A Fresh Perspective on Pilot-Jobs

ABSTRACT

There is no agreed upon definition of Pilot-Jobs; however a functional attribute of Pilot-Jobs that is generally agreed upon is they are tools/services that support multi-level and/or application-level scheduling by providing a scheduling overlay on top of the system-provided schedulers. Nearly everything else is either specific to an implementation, open to interpretation or not agreed upon. For example, are Pilot-Jobs part of the application space, or part of the services provided by an infrastructure? We will see that close-formed answers to questions such as whether Pilot-Jobs are system-level or application-level capabilities are likely to be elusive. Hence, this paper does not make an attempt to provide close-formed answers, but aims to provide appropriate context, insight and analysis of a large number of Pilot-Jobs, and thereby bring about a hitherto missing consilience in the community's appreciation of Pilot-Jobs. Specifically this paper aims to provide a comprehensive survey of Pilot-Jobs, or more generically of Pilot-Job like capabilities. A primary motivation for this work stems from our experience when looking for an interoperable, extensible and general-purpose Pilot-Job; in the process, we realized that such a capability did not exist. The situation was however even more unsatisfactory: in fact there was no agreed upon definition or conceptual framework of Pilot-Jobs. To substantiate these points of view, we begin by sampling (as opposed to a comprehensive survey) some existing Pilot-Jobs and the different aspects of these Pilot-Jobs, such as the applications scenarios that they have been used and how they have been used. The limited but sufficient sampling highlights the variation, and also provides both a motivation and the basis for developing an implementation agnostic terminology and vocabulary to understand Pilot-Jobs; Section §3 attempts to survey the landscape/eco-system of Pilot-Jobs. With an agreed common framework/vocabulary to discuss and describe Pilot-Jobs, we proceed to analyze the most commonly utilized Pilot-Jobs and in the process provide a comprehensive survey of Pilot-Jobs, insight into their implementations, the infrastructure that they work on, the applications and application execution modes they support, and a frank assessment of their strengths and limitations. An inconvenient but important question – both technically and from a sustainability perspective that must be asked: why are there so many similar seeming, but partial and slightly differing implementations of Pilot-Jobs, yet with very limited interoperability amongst them? Examining the reasons for this state-of-affairs provides a simple yet illustrative case-study to understand the state of the art and science of tools, services and middleware development. Beyond the motivation to understand the current landscape of Pilot-Jobs from both a technical and a historical perspective, we believe a survey of Pilot-Jobs is a useful and timely undertaking as it provides interesting insight into understanding issues of software sustainability.

HPDC'13 2013, New York, USA

1. INTRODUCTION

The seamless uptake of distributed infrastructures by scientific applications has been limited by the lack of pervasive and simple-to-use abstractions at multiple levels – at the development, deployment and execution stages. Of all the abstractions proposed to support effective distributed resource utilization, a survey of actual usage suggested that Pilot-Jobs were arguably one of the most widely-used distributed computing abstractions – as measured by the number and types of applications that use them, as well as the number of production distributed cyberinfrastructures that support them.

The fundamental reason for the success of the Pilot-Job abstraction is that Pilot-Jobs liberate applications/users from the challenging requirement of mapping specific tasks onto explicit heterogeneous and dynamic resource pools. In other words, at least in part, due to the decoupling between task/workload specification and task management. Pilot-Jobs also improve the efficiency of task assignment and shield applications from having to load-balance tasks across such resources. Another concern often addressed by Pilot-Jobs is fault tolerance which commonly refers the ability of the Pilot-Job system to verify the execution environment before executing jobs. The Pilot-Job abstraction is also a promising route to address specific requirements of distributed scientific applications, such as coupled-execution and application-level scheduling [21, 20].

A variety of PJ frameworks have emerged: Condor-G/ Glide-in [18], Swift [43], DIANE [25], DIRAC [12], PanDA [13], ToPoS [40], Nimrod/G [11], Falkon [32] and MyCluster [42] to name a few. Although they are all, for

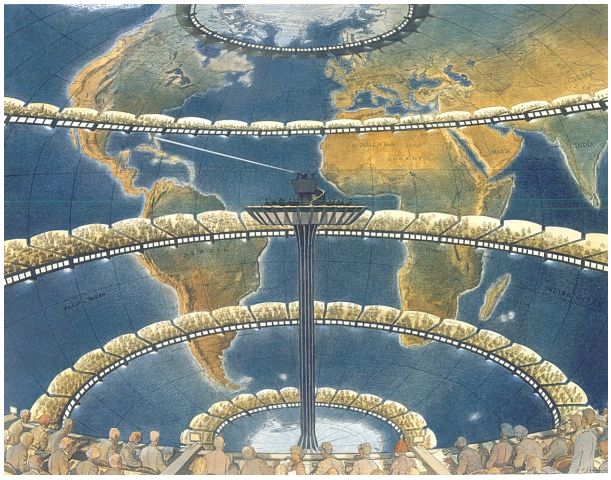

Figure 1: *Forecast Factory* as envisioned by Lewis Fry Richardson. Drawing by François Schuiten.

the most part, functionally equivalent – they support the decoupling of workload submission from resource assignment – it is often impossible to use them interoperably or even just to compare them functionally or qualitatively. The situation is reminiscent of the proliferation of functionally similar yet incompatible workflow systems, where in spite of significant a posteriori effort on workflow system extensibility and interoperability (thus providing post-facto justification of its needs), these objectives remain difficult if not infeasible.

The remainder of this paper is as follows yadayadayada. In section 2 we go back in time and look at how the concept of Pilot-Jobs has evolved by dissecting existing Pilot-Job systems and systems with pilot-like characteristics.

2. FUNCTIONAL EVOLUTION OF PILOT-JOBS

When Lewis Fry Richardson in 1922 devised his *Forecast Factory* (Figure 1) it might have been the first mentioning of large scale parallel computing. In determining the necessary processing power for weather forecasting, he estimated that 64000 *computers* (human beings in this case) would be required for solving the equations. These *computers* would all be assigned a part of the globe by a central *senior clerk*. The *computers* would perform their calculations and the results would be collected by the clerk. This was in effect a Master-Worker pattern [38].

As established in the introduction, Pilot-Jobs have proven to be an effective tool for task parallelism. (Note that the name *Pilot-Job* was not used before X, and was introduced by Y.) One common use for the Pilot-Job paradigm is the afore mentioned Master-Worker (M-W) scheme and it's associated frameworks.

In the context of distributed systems, the M-W scheme was initially used for farming tasks from a master to a various number of workers, and could easily be adapted to run in a platform-independent way across the potentially heterogeneous resources [19, 30]. M-W based frameworks could respond to the dynamically changing resources by adapting the number of workers to match the resource availability.

As the resources in distributed computing infrastructures adopted more and more batch queuing systems, users were

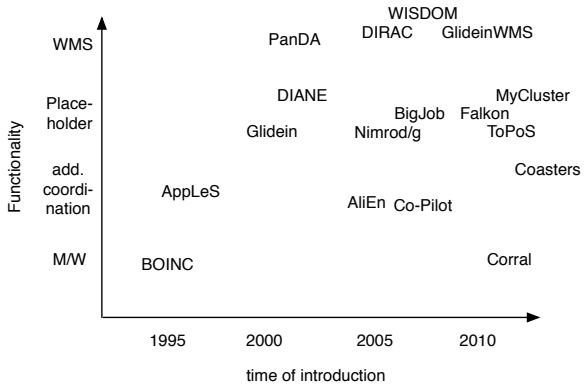

Figure 2: Introduction of terms and systems over time.

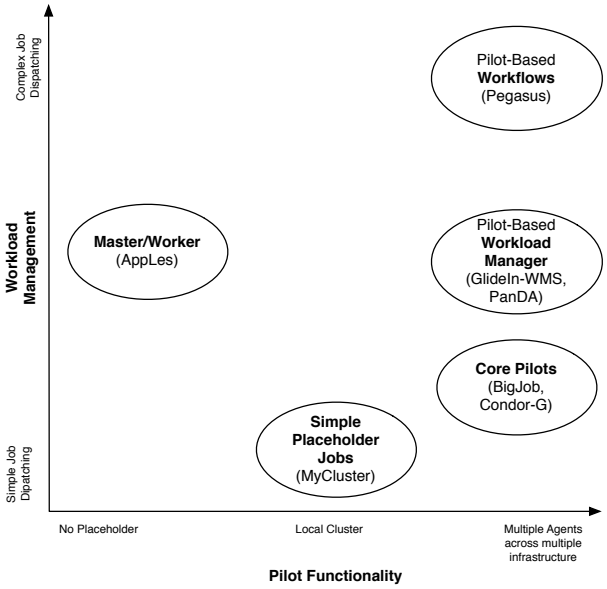

Figure 3: Pilot-Job Clustering

forced to submit their jobs individually to a scheduler. Often, the type of scheduler on a one machine was different than that of another machine. There was a need for managing the heterogenous, dynamic grid environments, especially in terms of dynamic scheduling.

This drove the creation of AppLeS [9], a framework for application-level scheduling. For this purpose, AppLeS provides an agent that can be embedded into an application enabling the application to acquire resources (e. g. via Globus, SSH or Legion) and efficiently schedule tasks onto these. Besides M-W, AppLeS provides also different application templates, e. g. for parameter sweep and moldable parallel applications [8].

