# A Comprehensive Perspective on Pilot-Abstraction

Matteo Turilli
RADICAL Laboratory, ECE
Rutgers University

Mark Santcroos
RADICAL Laboratory, ECE
Rutgers University

Shantenu Jha*
RADICAL Laboratory, ECE
Rutgers University

## ABSTRACT

This paper provides a comprehensive perspective of Pilot-Jobs, a type of middleware software that has been gaining rapid adoption among various scientific communities. For example, Pilot-Jobs systems are used to provide more than 700 million CPU hours a year to the Open Science Grid (OSG) communities and to process up to 5 million jobs a week for the ATLAS experiment. Notwithstanding the growing impact of Pilot-Jobs on scientific research, there is no agreement upon their definition, no clear understanding of the underlying abstraction and paradigm, no shared best practices or interoperability among their implementations. This lack of foundational understanding and of design convergence is hindering the full exploitation of the Pilot-Jobs potential dispersing the available resources across a fragmented development landscape. This paper offers the conceptual tools to promote shared understanding of the Pilot paradigm while critically reviewing the state of the art of Pilot-Jobs implementations. Five main contributions are provided: (i) an analysis of the motivations and evolution of the Pilot-Job abstraction; (ii) an outline of the minimal set of distinguishing functionalities; (iii) the definition of a core vocabulary to reason consistently about Pilot-Jobs; (iv) the description of core and auxiliary properties of Pilot-Jobs systems; and (v) a critical review of the current state of the art of their implementations. These contributions are brought together to illustrate the defining characteristics of the Pilot-Job paradigm, its generality, and the main opportunities and challenges posed by its support of distributed computing.

## 1. INTRODUCTION

The seamless uptake of distributed computing infrastructures by scientific applications has been limited by the lack of pervasive and simple-to-use abstractions at the development, deployment, and execution level. As suggested by a survey of actual usage , Pilot-Job is arguably one of the most widely-used distributed computing abstractions among all those proposed to support effective distributed resource utilization. Pilot-Jobs excel in terms of 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 facilitate the otherwise challenging mapping of specific tasks onto explicitly defined, possibly heterogeneous and dynamic, resource pools. Pilot-Jobs decouple the workload specification from the task management improving the efficiency of task assignment while shielding applications from having to manage tasks across such resources. Another concern often addressed by Pilot-Jobs is fault tolerance which commonly refers to 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 [34, 33].

A variety of Pilot-Job frameworks have emerged: Condor-G/ Glide-in [24], Swift [63], DIANE [43], DIRAC [12], PanDA [13], ToPoS [57], Nimrod/G [11], Falkon [48] and MyCluster [61] to name a few. These frameworks are for the most part functionally equivalent as they support the decoupling of workload submission from resource assignment. Nonetheless, their implementations often serve specific use cases, target specific resources, and lack in interoperability. 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, these objectives remain difficult if not unfeasible.

Thought mostly as a pragmatic solution to the need of improving throughput performance of distributed applications, Pilot-Jobs have been almost exclusively developed within preexisting frameworks and middleware dedicated to specific scientific needs. As a consequence, the development of Pilot-Jobs have not been grounded on a robust understanding of underpinning abstractions, or on a well-understood set of dedicated design principles. The functionalities and properties of Pilot-Jobs have been understood mostly in relation to the needs of the containing software frameworks or of the user cases justifying their development.

This approach is not problematic in itself and has led to ef-

*Corresponding author

fective implementations that serve many million jobs a year on diverse computing platforms. However, the lack of an explicit appreciation of the specific computing paradigm underlying Pilot-Jobs has undermined the full development of dedicated designs and implementations. This limitation is well illustrated not only by the effort duplication but also by an overall immaturity of the available frameworks in terms of functionalities, flexibility, portability, interoperability, and, most often, robustness.

This paper offer a critical analysis of the current state of the art providing the conceptual tooling required to appreciate the properties of the Pilot paradigm. The remainder of this paper is divided into four sections. §2 offers a critical review of the functional underpinnings of the pilot abstraction and how it has been evolving into Pilot systems and systems with pilot-like characteristics.

In §3, the minimal set of capabilities and properties characterizing the design of a Pilot system are derived. A vocabulary is then defined to be consistently used across different designs and Pilot system implementations.

In §4, the focus shifts from analyzing the design of a Pilot system to critically reviewing the characteristics of a representative set of its implementations. Core and auxiliary implementation properties are introduced and then used alongside the functionalities and terminology defined in §3 to compare Pilot systems implementations.

Finally, §5 closes the paper by outlining the Pilot paradigm, arguing for its generality, and elaborating on how it impacts and relates to both other middleware and the application layer. The outcome of the critical review of the current implementation state of the art is used to give insights about the future directions and challenges faced by the Pilot paradigm.

