based encryption," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2002, pp. 466–481.

[194] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Würthwein, "The pilot way to grid resources using glideinWMS," in *Proceedings of the World Congress on Computer Science and Information Engineering*, vol. 2. IEEE, 2009, pp. 428–432.

[195] G. Juve, E. Deelman, K. Vahi, and G. Mehta, "Experiences with resource provisioning for scientific workflows using Corral," *Scientific Programming*, vol. 18, no. 2, pp. 77–92, 2010.

[196] M. Santcroos, S. D. Olabarriaga, D. S. Katz, and S. Jha, "Pilot abstractions for compute, data, and network," *2012 IEEE 8th International Conference on E-Science*, vol. 0, pp. 1–2, 2012.

[197] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[198] J. He, "Software-defined transport network for cloud computing," in *Proceedings of the Optical Fiber*

Communication Conference. Optical Society of America, 2013, pp. OTh1H–6.

[199] S. Camarasu-Pop, T. Glatard, J. T. Mościcki, H. Benoit-Cattin, and D. Sarrut, "Dynamic partitioning of GATE monte-carlo simulations on EGEE," *Journal of Grid Computing*, vol. 8, no. 2, pp. 241–259, 2010.

[200] Apache Hadoop, The Apache software foundation, http://hadoop.apache.org/, accessed: 2015-11-5.

[201] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[202] Apache Tez, The Apache software foundation, https://tez.apache.org/, accessed: 2015-11-5.

[203] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[204] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[205] Titan overview, OLCF, https://www.olcf.ornl.gov/titan/, accessed: 2015-11-5.

[206] S. Panitkin, D. Oleynik, K. De, A. Klimentov, A. Vaniachine, A. Petrosyan, T. Wenaus, and J. Schovancova, "Integration of panda workload management system with titan supercomputer at olcf," ATL-COM-SOFT-2015-021, Tech. Rep., 2015.

[207] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *ACM Sigmod Record*, vol. 34, no. 3, pp. 44–49, 2005.

[208] A. Barker and J. Van Hemert, "Scientific workflow: a survey and research directions," in *Parallel Processing and Applied Mathematics*. Springer, 2008, pp. 746–753.

[209] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, vol. 2. IEEE, 2005, pp. 759–767.