Although systems such as AppLeS, gave user-level control of scheduling, initial queue reservation time still was an issue. This brought about the idea of placeholder scheduling [31]. A placeholder was an early Pilot mechanism in that it was an abstraction layer above the various batch queuing systems available on different resources. It held a *place* in the regular batch queue, and when it became active,

it could pull tasks to execute. Placeholder scheduling was advantageous in that it did not require any special superuser privileges on the machines, which was something most grid users did not have access to. It also provided a means of load balancing across the different resources. As placeholder scheduling evolved, it came to include dynamic monitoring and throttling of the different placeholders based on the queue times on the machines.

Around the same time as AppLeS was introduced, volunteer computing projects started using the M-W approach to achieve high-throughput calculations for a wide range of problems. These volunteer resources could be distributed around the world, users merely have to download a client program, and it runs in the background on their computer. The volunteer workers were essentially heterogeneous and the pool of workers more dynamic, as opposed to essentially homogeneous and static in AppLeS. The idea of farming out tasks in a distributed environment including personal computers was powerful in that it essentially made a computing grid out of many less powerful machines.

The first public volunteer computing project was The Great Internet Mersenne Prime Search effort[44], shortly followed by distributed.net [22] in 1997 to attack the RC5-56 secret-key challenge. The third large volunteer computing project was the SETI@Home project, which set out to analyze radio telescope data. Out of this last project, the generic BOINC distributed master-worker framework grow [5]. Applications can be built on top of BOINC for their own scientific endeavors.

While in the AppLes and placeholder job scenarios the resources were planned from a central place, the resources in the volunteer computing were more of an ad-hoc nature. The central master would in the latter case not know which resources could come available at any given time, and therefore there was no advance allocation of work to a specific worker. This in contrast with when there is an orchestrated set of resources requested by the master, in that case the master can for example equally divide the tasks over the allocated masters.

Condor is a high-throughput distributed batch computing system. Originally Condor was made for systems within one administrative domain. With Flocking [17], it was possible to group multiple Condor systems (pools) together so that there resources could be used in an aggregative manner. However, this mode of operation was limited to system space by the owners of the individual Condor systems and could done on application level. Condor-G/Glide-in [18] is one of the pioneers of the Pilot-Job concept. Glide-in is a mechanism by which a user can add remote grid resources to the local condor pool and run their jobs on the added resource the same way that all condor jobs are submitted. On the remote resource a set of Condor daemons is started, which then registers the available job slots with the central Condor pool. The resources added are available only for the user who added the resource to the pool, thus giving complete control over the resources for managing jobs without any queue waiting time. Glide-in installs and executes necessary Condor daemons and configuration on the remote resource, such that the resource reports to and joins the local Condor pool. Various systems that built on the Pilot capabilities of Condor-G/Glide-in have been developed, e. g. Bosco [28].

While core PJ systems mainly focus on providing simple Pilot capabilities commonly in application space, many of these systems evolved towards or were extended to Pilot-based workload managers. For example, various systems that built on the Pilot capabilities have been developed, e. g. GlideinWMS and GlideCAF [7], and PanDa [45]. These higher-level systems which are often centrally hosted, move critical functionality from the client to the server (i.e. a service model). These systems usually deploy Pilot factories that automatically start new Pilots on demand and integrate security mechanisms to support multiple users simultaneously.

Several of these have been developed in the context of the LHC experiment at CERN, which is associated with a major increase in the uptake and availability of Pilots, e. g. GlideInWMS, DIRAC [12], PanDa [45], AliEn [6] and Co-Pilot [1]. Each of these Pilots serves a particular user community and experiment. Interestingly, we observe that these Pilots are functionaly very similar, work on almost the same underlying infrastructure and serve applications with very similar (if not identical) characteristics.

GlideinWMS [37] is a higher-level workload management system that is based on the Pilot capabilities of Condor-G/Glide-in. The system can, based on the current and expected number of jobs in the pool, automatically increase or decrease the number of active Glide-ins (Pilots) available to the pool. GlideinWMS is a multi-user Pilot-Job system commonly deployed as a hosted service. In contrast to low-level Pilot-Job systems, GlideinWMS attempts to hide rather than expose the Pilot capabilities from the user. GlideinWMS is currently deployed in production on the Open Science Grid (OSG) [29] and is the recommended mode for accessing OSG resources.

In comparison with the generic functionality of GlideinWMS, PanDA [45] workload management systemsis much more specific to its application/community. However it has been extended to use Condor-G/Glidein, in addition to its native Pilot system. Furthemore, in contrast with a Condor-G only approach, PanDA can utilize multiple queues; each Pilot is assigned to a certain PanDA internal queue. PanDA also provides the ability to manage data associated the jobs managed by the PanDA workload manager.

In addition to processing, AliEn [6] also provides the ability to tightly integrate storage and compute resources and is also able to manage file replicas. While all data can be accessed from anywhere, the scheduler is aware of data localities and attempts to schedule compute close to the data. In contrast with GlideinWMS and PanDA, AliEn deploys a pull-based model [36].

DIRAC [12] is another comprehensive workload management system built on top of Pilots. As PanDA and AliEn it supports the management of data, which can be placed in different kinds of storage elements (e. g. based on SRM).

Another interesting Pilot that is used in the LHC context is Co-Pilot [1]. Co-Pilot serves as integration point between different grid Pilot-Job systems (such as AliEn and PanDA) and clouds. Co-Pilot is based more on the actual submission of jobs and is limited in its user-level controllability and allowance of application-level programming.

In addition to the Pilot-Job systems developed around the LHC experiment, several other systems emerged. GWPilot [34] is a Pilot systems that is based on the GridWay metascheduler. GWPilot in particular, emphasizes its multi-user support and the support for standards, such as DRMAA,

OGSA-BES and JSDL.

In the context of scientific workflows, Pilot-Job systems have proven an effective tool for managing the workload. For the Pegasus project, the Corral system [35] was developed as a front-end to GlideinWMS. In contrast to GlideinWMS, Corral provides more explicit control over the placement and start of Pilots to the end-user. Corral has been developed to support the requirements of the Pegasus workflow system in particular to optimize the placements of Pilots with respect to their workload.

SWIFT [43] is a scripting language designed for expressing abstract workflows and computations. The language provides among many things capabilities for executing external application as well as the implicit management of data flows between application tasks. The runtime environment handles the allocation of resources and the spawning of the compute tasks. Both data- and execution management capabilities are provided via abstract interfaces.

The Coaster system [14] has been developed to address the workload management requirements of Swift supporting various infrastructures, such as both cloud and grid. Using the Coaster service, one executes a Coaster Master on a head node, and the Coaster workers run on compute nodes to execute jobs. In the case of cloud computing, Coasters offers a zero-install feature in which it deploys itself and installs itself from the head node and onto the virtual machines without needing any prior installation on the machine. Coaster relies on a master/worker coordination model. SWIFT supports various scheduling mechanisms on top of Coaster, e. g. a FIFO and a load-aware scheduler.

Further, SWIFT can be used in conjunction with other Pilot systems, e. g. Falkon [32]. Falkon refers to pilots as the so called provisioner, which are created using the Globus GRAM service. The provisioner spawns a set of executor processes on the allocated resources, which are then responsible for managing the execution of task. Tasks are submitted via a so called dispatcher service. Falkon also utilizes a master-work coordination model, i. e. the executors periodically query the dispatcher for new tasks. Falkon was specifically engineered for many small tasks and shows high performance compared to native queuing systems.

WISDOM [4, 10] is an application-centric environment for supporting drug discovery. The architecture utilizes an agent run as a Grid job to pull tasks from a central metadata service referred to as AMGA.

3. UNDERSTANDING THE LANDSCAPE: DEVELOPING A VOCABULARY

The overview presented in §2 shows a degree of heterogeneity both in the functionalities and the vocabulary adopted by different Pilot-Job systems. Implementation details sometimes hide the functional commonalities and differences among Pilot-Jobs systems while features and capabilities tend to be named inconsistently, often with the same terms referring to multiple concepts or the same concept named in different ways.

This section offers an analysis of the architectural components, functionalities, and terminology shared by every Pilot-Job system. The goal is to offer both a conceptual description of an exemplar Pilot-Job system and a well- defined vocabulary. Both will be leveraged in §4 and §5 to produce a comparative and critical analysis of diverse Pilot-Job frameworks.

3.1 Core Functionalities of Pilot-Jobs

All the Pilot-Job systems introduced in §2 are engineered to allow for the execution of multiple types of workloads on Distributed Computing Infrastructures (DCIs) such as, Grids, Clouds or HPC facilities. This is achieved differently, depending on use cases, design and implementation choices, but also on the constraints imposed by the peculiarities of the targeted DCI. Finding common denominators among these systems requires an analysis along multiple dimensions.

Each Pilot-Job system exhibits architectural, provisioning and execution characteristics that are integral to the Pilot-Job paradigm itself. However, many Pilot-Job systems also exhibit specific or non-integral characteristics. Discerning and distinguishing characteristics into these categories, requires isolating and defining a minimal set of concepts and terms that have to apply to every Pilot-Job implementation.

At some level, all Pilot-Job systems introduced in §2 leverage a similar architecture to accomplish workload execution. Such an architecture is based on two separate, but intercommunicating logical components: a **Workload Manager** and a **Task Executor**. The Workload Manager is related to the management of the workloads and their tasks; the Task Executor to the execution of the tasks on target resources. As will be shown in §4, the implementations of both components significantly vary from system to system, with one or more elements responsible for specific functionalities both on application as well as infrastructure level. Nevertheless, the two logical components can be consistently distinguished and isolated across different Pilot-Job systems.