## 2. EVOLUTION OF PILOT ABSTRACTION AND SYSTEMS

The origin and motivations for devising the Pilot abstraction, developing its many implementations and realize a full-fledge Pilot paradigm can be traced back to five main notions: task-level distribution and parallelism, master-worker pattern, multi-tenancy, multi-level scheduling, and resource placeholder. This section offers an overview of these five notions and an analysis of their relationship with the Pilot abstraction. A chronological perspective is taken so to contextualize the inception and evolution of the Pilot abstraction into its implementation systems.

### 2.1 Functional Underpinnings of the Pilot Abstraction

To the best of the authors' knowledge, the term 'Pilot' was first coined in 2004 in the context of the Large Hadron Collider (LHC) Computing Grid (LCG) Data Challenge[1], and then introduced in writing as 'pilot-agent' in a 2005 LHCb report[56]. Despite its relatively recent explicit naming, the Pilot abstraction addresses a problem already well-known at the beginning of the twentieth century: task-level distribution and parallelism on multiple resources.

Lewis Fry Richardson devised in 1922 a Forecast Factory (Figure 1) to solve systems of differential equations for weather forecasting. This factory required 64,000 'human computers' supervised by a senior clerk. The clerk would distribute portions of the differential equations to the computers so that they could forecast the weather of specific regions of the globe. The computers would perform their calculations and then send the results back to the clerk. The Forecast Factory was not only an early conceptualization of what is called today a "supercomputer" [2] but also of the coordination pattern for distributed and parallel computation called "master-worker".

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

The clerk of the Forecast Factory is the 'master' while the human computers are her 'workers'. Requests and responses go back and forth between the master and all its workers. Each worker has no information about the overall computation nor about the states of any other worker. The master is the only one possessing a global view both of the overall problem to compute and of its progress towards a solution. As such, the master-worker is a coordination pattern allowing for the structured distribution of tasks so to orchestrate their parallel execution. This directly translates into a better time to completion of the overall computation when compared to a coordination pattern in which each equation is sequentially solved by a single worker.

Modern silicon-based supercomputers brought at least three key differences when compared to the carbon-based Forecast Factory devised by Richardson. Most of modern supercomputers are meant to be used by multiple users, i.e. they support multi-tenancy. Furthermore, diverse supercomputers were made available to the scientific community, each with both distinctive and homogeneous properties in terms of architecture, capacity, capabilities, and interfaces. Furthermore, supercomputers supported different types of applications, depending on the applications' communication and coordination models.

Multi-tenancy has defined the way in which high performance computing resources are exposed to their users. Job schedulers, often called "batch queuing systems" [16] and first used in the time of punch cards [32, 55], leverage the

[1]Based on private communication.

[2]Supercomputer is here used as a general term indicating a system with a very large computing capacity. As such, this term makes no distinction among alternative supercomputer architectures.

batch processing concept to promote efficient and fair resource sharing. Job schedulers implement a usability model where users submit computational tasks called "jobs" to a queue. The execution of these job is delayed waiting for the required amount of resources to be available. The amount of delay mostly depends on the size and duration of the submitted job, resource availability, and fair usage policies.

Supercomputers are often characterized by several types of heterogeneity and diversity. Users are faced with different job description languages, job submission commands, and job configuration options. Furthermore, the number of queues exposed to the users and their properties like walltime, duration, and compute-node sharing policies vary from resource to resource. Finally, each supercomputer may be designed and configured to support only specific types of application.

The resource provision of multi-tenant and heterogeneous supercomputers is limited, irregular, and largely unpredictable [18, 64, 36, 58]. By definition, the resources accessible and available at any given time can be less than those demanded by all the active users. Furthermore, the resource usage patterns are not stable over time and alternating phases of resource availability and starvation are common [25, 38]. This landscape led not only to a continuous optimization of the management of each resource but also to the development of alternative strategies to expose and serve resources to the users.

Multi-level or meta scheduling is one of the strategies devised to improve resource access across multiple supercomputers. The idea is to hide the scheduling point of each supercomputer behind a single (meta) scheduler. The users or the applications submit their tasks to the single scheduler that negotiates and orchestrates the distribution of the tasks via the scheduler of each available supercomputer. While this approach promises an increase in both scale and usability of applications, it also introduces diverse types of complexity across resources, middleware, and applications.

Several approaches have been devised to manage the complexities associated with multi-level scheduling. Some approaches, for example those developed under the umbrellas of grid computing or cloud computing, targeted the resource layer, others the application layer as, for example, with workflow frameworks. All these approaches offered and still offer some degree of success for specific applications and use cases but a general solution based on well-defined and robust abstractions has still to be devised and implemented.

One of the persistent issues besetting resource management across multiple supercomputers is the increase of the implementation complexity imposed on the application layer. Even with solutions like grid computing aiming at effectively and, to some extent, transparently integrating diverse resources, most of the requirements involving the coordination of task execution still lays with the application layer. This translates into single-point solutions, extensive redesign and redevelopment of existing applications when they need to be adapted to new use cases or new resources, and lack of portability and interoperability.