For example, looking at the core Pilot-Job systems introduced in §**??**, the scheduler and dispatcher of Nimrod-G belong to the logical component dedicated to workload management, while the job-wrapper belongs to the component responsible for task execution. Similar examples can be given for the advanced Pilot-Job systems and for Pilot-based frameworks: GlideinWMS offers an extended set of features when compared to the scheduler and dispatcher of Nimrod-G, but all those features belong to the same, well-defined and self-contained logical component — i.e. the one in charge of managing the workloads. The same is valid for Corral and SWIFT: both offer evolved functionalities and interfaces to manage workloads but, as such, both are part of an architecture where workload management and task execution belong to two distinct components.

The Workload Manager and the Task Executor are separate and well-defined logical components that communicate and coordinate in order to exchange tasks, input and output files, and related data. Communication and coordination are two fundamental characteristics of every distributed architecture , but we posit that it is not an integral logical component of Pilot-Job systems, nor is any specific communication and coordination pattern a defining feature of the Pilot-Job paradigm.

For example, as seen in §**??**, the Master-Worker paradigm is very common among Pilot-Jobs. Functionally, the Master can be identified with the Workload Manager, while the Worker with the Task Executor. However, the Master-Worker paradigm usually implies a specific distribution of capabilities between Master and Worker, and, in some cases, also a communication model. Moreover, Master-Worker is

applied to a non-fixed set of capabilities while referring not only to architectures but also to frameworks and applications. This flexibility makes Master-Worker a viable choice for implementing Pilot-Jobs but not one of its defining characteristics.

The described logical components of Pilot-Job systems support flexible execution strategies along with improved performance in the task submission process when compared to scheduling tasks directly on DCIs. The tasks of a workload are bound to one or more Task Executor and, as such, to a pilot, without having to be directly scheduled on the DCI's job management system. Furthermore, the binding of tasks to a pilot might happen before or after the pilot has been assigned a portion of the DCI's resources, and it can last for the whole time the pilot can hold on its resources. This execution strategy requires dedicated functionalities both for performing the binding of tasks to pilots but also for the provisioning of pilots so that they can be assigned portions of the DCI's resources.

The minimal set of functionalities that needs to be implemented by a Pilot-Job system involves **Pilot provisioning** and **task dispatching**. Pilot provisioning is a core functionality of every Pilot system because it is essential for the creation of resource overlays. This type of overlay allows for tasks to utilize resources without directly depending on the capabilities exposed by the targeted DCI. Pilots are scheduled to the DCI resources by means of the DCI capabilities but, once scheduled and then run, pilots make those resources directly available for the execution of the tasks of a workload. Thanks to resource overlays, Pilot systems can also implement functionalities tailored to the requirements of specific user communities [cit, cit, cit].

The procedures and mechanisms to provision pilots depend on the capabilities exposed by the targeted DCI and on the implementation of the considered Pilot system. Typically, for a DCI adopting queue, batch and scheduling systems, provisioning a pilot involves it being submitted as a job. Conversely, for infrastructures that do not adopt a job-based approach, a pilot would be executed by means of other types of logical container as, for example, the one implemented by means of a Virtual Machine (VM) on so called Infrastructures as a Service (IaaS).

Once pilots are bound to DCI resources, tasks need to be dispatched to those pilots for execution. Thanks to resource overlays, task dispatching does not depend on other functionalities provided by the DCI and can be implemented within the boundaries of the Pilot systems. This independence allows to shift the control of tasks of a workload directly and exclusively to the Pilot system, a distinguishing characteristic of such systems.

Data management may have an important role within Pilot systems. For example, functionalities can be provided to support data staging for task execution or for managing task-related data according to the capabilities offered by specific DCI. Nonetheless, Pilot systems can be devised in which tasks do not require data management because they (i) do not necessitate input files, (ii) do not produce output files, (iii) data is already locally available or (iv) data management is left to the application itself. As such, data management should not be considered a necessary functionality of Pilot systems even when present in many of them.

In the following subsection, a minimal set of terms related to the capabilities just described is defined.

3.2 Terms and Definitions

The name 'Pilot-Job' indicates the primary role played by the concepts of 'pilot' and 'job' in this type of system. The definition of both concepts is context-dependent and several other terms needs to be clarified in order to offer a coherent terminology. Both 'job' and 'pilot' needs to be understood in the context of DCIs, the infrastructures where Pilot-Jobs systems are provisioned. DCIs offer compute, storage, and network resources and Pilot-Jobs allow for the users to utilize those resources to execute the tasks of a workload.

Application. Responsible for the definition of the workload and consumer of the Pilot-Job system.

Workload. A set of tasks, possibly correlated, that are instrumental to the application to achieve its goal.

Task. A set of operations specified by the application, that are encoded into one or more programs to be executed on a DCI.

Resource. Finite, typed and physical quantity utilized when executing one or more workloads. Compute cores, data storage space, or bandwidth are all examples of resources commonly utilized by running workloads.

Infrastructure. Structured set of resources, possibly geographically and institutionally separated from the users utilizing those resources to execute one or more workloads. [cit, cit] Infrastructures can be logically partitioned, with a direct or indirect mapping onto individual pools of hardware. Commonly, Infrastructure can be used also to indicate a DCI.

As seen in §**??**, most of the DCIs where Pilot-Jobs systems are executed utilize 'queues', 'batch systems' and 'schedulers'. In such DCIs, jobs are scheduled and then executed by a batch system.

Job. A container for one or more programs, a description of their properties and indications on how they should be executed.

In the context of a Pilot-Job system, jobs and tasks are functionally analogous but qualitatively different; it is therefore relevant to highlight their distinction. Functionally, both jobs and tasks are containers — i.e. a set of programs with metadata but the term 'task' is used when reasoning about workloads while 'job' is used in relation to a specific type of infrastructure where such a container can be executed. Accordingly, tasks are considered as the functional units of a workload, while jobs as a way to execute programs on a given infrastructure. It should be noted that the two terms can be used interchangeably when considered outside the context of Pilot-Job systems. Workloads are encoded into jobs when they have to be directly executed on infrastructures that support or require that type of container.

The capabilities exposed by the job submission system of the target infrastructure determine the submission process of pilots: pilots are programs with specific capabilities and are submitted as jobs on the type of DCIs just described. In such a context, schedulers, batch processing and queuing define the practical boundaries for how and when the pilots can be provisioned on a given DCI.

Terms	Functionality	Logical Component
Workload	Task Dispatching	Workload Manager
Task	Task Dispatching	Task Executor
Resource	Pilot Provisioning	Workload Manager
Infrastructure	Pilot Provisioning	Workload Manager
Job	Pilot Provisioning	Workload Manager
Pilot	Pilot Provisioning	Workload Manager
Multi-level scheduling	Task Dispatching	Task Executor
Early binding	Pilot Provisioning	Workload Manager
Late binding	Pilot Provisioning	Workload Manager

Table 1: Mapping of the core terminology of Pilot systems into the functionalities and logical components described in §3.1.

Showing that the way in which a pilot is provisioned depends on the capabilities exposed by the target DCI illustrates the limits of choosing the term 'job'. The use of that term is due to a historical contingency, viz., the targeting of a specific class of DCIs in which the term 'job' was — and still is — meaningful. Nonetheless, with the development of new types of DCI, the term 'job' has become too restrictive, a situation that can lead to terminological and conceptual confusion, and to the use of synonyms for both the terms 'pilot' and 'pilot-job'.

The term 'pilot' is often associated with that of 'placeholder' so to emphasize the two distinctive capabilities that every pilot system has to implement: acquiring a set of resources by running on a DCI, and executing tasks that will utilize those resources. A pilot is a 'placeholder' because it holds portion of the DCI resources for a user or a group of users, depending on implementation details, to gain exclusive control over the binding and execution of a workload.

Pilot. A resource placeholder running on a given infrastructure and capable of executing pushed or pulled tasks while managing data.

From a terminology point of view, the term 'pilot' as defined here is named differently across multiple Pilot-Job systems. Depending upon context, in addition to the term 'placeholder', pilot is also named 'agent' and, in some cases, 'Pilot-Job' [cit]. All these terms should be considered synonyms and their use to indicate the same concept is a clear indication that a minimal and consistent vocabulary is needed when reasoning about multiple Pilot-Job systems.

Once one or more pilots become available through a Pilot-Job system on the targeted infrastructures, users can start to execute their tasks without having to deal with the job submission system of that infrastructure. How tasks are assigned to pilots is a matter of implementation. For example, a dedicated scheduler could be adopted, or tasks might be directly assigned to a pilot by the user.

The simplification obtained by circumventing the job submission system of the DCI is one of the main reasons for the success of the Pilot-Job systems. The overhead and lack of control imposed by a centralized job management system shared among multiple users are bypassed, allowing for a faster and more reliable execution of workloads.

The Pilot-Job systems are said to implement multi-level scheduling because the assignment of resources to tasks happens in at least two stages. A portion of the resources of an infrastructure are first bound to a pilot and then to one or more tasks for consumption. This is an important feature of Pilot-Job systems because it allows for a task to be bound to a pilot before it is in turn bound to the resources.

The binding of tasks to pilots depends on the state of the pilot. A pilot is inactive until it is executed on a DCI, active after that. Early binding indicates the binding of a task to an inactive pilot; late binding the binding of a task to an active pilot. Early binding is useful to increase the flexibility with which pilots are deployed. By knowing in advance the properties of the tasks that are bound to a pilot, specific deployment decisions can be made for that pilot. Late binding is critical in assuring high throughput by allowing for tasks to be executed without waiting in a queue or waiting for a specific container - for example a job or a VM - to be instantiated.

Binding pilots to their resources and tasks to pilots depends on placement choices. As such, the binding process in itself is often an instance of scheduling. Pilot systems implement multi-level scheduling because they require the scheduling of two types of entities - pilots and tasks - in a distinct chronological order.

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

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

Multi-level scheduling. Scheduling pilots onto resources and tasks onto active or inactive pilots in distinct chronological order.

Depending on the specific capabilities implemented in the workload management component, some Pilot-Job systems allow for pilots to be specified by taking into consideration the properties of the task of a workload. This type of specification process should not be confused with early binding as the latter requires for a pilot to have been already specified but not yet bound to a resource.

4. PILOT-JOB SYSTEMS IMPLEMENTATIONS

Section §3 offered two main contributions: A minimal description of the logical components and the functionalities of Pilot-Job systems, and a well-defined core terminology to support reasoning about such systems. The former sets the necessary and sufficient requirements for a distributed system to be a Pilot-Job system, while the latter enables consistency when referring to different Pilot-Job systems. Both these contributions are leveraged in this Section in order to review critically a set of relevant Pilot-Job implementations.

Core Propriety	Functionality
Pilot Characteristics	Pilot Provisioning
Resource Dependences	Pilot Provisioning
Workload Semantics	Task Dispatching
Task Binding Characteristics	Task Dispatching
Deployment Strategies	Task Execution

Table 2: Mapping of the Core Properties of Pilot-Job system implementations into the functionalities described in §3.1.

The goal of this Section is twofold. Initially, the set of functionalities presented in §3 are used as the basis to infer a set of core implementation properties. A set of auxiliary properties is also defined when useful for a critical comparison among different Pilot-Job implementations. Subsequently, several Pilot-Job implementations are analyzed and then clustered around the properties previously defined. In this way, insight is offered about how to choose a Pilot-Job system based on functional requirements, how Pilot-Job systems are designed and engineered, and the theoretical properties that underly such systems.

4.1 Core and Auxiliary Properties

This Section discusses the properties of diverse implementations of a Pilot-Job system. Two sets of properties are introduced: Core and auxiliary. *Core* properties are common to all Pilot-Job implementations while the auxiliaries characterize specific Pilot-Job systems. Therefore, auxiliary properties are not shared among all Pilot-Job implementations.

As shown in Table 2, the set of Core Properties is derived consistently with the set of functionalities presented in §3 - Pilot Provisioning, Task Dispatching, and Task Execution. As such, Core Properties are both necessary and sufficient for an implementation of a distributed system to be classified as a Pilot-Job. On the contrary, auxiliary properties are not defining of a Pilot-Job system but they are implementation details that further characterize a Pilot-Job system. The set of auxillary properties we discuss is not closed, i.e., the complete set of auxillary properties includes, but is not limited to the set of auxillary properties we discuss.

Implementations of Pilot Provisioning are analyzed by focusing on two specific properties: the requirements and dependencies between the Pilot-Job system and the underlying distributed resources; and the type of resources the pilot system exposes in terms of computing, data and networking. For example, in order to schedule a pilot onto a specific resource, the Pilot-Job system will need to know what type of container to use - e.g. pilot, virtual machine, what type of scheduler the resource exposes, but also what kind of functionalities will be available on the nodes of the resource in terms of compute, data and networking.

Two properties are also used to analyze the implementations of Task Dispatching: The semantics of workloads, and how the tasks of a given workload can be bound to single or multiple pilots. Semantically, a workload description contains all the information necessary for it to be dispatched to the appropriate resource. For example, information related to both space and time should be available when deciding how many resources of a specific type should be used to execute the given workload but also for how long such resources should be available. Executing a workload requires for its tasks to be bound to the resources. Both the temporal and spatial dimensions of the binding operations are relevant for the implementation of Task Dispatching. Depending on the concurrency of a given workload, tasks could be dispatched to one or more pilots for an efficient execution. Furthermore, tasks could be bound to pilots before or after its instantiation, depending on resource availability and scheduling decisions.

Finally, implementations of Task Execution are analyzed by reviewing different strategies for task scheduling and by describing how Pilot-Job systems support task execution. Pilot-Job implementations may offer multiple scheduling strategies depending on varying factors related to the nature of the workload, the state of the resources, or the capabilities exposed by the underlying middleware. Furthermore, Pilot-Job systems often are responsible for setting up the execution environment for the tasks of the given workload. While each task can be seen as a self-contained and self-sufficient unit with a kernel ready to be executed on the underlying architecture, often tasks require their environment to be set up so that some libraries, data or accessory programs are made available.

Several auxiliary properties play a fundamental role in distinguishing among Pilot-Job systems implementations, as well as address, set and provide constraints on their usability. Programming and user interfaces; interoperability across differing middleware and other Pilot-Job systems; multitenancy; strategies and abstractions for data management; security including policies alongside authentication and authorization; support for multiple usage modes like HPC or HTC; or robustness in terms of fault-tolerance and high-availability; are all examples of properties that might characterize a Pilot-Job implementation but in of themselves, would not distinguish a Pilot-Job as a unique system.

Both core and auxiliary properties have a direct impact on the multiple use cases for which Pilot-Job systems are currently engineered and deployed. For example, while every Pilot-Job system offers the opportunity to schedule the tasks of a workload on a pilot, the degree of support of specific workloads varies vastly across implementations. Furthermore, some Pilot-Job systems support Virtual Organizations and running tasks from multiple users on a single pilot while others support jobs leveraging a Message Passing Interface (MPI). Analogously, every Pilot-Job systems, support the execution of one or more type of workload but they differ when considering execution modalities that maximize throughput (HTC), computing (HPC) or container-based high scalability (Cloud).

.

4.1.1 Core properties

- **Pilot Characteristics**: In §**??** pilots have been defined as placeholders for resources. As such, the characteristics of each pilot depends upon the type and capabilities of the resource it exposes. Pilots usually expose computing resources but, depending on the capabilities offered by the infrastructure where the pilot is instantiated, they might also expose data and network capabilities. Within the domain of each type of resource and infrastructure, some of the typical characteristics of pilots are: Size (e.g. number of cores), lifespan, intercommunication (e.g. low-latency or inter-domain), computing platforms (e.g. x86, Cray,

or CUDA), file systems (e.g. local, shared, or distributed).

- **Resource Dependences**: The provisioning of pilots depends on how the Pilot-Job system interfaces with one or more targeted infrastructure(s). In this context, the degree of coupling between the Pilot-Job system and the infrastructure can vary, depending on how the Pilot-Job system is deployed, how much it is integrated with the middleware used within the targeted infrastructure, whether the Pilot-Job system is interoperable across multiple types of middleware, and how much information the Pilot-Job system needs about the states and capabilities of the targeted infrastructure.
- **Coordination and Communication**.
- **Workload Semantics**: The tasks of a workload are dispatched to pilots depending on their semantics. Specifically, dispatching decisions depends on the relationship held by a task with the other tasks belonging to the workload, the affinity they require between data and compute resources, and the type of capabilities they require in order to be executed. Pilot-Job implementations support a varying degree of semantic richness for the workload and its tasks.
- **Task Binding Characteristics**: One of the core functionalities implied by Task Dispatching is the binding of tasks to pilots. Without such capability, it would not be possible to know where to dispatch tasks, pilots could not be used to run tasks and, as such, the whole Pilot-Job system would not be usable. As seen in §3, Pilot-Job systems may allow for two main types of binding between tasks and pilots: early binding and late binding. Pilot-Job implementations differ in whether and how they support these two types of binding. Specifically, while there might be implementations that only support a single type of binding behavior, they might also differ in whether they allow for the users to control directly what type of binding is performed, and in whether both types of binding are available on an heterogeneous pool of resources.
- **Deployment Strategies**: Once the tasks are dispatched to a pilot, their execution may require for a specific environment to be set up. Pilot-Job implementations differ in whether and how they offer such a capability. Pilot-Job implementations may adopt dedicated components for managing execution environments, or they may relay an ad hoc configuration of the pilots. Furthermore, execution environments can be of varying complexity, depending on whether the Pilot-Job implementation allows for data retrieval, dedicated software and library installations, communication and coordination among multiple execution environment and, in case, pilots.

4.1.2 Auxiliary properties

- **Architecture**: Pilot-Job systems may be implemented by means of different type of architectures (e.g service-oriented, client-server, or peer- to-peer). For example, it is conceivable that architectural choices influence if not preclude certain deployment strategies. The analysis and comparison of architectural choices is limited to the trade-offs implied by such a choice, especially when considering how they affect the Core Properties listed in the previous Subsection.
- **Interfaces**: The implementations of Pilot-Job systems may present several types of interfaces. For example, there are interface considerations between the main components of the Pilot-Job system, between the application and pilot layer, between end users and the Pilot-Job system, and for one or more programming languages.
- **Interoperability**: Two types of interoperability are relevant when analyzing different Pilot-Job implementations: Interoperability with multiple type of resources and interoperability across diverse Pilot-Job systems. The former allows for the Pilot-Job to provision pilots and execute workloads on different type of resources and systems (e.g. HTC, HPC, Cloud but also Condor, LSF, Slurm or Torque), while the latter becomes relevant when considering a landscape where multiple Pilot-Job implementations are available with diversified characteristics and capabilities.
- **Multitenancy**: Pilot-Job system may offer multitenancy at both system and local level. When offered at system level, multiple users are allow to utilize the same instance of a Pilot-Job system, while when available at local level, multiple users may use the same pilot instance or any other component implemented within the Pilot-Job system.
- **Robustness**: Used to identify those properties that contribute towards the resilience and the reliability of a Pilot-Job implementation. In this Section, the analysis focuses on fault-tolerance, high-availability and state persistence. These properties are considered indicators of both the maturity of the development stage of the Pilot-Job implementation, and the type of support offered to the paradigmatic use cases introduced in Section 2.
- **Security**: While the properties of Pilot-Job implementations related to security would require a dedicated analysis, we limit the discussion to authentication, authorization and policies. The scope of the analysis is further constrained by focusing the analysis only on those elements of these properties that impact the Core Functionalities as defined in §3.
- **Usage Modes**: As shown in §2 and 3, Pilot-Job systems support many different use cases involving diversified usage modes - e.g. HPC, HTC and Cloud.

4.2 Analysis of Pilot-Job Implementations

4.2.1 AppLeS

Rationale.

AppLeS is an ideal introductory software package to be studied as it is a *pre-Pilot* framework; it is capable of addressing some of the issues Pilot-Jobs are designed to handle, but lacks some fundamental components which limit its usability compared to the full Pilot-Job approach.

Pilot-Job System	Pilot (Resource) Provisioning	Job-to-Resource (Pilot) Binding	Single-/Multi-User Pilots	Classifier IV
AppLeS	N/A	Automatic (algorithmic/profiling)	Single (?)	–
MyCluster	Manual, Explicitly User-controllable	Manual	Single (?)	–
PanDA	Automatic, Not user-controllable	Manual(?)	Single (?)	–
GlideinWMS	Automatic, Not user-controllable	Manual (rule-driven) / Automatic	Multi / Single (via Corral)	–
SWIFT/Coaster	–	–	–	–
BoSCO	Manual, Explicitly User-controllable	Manual (rule-driven) / Automatic	Multi / Single	–
BigJob	Manual, Explicitly User-controllable	Manual (rule-driven) / Automatic	Single	–

Table 3: Different pilot-job systems and their key.

Design Goals.

The main design goal of AppLeS is adaptivity, granting applications the ability to generate schedules dynamically and incorporating application performance characteristics into these schedules. The concept requires several further design goals so that changing conditions on heterogeneous resources can be adapted to, including granting applications the ability to generate schedules dynamically and incorporating application performance characteristics into these schedules.

Applications.

AppLeS has been used to choose destinations for storing satellite image files (Simple SARA), optimize the execution of data-parallel matrix calculations (Jacobi 2D AppLeS), distribute computational biology genetic searches across resources with varying response/availability times (AppLeS for Complib), and analyze biochemical interactions while dynamically recreating schedules to account for changes in performance predictions (AppLeS for MCell).

Deployment Scenarios (VO/multiuser).

The deployment characteristics of AppLeS-enabled software rely mainly upon the application writers themselves; applications may be adapted to use AppLeS which are either single or multi-user. AppLeS templates exist, which are application-class-based (e.g. parameter sweep, master-worker) software frameworks created to ease the adaptation of existing applications to make use of AppLeS.

Resource Landscape (e.g. grid/cloud/hpc/etc).

Heterogeneous grid computing resources were targeted by AppLeS, with the dynamism inherent in grid resources (availability, computational power, etc) being a driving motivator for AppLeS itself. AppLeS was first developed in 1996, precluding the development of clouds; regardless, as with "true" Pilot-Job approaches, adaptations and additions which enable cloud functionality could feasibly be built on top of the AppLeS framework as e.g. cloud middleware service interfaces. To expand the reach of AppLeS, middleware-specific extensions must be implemented.

Architecture & Interface.

AppLeS requires a tight coupling between the target application and the AppLeS code. *AppLeS templates* were created to ease this process, but the application code in essence must be modified to make use of AppLeS, as opposed to Pilot-Job systems which can generally be taken advantage of without requiring code modification of the underlying applications being executed.

The lack of a *Pilot* limits the agility of application-level scheduling, as tasks must wait in unpredictable resource manager queues before being executed, undermining the level of control. For this reason, AppLeS is not a "true" Pilot-Job system, as a *Pilot* is required to enable fundamental Pilot-Job functionality such as *multi-level scheduling*.

AppLeS Conclusion.

AppLeS shows the power of scheduling tasks across multiple computing resources via the *Master-Worker* approach which many Pilot-Jobs use as a partial foundation. However, the following "true" Pilot-Jobs will show the flexibility/power that a full Pilot-Job approach offers.

4.2.2 MyCluster

Rationale.

MyCluster was developed to allow users to submit and manage jobs across heterogeneous NSF TeraGrid resources in a uniform, on-demand manner. TeraGrid existed as a group of compute clusters connected by high-bandwidth links facilitating the movement of data, but with many different job deployment middlewares requiring cluster-specific submission scripts. MyCluster allowed all cluster resources to be submitted to via a Pilot-Job-based approach, reducing the complexity of multi-cluster submission and increasing scheduling flexibility.

Design Goals.

The authors describe their system as a "personal cluster" with the goal of achieving on-demand, user-schedulable computing resources aligning closely with what is provided by our formal definition of Pilot-Jobs. This enhancement to user control was envisioned as a means of allowing users to submit and manage thousands of jobs at once across heterogeneous distributed computing infrastructures, while providing an familiar interface for submission.

Applications.

Applications are launched via MyCluster in a "traditional" HPC manner, via a `virtual login session` which contains usual queuing commands to submit and manage jobs. This means that applications do not need to be explicitly rewritten to make use of MyCluster functionality; rather, MyCluster provides user-level Pilot-Job capabilities which users can then use to schedule their applications with.

Deployment Scenarios (VO/multiuser).

MyCluster is designed for a single-user. A MyCluster installation resides in userspace, and may be used to marshal multiple resources ranging from small local resource pools (e.g. departmental Condor or SGE systems) to large HPC installations including TeraGrid (now XSEDE).

Resource Landscape (e.g. grid/cloud/hpc/etc).

MyCluster was designed for and successfully executed on NSF TeraGrid resources, enabling large-scale cross-site submission of ensemble job submissions via its virtualized cluster interface. This approach makes the *multi-level scheduling* abilities of Pilot-Jobs explicit; rather than directly *binding* to individual TeraGrid resources, users allow the virtual grid overlay to *schedule* tasks to multiple allocated TeraGrid sites presented as a single cohesive, unified resource.

Architecture & Interface.

MyCluster implements several Pilot-Job analogues: the `Master Node Manager`, which acts as a *Pilot-Job Manager*, the `Task Manager`, which acts as an *Pilot-Job Agent*, `Job Proxies`, which act as *placeholders*, and `Slave Node Managers`, which handle *Pilot-Job Tasks* Both resource provisioning and job-to-resource *bindings* are handled manually by the end user. The end result is a complete *basic Pilot-Job system*, despite the authors of the system being constructed not having used the word "pilot" once! [42]

Additional features above and beyond the *basic Pilot-Job system* concept include: the inclusion of a MyCluster `virtual login session`, which is a commandline interface designed to allow users to interactively monitor the status of their *placeholders* and *tasks*, the automatic recovery of placeholders after loss due to exceeding wallclock limitations or node reboots; and authentication of all TCP connections used in the system via 64-bit key encryption and GSI authentication mechanisms.

MyCluster Conclusion.

MyCluster illustrates how an approach aimed at *multi-level scheduling* by marshalling multiple heterogeneous resources lends itself perfectly to a Pilot-Job-based approach. The fact that the researchers behind it formed a complete Pilot-Job system while working toward interoperability/uniform access is a testament to the usefulness of Pilot-Jobs in attacking these problems.

4.2.3 PanDA

Rationale.

PanDA was developed to provide a multi-user Pilot-Job system for ATLAS [3], which is a particle detector at the Large Hadron Collider which could handle large numbers of jobs for data-driven processing workloads. In addition to the logistics of handling large-scale job submission, there was a need for integrated monitoring for analysis of system state and a high degree of automation to reduce the need for user/administrative intervention.

Design Goals.

PanDA was designed as an advanced Pilot-Job system, satisfying requirements such as the ability to manage data, monitor job/disk space, and recover from failures. A modular approach allowing the use of plug-ins was chosen in order to incorporate additional features in the future. PanDA was designed from the ground-up to meet these requirements while scaling to handle the large scale of jobs and data produced by the ATLAS experiment.

Applications.

PanDA has been used to process data and jobs relating to ATLAS. Approximately a million jobs a day are managed [15] which handle simulation, analysis, and other work [24]. The ATLAS experiment itself produces several petabytes of data a year which must be processed and analyzed.

Deployment Scenarios (VO/multiuser).

PanDA has been initially deployed as an HTC-oriented multi-user WMS system for ATLAS, consisting of 100 heterogeneous computing sites [24]. Recent improvements to PanDA have been designed to extend the range of deployment scenarios to non-ATLAS infrastructures making PanDA a general-use Pilot-Job [26].

Resource Landscape (e.g. grid/cloud/hpc/etc).

PanDA began as a specialized Pilot-Job for the ATLAS grid, and has been extended into a generalized Pilot-Job which is capable of working across other grids as well as HPC and cloud resources. Cloud capabilities have been used as part of Helix Nebula (CloudSigma, T-Systems, ATOS), the FutureGrid and Synnefo clouds, and the commercial Google and EC2 cloud offerings as well [15], extending the reach of PanDA to cloud resources as well. Future PanDA developments seek to interoperate with HPC resources [15].

Architecture & Interface.

PanDA's *manager* is called a `PanDA server`, and matches jobs with Pilots in addition to handling data management. PanDA's *Pilot* is called, appropriately enough, a `pilot`, and handles the execution environment. These pilots are generated via PanDA's `PilotFactory`, which also monitors the status of pilots. Pilot-resource *provisioning* is handled by PanDA itself and is not user-controllable, whereas job-to-resource *binding* is handled manually by users. A central job queue allows users to submit jobs to distributed resources in a uniform manner. This basic functionality (pilot creation and management) provides PanDA with all of the baseline capabilities required for a Pilot-Job.

As PanDA is an *Advanced Pilot-Job System*, it enables functionality beyond that of a *Basic Pilot-Job*. PanDA contains some additional backend features such as `AutoPilot`, which tracks site statuses via a database, `Bamboo` which adds ATLAS database interfacing, and automatic error handling and recovery. PanDA contains support for queues in clouds, including EC2, Helix Nebula, and FutureGrid [15]. En-

hancements to the userspace include `Monitor`, for web-based monitoring, and the `PanDA client`. `PanDA Dynamic Data Placement` [24] allows for additional, automatic data management by replicating popular or backlogged input data to underutilized resources for later computations and enables jobs to be placed where the data already exists.

PanDA Conclusion.

PanDA has extended beyond its origins at the ATLAS experiment, and has also been used for other projects such as the Open Science Grid. This illustrates the importance of designing Pilot-Jobs which are easily adaptable such that Pilots are not tied to any particular DCI; in this manner, their advantages can be made available beyond their original design scope.

Mapping to section 3:

- Term 1 =
- Term 2 =

Mapping to table 1:

- Pilot (Resource) Provisioning:
- Job-to-Resource (Pilot) Binding:
- Singe-/Multi-User pilots:

4.2.4 HTCondor

HTCondor can be considered one of the most prevalent distributed computing project of all time in terms of its pervasiveness and size of its user community. Is often cited as the project that coined the term Pilot-Job. But despite this fact, describing HTCondor along the lines of a Pilot-Job system has turned out to be difficult, if not partly impossible for several reasons.

HTCondor is used in multiple different contexts: the HTCondor *project*, the HTCondor *software* and HTCondor *grids*. But even if we only look at the software parts of the landscape, we are faced with a *plethora* of concepts, components and services that have been grown and curated opportunistically for the past 20 years.

Core System.

A "complete" HTCondor system that accepts user tasks and executes them on one or more resources is called an HTCondor **pool**. A pool consists of multiple loosely coupled, usually distributed components that together implement the functionality of Workload Manager and Task Executor in a PilotJob system.

In a pool, user tasks (**jobs**) are represented through job description files and submitted to a (user-)**agent** via command-line tools. Agents are deployed as system services (`schedd`) on so-called *gateway machines* (user front-ends of an HTCondor pool) and accept tasks from multiple users. Agents implement three different aspects of the overall architecture: (1) they provide persistent storage for user jobs, (2) they find resource for a task to execute by contacting the **matchmaker** and (3) they marshal task execution via a so- called **shadow**.

The matchmaker, also called the *central manager*, is another system service that realizes the concept of late binding by matching user tasks with one or more of the resources available to an HTCondor pool. The matchmaking process is based on *ClassAds* a description language that can capture both, resource capabilities as well as task requirements.

Resources are tied into an HTCondor pool by `startd` system services that are analogous to our definition of a Pilot. The `startd` Pilots are deployed on the pool's compute resources. They report resource state and capabilities back to the matchmaker and start tasks submitted by agents on the resource in encapsulated **sandboxes**.

Resources in a pool can span a wide spectrum of system. While some pools are comprised of regular desktop PCs (sometimes called a campus grid), other pools incorporate large HPC clusters and cloud resources. Hybrid pools with heterogeneous sets of resources are also common.

It is possible for two or more HTCondor pools to "collaborate" so that one pool has access to the resources of another pool and vice versa. In HTCondor, this concept is called **flocking** and allows agents to query matchmakers outside their own pool for compatible resources. Flocking is used to implement load-balancing between multiple pools but also to provide a broader set of heterogeneous resources to user communities.

The components described above, jobs, agent, matchmaker, resource, shadow and sandboxes are sometimes collectively referred to as the **HTCondor Kernel** and satisfy the requirements for a Pilot-Job system as outlined in Section 3.1. However, the provisioning, allocation and usage of resources within a pool can differentiate between different pools and multiple different approaches and software systems have emerged over time, all under the umbrella of the wider HTCondor project.

Condor-G – Condor via Globus.

Condor-G is an alternative (user-)agent for HTCondor that can "speak" the Globus GRAM (Grid Resource Access and Management) protocol. GRAM services are often deployed as remote job submission endpoints on top of HPC cluster queuing systems. Condor-G allows users to incorportate those HPC resources temporarily to an HTCondor pool.

Condor-G agents use the GRAM protocol to launch HTCondor `startd` Pilots ad hoc via a GRAM endpoint service on a remote system. Tasks submitted through the Condor-G (user-)agent are then assigned by a local matchmaker to these ad-hoc provisioned Pilots. This concept is called *gliding-in* or **glide-in**. It implements **late-binding** on top of GRAM/HPC systems: the (user-)agent can assign tasks to `startd`s *after* they have been scheduled and started through the HPC queueing system. This effectively decouples resource allocation (`startd` scheduling through GRAM) and task assignment.

glidein-WMS – Automated Pilot Provisioning.

The glidein workload management system (WMS) [37] introduces advanced Pilot-Job capabilities to HTCondor by providing automated Pilot (`startd`) provisioning based on the state of an HTCondor pool.

BoSCO.

BoSCO is a user-space job submission system based on HTCondor. BoSCO was designed to allow individual users to utilize heterogeneous HPC and grid computing resources through a uniform interface. Supported backends include

PBS, LSF and GridEngine clusters as well as other grid resource pools managed by HTCondor. BoSCO supports both, an agent-based (*glidein* / worker) and a native job execution mode through a single user-interface.

BoSCO exposes the same *ClassAd*-based user-interface as HTCondor, however, the backend implementation for job management and resource provisioning is significantly more lightweight than in HTCondor and explicitly allows for ad hoc user-space deployment. BoSCO provides a Pilot-Job-based system that does not require the user to have access to a centrally- administered HTCondor campus grid or resource pool. The user has direct control over Pilot-Job agent provisioning (via the `bosco_cluster` command) and job-to-resource binding via *ClassAd* requirements.

The overall architecture of BoSCO is very similar to that of HTCondor. The *BoSCO submit-node* (analogous to Condor `schedd`) provides the central job submission service and manages the job queue as well as the worker agent pool. Worker agents communicate with the *BoSCO submit-node* via pull-requests (TCP). They can be dynamically added and removed to a *BoSCO submit-node* by the user. BoSCO can be installed in user-space as well as in system space. In the former case, worker agents are exclusively available to a single user, while in the latter case, worker agents can be shared among multiple users. The client-side tools to submit, control and monitor BoSCO jobs are the same as in Condor (`condor_submit`, `condor_q`, etc).

- Workflow (doesn't seem to be explicit support)
- Workload (is this just a list of user jobs that are queued up)
- Placeholder (isn't this the same kind of deal as a Pilot-Job/PJ waiting in a queue?)
- Pilot framework (The entire GlideinWMS qualifies as a "pilot framework", correct?)
- Ensemble
- Platform
- Master-Worker

CorralWMS: CorralWMS is an alternative front-end for GlideinWMS-based infrastructures. It replaces or complements the regular GlideinWMS front-end with an alternative API which is targeted towards workflow execution. Corral was initially designed as a standalone pilot (glidein) provisioning system for the Pegasus workflow system where user workflows often produced workloads consisting of many short-running jobs as well as mixed workloads consisting of HTC and HPC jobs.

Over time, Corral has been integrated into the GlideinWMS stack as CorralWMS. While CorralWMS still provides the same user-interface as the initial, stand- alone version of Corral, the underlying pilot (glidein) provisioning is now handled by the GlideinWMS factory.

The main differences between the GlideinWMS and the CorralWMS front-ends lie in identity management and resource sharing. While GlideinWMS pilots (glidins) are provisioned on a per-VO base and shared / re-used amongst members of that VO, CorralWMS pilots (glideins) are bound to one specific user via personal X.509 certificates. This enables explicit resource provisioning in non-VO centric environments, which includes many of the HPC clusters that are part of U.S. national cyberinfrastructure (e.g., XSEDE).

4.2.5 Falkon

Mapping to section 3:

- Term 1 =
- Term 2 =

Mapping to table 1:

- Pilot (Resource) Provisioning:
- Job-to-Resource (Pilot) Binding:
- Singe-/Multi-User pilots:

Rationale.

The Fast and Light-weight tasK executiON framework (Falkon) [32] was created with the primary objective of enabling many independent tasks to run on large computer clusters (an objective shared by most Pilot-Job systems). A particular focus on performance and time-to-completion for jobs on such clusters drove Falkon development. In addition to being *fast*, Falkon, as its name suggests, also focused on lightweight deployment schemes.

Design Goals.

As previously stated, the design of Falkon was centered around the goal of providing support to run large numbers of jobs efficiently on large clusters and grids. Falkon realizes this goal through the use of (i) a dispatcher to reduce the time to actually place tasks as jobs onto specific resources (such a feature was built to account for different issues amongst distributed cyberinfrastructure - such as multiple queues, different task priorities, allocations, etc), (ii) a provisioner which is responsible for resource management, and (iii) data caching in a remote environment [32].

Applications.

Falkon has been shown to work with many large-scale applications across various domains. Falkon has been integrated with the Karajan workflow language and execution engine, meaning that applications that utilize Karajan to describe their workflow will be able to be executed by Falkon. Simpler task execution can be achieved without modifying the existing executables - sufficient task description in the web service is all that is required to utilize the Falkon system.

Falkon has been tested for throughput and performance in such as applications as fMRI (medical imaging), Montage (astronomy workflows), and MolDyn (molecular dynamics simulation) and has shown favorable results in terms of overall execution time when compared to GRAM and GRAM/-Clustering methods [32].

Deployment Scenarios.

The Swift parallel programming system [43] was integrated with Falkon for the purpose of task dispatch. The overall provider mechanism of Falkon is roughly 840 lines of Java code and meant to be as lightweight as possible. The

Dispatcher service in Falkon is implemented by means of a web service. This Dispatcher implements a factory/instance deployment scenario. When a new client sends task submission information, a new instance of a Dispatcher is created. Each instantiation of the Dispatcher maintains its own task queue and state - in this way, Falkon can be considered a single-user deployment scheme, wherein the "user" in this case refers to an individual client request.

Resource Landscape (e.g. grid/cloud/hpc/etc).

Falkon was originally developed for use on large computer clusters in a grid environment, but has since been expanded to work on clouds and other

Falkon has been shown to run on TeraGrid (now XSEDE), TeraPort, Amazon EC2, IBM Blue Gene/L, SiCortex, and Workspace Service [32]. Work has also been done on Falkon to expand its data capabilities. In addition to data caching and efficient data scheduling techniques, Falkon has adopted a data diffusion approach. Using this approach, resources for both compute and data are acquire dynamically and compute is scheduled as close as possible to the data it requires. If necessary, the data diffusion approach replicates data in response to changing demands [33].

Architecture & Interface.

Falkon's architecture relies on the use of multi-level scheduling as well as efficient dispatching of tasks to heterogeneous DCIs. As mentioned above, there are two main components of Falkon: (i) the Dispatcher for farming out tasks and (ii) the Provisioner for acquiring resources.

The overall task submission mechanism can be considered a 2-tier architecture; the Dispatcher (using the above terminology, this is the Pilot-Manager) and the Executor (the Pilot-Agent). The Dispatcher is a GRAM4 web service whose primary function is to take task submission as input and farm out these tasks to the executors. The Executor runs on each local resource and is responsible for the actual task execution. Falkon also utilizes *provisioning* capabilities with its Provisioner. The Falkon Provisioner is the closest analogous entity to a Pilot: it is the creator and destroyer of Executors, and is capable of providing both static and dynamic resource acquisition and release.

Falkon has also extended itself beyond basic Pilot-Job functionalities and supports a fault tolerance mechanism which suspends and dynamically readjusts for host failures. The *data management* capabilities of Falkon also extend beyond the core Pilot-Job functionalities as described above. Further, in order to process more complex workflows, Falkon has been integrated with the Karajan workflow execution engine [41]. This integration allows Falkon to accept more complex workflow-based scientific applications as input to its Pilot-like job execution mechanism.

Falkon Conclusion.

Falkon contains many of the basic functionalities required to qualify it as a Pilot-Job system, in addition to some advanced resource provisioning capabilities, fault tolerance, and workflow-execution integration. Falkon has been demonstrated to achieve throughput in the range of hundreds to thousands of tasks per second for very fine-grained tasks. The per task overhead of Falkon execution has been shown to be in the millisecond range. Falkon has extended its capabilities to encompass advanced data-scheduling and caching. Falkon does not support MPI jobs, however - a limiting factor in its adoption to certain scientific applications.

–

4.2.6 BigJob

Mapping to section 3:

- Term 1 =
- Term 2 =

Mapping to table 1:

- Pilot (Resource) Provisioning:
- Job-to-Resource (Pilot) Binding:
- Singe-/Multi-User pilots:

Rationale.

BigJob was designed as a flexible and extensible Pilot-Job to work on a variety of infrastructures. It was designed to natively support many scientific applications, including MPI jobs **??**.

Design Goals.

BigJob supports the basic functionality required of a Pilot-Job, with the added capability of interfacing with a number of different batch queuing systems and infrastructures (grids, clouds, clusters, etc.). BigJob was also designed to support MPI jobs without adding additional configuration requirements to the end-user. Lastly, the application-level programmability that BigJob offers was incorporated as a means of giving the end-user more flexibility and control over their job management.

Most recently, BigJob has been extended to work with data and, similarly to Pilot-Jobs, abstract away direct user communication between different storage systems. This work has extended BigJob from being a purely Pilot-Job-based system to a more complete job and data management system.

Applications.

BigJob has been used for many different ensemble-based or replica-exchange-based applications in both the computational chemistry and bioinformatics disciplines. It has been shown to work across multiple XSEDE **??** machines and scale up to thousands of concurrent jobs. It has also been used as the underlying job management layer in the release of an asynchronous replica-exchange software package **??**.

Deployment Scenarios (Single User).

BigJob is installable by a user onto the resource of his or her choice. It is capable of running only in "single user" mode; that is, a Pilot belongs to the user who spawned it and cannot be accessed by other users.

Resource Landscape (e.g. grid/cloud/hpc/etc).

BigJob uses the Simple API for Grid Applications (SAGA) **??** in order to interface with different grid middleware. It can work on HPC grid environments, such as XSEDE or FutureGrid , as well as personal clusters with batch queuing systems. In addition, BigJob has been shown to work on

clouds **??**, where it has the capability to both launch VMs and submit jobs to these VMs.

Architecture & Interface.

In the BigJob framework, a Pilot is, appropriately enough, called a Pilot. The Pilot-*Manager* is called a *BigJob- Manager*, which is the central coordinator of the framework. The *BigJob-Manager* is responsible for the orchestration and scheduling of Pilots. It runs locally on the machine used to initiate the distributed application which may or may not be the same resource as the machine used on which the distributed application executes.

The Manager ensures that tasks are launched onto the correct resource using the correct number of processes. The BigJob-Manager submits a Pilot to a remote resource's batch queueing system; when these Pilots becomes active, a *BigJob-Agent* on the remote resource gathers information about its resource and executes the actual tasks the resource. The *BigJob-Agent* represents the Pilot-Job, and thus, the application-level resource manager on the respective resource.

The communication between the Manager and Agent(s) is achieved through the *Distributed Coordination Service*, which is most often a database that stores information about jobs, information about the jobs (executable, input data, etc.), and job status. In the BigJob framework, this Distributed Coordination Service is achieved through the use of the *redis* database. The Pilot creation and management capabilities of BigJob meet the baseline capabilities for a Pilot-Job as described in Section 3.

BigJob Conclusion.

Using SAGA has enabled BigJob to expand to many of the changing and evolving architectures and middlewares. As a result, BigJob can grow to accomodate new usage modes on heterogenous architectures. The uptake of BigJob by the scientific community on both grid and cloud architectures shows the dynamism Pilot-Jobs offer. The use of a Pilot-Job system, such as BigJob, to marshal VMs as "Pilots" also lends itself to the extensibility of the Pilot-Job concept.

4.2.7 DIANE

Mapping to section 3:

- Term 1 =
- Term 2 =

Mapping to table 1:

- Pilot (Resource) Provisioning:
- Job-to-Resource (Pilot) Binding:
- Singe-/Multi-User pilots:

Rationale.

DIANE [25] is a task coordination framework, which was originally designed for implementing master/worker applications, but also provides PJ functionality for job-style executions.

Design Goals.

DIANE utilizes a single hierarchy of worker agents as well as a PJ *manager* referred to as `RunMaster`. The manager creates placeholders called `Workers`, which register with and receive jobs from the manager. The *scheduler* for DIANE is known as the `Planner`.

Further, DIANE supports fault tolerance: basic error detection and propagation mechanisms are in place. Further, an automatic re-execution of WUs is possible.

Applications.

DIANE was originally conceived for HEP applications at CERN, but has since been used for various other domains, though mainly in Life Sciences.

Deployment Scenarios (Single User).

Deployment Scenarios (Multi User/VO).

Architecture & Interface.

DIANE's architecture is based on the *Inversion of Control* design pattern. In DIANE's case, this means that it is a python-based framework, that formulates certain hooks that an "application" can be programmed against. These DIANE-applications are then started through a diane-run command.

The workflow is then such that a user submits a parallel job to the grid using a DIANE client. The client then creates the Pilot manager and remains in contact with it to control its Pilot-Jobs. Results are aggregated through use means of the DIANE `Integrator`. DIANE includes a simple capability matcher and FIFO-based task scheduler to help facilitate the execution of jobs. Plugins for other workloads, e. g. DAGs or for data-intensive application, exist or are under development. The framework is extensible: applications can implement a custom application-level scheduler.

For communication between the RunMaster and worker agents point-to-point messaging based on CORBA [27] is used. CORBA is also used for file staging. DIANE is a single-user PJ, i. e. each PJ is executed with the privileges of the respective user. Also, only WUs of this respective user can be executed by DIANE. DIANE supports various middleware security mechanisms (e. g. GSI, X509 authentication). For this purpose it relies on GANGA. The implementation of GSI on TCP-level is possible, but currently not yet implemented.

Resource Landscape (e.g. grid/cloud/hpc/etc).

DIANE is primarily designed with respect to HTC environments (such as EGI [16]), i. e. one PJ consists of a single worker agent with the size of 1 core.

DIANE Conclusion.

5. CONCLUSION AND DISCUSSION

It should be noted that cloud-based DCIs introduce notable exceptions and differences in the way in which pilots can be provisioned.[cit, cit] Within a IaaS [cit], Virtual Machines (VMs) and not jobs are used for their provisioning. VMs can often be instantiated without waiting into a queue, and limitations on the execution time of a VM can be virtually absent. Clearly, overheads are introduced by having to deal with VMs and not simple jobs and the model adopted within a IaaS-based DCI to assign resources to each VM can affect the flexibility of the whole Pilot-Job system. [cit] A

similar assessment could be done for a DCI deploying a PaaS model of cloud computing.[cit]

Another important detail to notice is that while the operations of a task performed by means of a Pilot are usually computational in nature, in principle, they might also operate on other type of resources such as data or network bandwidth, depending on the resources made available by an infrastructure to its users. In this context, it would be likely for some Pilot-Job systems to use the terms 'pilot data' or 'pilot network'.

5.1 revisit the "myths"

5.2 The need for a common minimum model

"pilot-abstractions works!" , p* as a model ok.

Our initial investigation [23] into Pilot-Abstractions was motivated by the desire to provide a single conceptual framework — referred to as the P* Model, that would be used to understand and reason the plethora and myriad Pilot-Job implementations that exist.

Once a common and uniform conceptual model was available, the notion of Pilot-Data was conceived using the power of symmetry, i.e., the notion of Pilot-Data was as fundamental to dynamic data placement and scheduling as Pilot-Jobs was to computational tasks. As a measure of validity, the P* model was amenable and easily extensible to Pilot-Data. The consistent and symmetrical treatment of data and compute in the model led to the generalization of the model as the *P* Model of Pilot Abstractions*.

5.3 Lessons for Workflow System

The current state of workflow (WF) systems [2, 39] provides a motivating example for the P* Model and the Pilot-API: even though many WF systems exist (with significant duplicated effort), they provide limited means for extensibility and interoperability. We are not naive enough to suggest a single reason, but assert that one important contributing fact is the lack of the right interface abstractions upon which to construct workflow systems; had those been available, many/most WF engines would have likely utilized them (or parts thereof), instead of proprietary solutions.

Significant effort has been invested towards WF interoperability at different levels – if nothing else, providing post-facto justification of its importance. The impact of missing interface abstractions on the WF world can be seen through the consequences of their absence: WF interoperability remains difficult if not infeasible. The Pilot-API in conjunction with the P* Model aims to prevent similar situation for Pilot-Jobs.

The section for Unresolved Ideas and Issues

6. DISCUSSION AREA

Acknowledgements

This work is funded by NSF CHE-1125332 (Cyber-enabled Discovery and Innovation), NSF-ExTENCI (OCI-1007115) and "Collaborative Research: Standards-Based Cyberinfrastructure for Hydrometeorologic Modeling: US-European Research Partnership" (OCI-1235085) and Department of Energy Award (ASCR) DE-FG02-12ER26115. This work has also been made possible thanks to computer resources provided by TeraGrid TRAC award TG-MCB090174 and BiG Grid. This document was developed with support from the US NSF under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed".

7. REFERENCES

[1] "Co-Pilot: The Distributed Job Execution Framework", Predrag Buncic, Artem Harutyunyan, Technical Report, Portable Analysis Environment using Virtualization Technology (WP9), CERN.

[2] Final Report of NSF Workshop on Challenges of Scientific Workflows. 2006. `http://www.isi.edu/nsf-workflows06`.

[3] 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*, 3(08):S08003, 2008.

[4] S. Ahn, N. Kim, S. Lee, S. Hwang, D. Nam, B. Koblitz, V. Breton, and S. Han. Improvement of task retrieval performance using amga in a large-scale virtual screening. In *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 01*, NCM '08, pages 456–463, Washington, DC, USA, 2008. IEEE Computer Society.

[5] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[6] S. Bagnasco, L. Betev, P. Buncic, F. Carminati, C. Cirstoiu, C. Grigoras, A. Hayrapetyan, A. Harutyunyan, A. J. Peters, and P. Saiz. Alien: Alice environment on the grid. *Journal of Physics: Conference Series*, 119(6):062012, 2008.

[7] S. Belforte, I. Hsu, E. Lipeles, and M. Norman. GlideCAF: A Late Binding Approach to the Grid. *CHEP06*, 2006.

[8] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, Apr. 2003.

[9] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Nov. 1996.

[10] T. Q. Bui, T. T. Doan, H. T. Nguyen, Q. M. Pham, S. V. Nguyen, V. Breton, L. Q. Pham, H. M. Le, H. Q. Nguyen, and E. Medernach. On the Performance Enhancement of WISDOM Production Environment. In *Proceedings of The International Symposium on Grids and Clouds (ICGC 2012). 26 February-2 March. Taipei, Taiwan. Published online at <A href="http://pos.sissa.it/cgi-bin/reader/conf.cgi?confid=153">http://pos.sissa.it/cgi-bin/reader/conf.cgi?confid=153</A>, id.1*, 2012.

[11] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An

architecture for a resource management and scheduling system in a global computational grid. *International Conference on High-Performance Computing in the Asia-Pacific Region*, 1:283–289, 2000.

[12] A. Casajus, R. Graciani, S. Paterson, and A. Tsaregorodtsev. Dirac pilot framework and the dirac workload management system. *Journal of Physics: Conference Series*, 219(6):062049, 2010.

[13] P.-H. Chiu and M. Potekhin. Pilot factory – a condor-based system for scalable pilot job generation in the panda wms framework. *Journal of Physics: Conference Series*, 219(6):062041, 2010.

[14] Coasters. `http://wiki.cogkit.org/wiki/Coasters`, 2009.

[15] K. De. Next generation workload management system for big data. Presented at the BNL HPC Workshop, 2013.

[16] EGI. `http://www.egi.eu/`, 2012.

[17] 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*, 12(1), May 1996.

[18] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, July 2002.

[19] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, pages 214–227, London, UK, UK, 2000. Springer-Verlag.

[20] 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 HPDC*, pages 477–488, 2010.

[21] 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*, CCGRID '10, pages 349–358, Washington, DC, USA, 2010. IEEE Computer Society.

[22] G. Lawton. Distributed net applications create virtual supercomputers. *Computer*, 33(6):16–20, 2000.

[23] A. Luckow, S. Jha, J. Kim, A. Merzky, and B. Schnor. Distributed replica-exchange simulations on production environments using saga and migol. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 253–260, Washington, DC, USA, 2008. IEEE Computer Society.

[24] T. Maeno, K. De, and S. Panitkin. PD2P: PanDA dynamic data placement for ATLAS. In *Journal of Physics: Conference Series*, volume 396, page 032070, 2012.

[25] J. Moscicki. Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data. In *Nuclear Science Symposium Conference Record, 2003 IEEE*, volume 3, pages 1617 – 1620, 2003.

[26] 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 *Journal of Physics: Conference Series*, volume 396, page 032080. IOP Publishing, 2012.

[27] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, M 2004.

[28] Open Science Grid. Bosco. `http://bosco.opensciencegrid.org/`.

[29] Open Science Grid Consortium. Open Science Grid Home. `http://www.opensciencegrid.org/`.

[30] J. pierre Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Cluster Computing*, pages 43–50. Society Press, 2000.

[31] C. Pinchak, P. Lu, and M. Goldenberg. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. In *In Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP*, pages 85–105. Springer Verlag, 2002.

[32] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: A Fast and Light-Weight TasK ExecutiON Framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[33] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 9–18. ACM, 2008.

[34] A. J. Rubio-Montero, E. Huedo, and R. Mayo-Garcia. Gwpilot: a personal pilot system. In *Proceedings of EGI Forum*, 2012.

[35] M. Rynge, G. Juve, G. Mehta, E. Deelman, K. Larson, B. Holzman, I. Sfiligoi, F. Wurthwein, G. B. Berriman, and S. Callaghan. Experiences using glideinwms and the corral frontend across cyberinfrastructures. In *Proceedings of the 2011 IEEE Seventh International Conference on eScience*, ESCIENCE '11, pages 311–318, Washington, DC, USA, 2011. IEEE Computer Society.

[36] P. Saiz, P. Buncic, and A. J. Peters. AliEn Resource Brokers. June 2003.

[37] I. Sfiligoi. Glideinwms—a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119(6):062044, 2008.

[38] G. Shao, F. Berman, and R. Wolski. Master/Slave Computing on the Grid. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*. IEEE Computer Society, May 2000.

[39] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[40] Topos - a token pool server for pilot jobs. `https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS`, 2011.

[41] G. Von Laszewski, M. Hategan, and D. Kodeboyina. Java cog kit workflow. In *Workflows for e-Science*,

pages 340–356. Springer, 2007.

[42] E. Walker, J. Gardner, V. Litvin, and E. Turner. Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pages 95–103, 0-0 2006.

[43] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[44] G. Woltman and S. Kurowski. The great internet mersenne prime search, 2004.

[45] X. Zhao, J. Hover, T. Wlodek, T. Wenaus, J. Frey, T. Tannenbaum, M. Livny, and the ATLAS Collaboration. Panda pilot submission using condor-g: Experience and improvements. *Journal of Physics: Conference Series*, 331(7):072069, 2011.