Consider for example a simple distributed application implementing the master-worker pattern. With a single supercomputer, the application requires the capability of concurrently submitting tasks to the queue of the supercomputer scheduler, and retrieve and aggregate their outputs. When multiple supercomputers are available, the application requires directly managing submissions to several queues or the capability to leverage a third-party meta-scheduler and its specific execution model. In both scenarios, the application requires a large amount of development and capabilities that are not specific to the given scientific problem but pertain instead to the coordination and management of its computation.

The notion of resource placeholder was devised as a pragmatic and relatively cheap to implement attempt to reduce or at least better manage the complexity of executing distributed applications. A resource placeholder decouples the acquisition of remote compute resource from their use to execute the tasks of a distributed application. Resources are acquired by scheduling a job onto the remote supercomputer. Once executed, the job runs an agent capable of retrieving and executing application tasks.

Resource placeholders bring together multi-level scheduling to enable parallel execution of the tasks of distributed applications. Multi-level scheduling is achieved by scheduling the agent and then by enabling direct scheduling of application tasks to that agent. The master-worker pattern is often an effective choice to manage the coordination of tasks execution on the available agent(s). Multi-level scheduling can be extended to multiple resources by instantiating resource placeholders on diverse supercomputers and then using a meta-scheduler to schedule tasks across all the placeholders.

It should be noted that resource placeholders also mitigate the side-effects introduced by a multi-tenant scheduling of resource placeholders. A placeholder still spends a variable amount of time waiting to be executed by the batch system of the remote supercomputer, but, once executed, the user – or the master process of the distributed application – may hold total control over its resources. In this way, tasks are directly scheduled on the placeholder without competing with other users for the supercomputer scheduler.

## 2.2 Evolution of Pilot System Implementations

The Pilot abstraction has a rich set of properties [40] that have been progressively implemented into multiple Pilot systems. Figure 2 shows the introduction of Pilot systems over time while Figure 3 shows their clustering along the axes of workload management and pilot functionalities complexity. Starting from a set of core functionalities focused on acquiring remote resources and utilize them independently from the supercomputer management, Pilot systems progressively evolved including advanced capabilities like workload and data management.

AppLeS [8] is a framework for application-level scheduling and offers an example of an early implementation of resource placeholder. AppLeS provides an agent that can be embedded into an application thus enabling the application to acquire resources and to schedule tasks onto these. Besides master-worker, AppLeS also provides application templates, e. g. for parameter sweep and moldable parallel applications [7].

AppLeS offered user-level control of scheduling but did not isolate the application layer from the management and coordination of task execution. Any change in the coordination mechanisms directly translated into a change of the application code. The next evolutionary step was to create

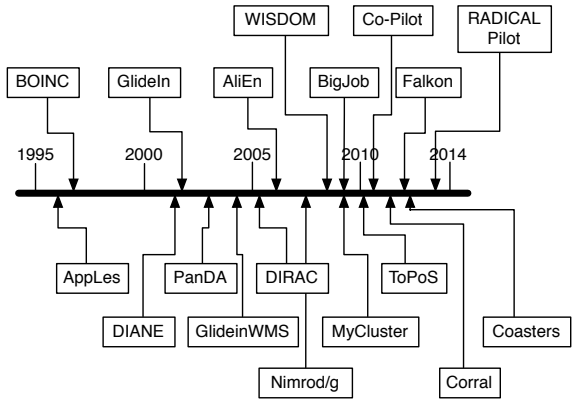


**Figure 2: Introduction of systems over time. When available, the date of first mention in a publication or otherwise the release date of software implementation is used.**

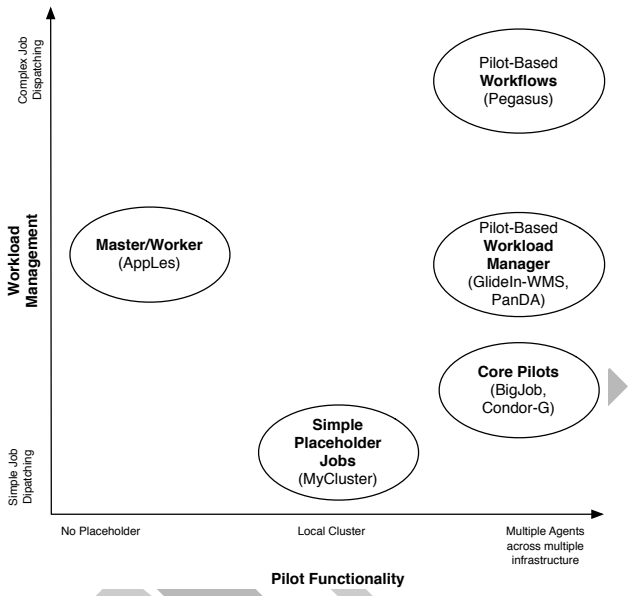


**Figure 3: Pilot-Job Clustering**

a dedicated abstraction layer between those of the application and of the various batch queuing systems available at different remote systems.

Around the same time as AppLeS was introduced, volunteer computing projects started using the master-worker coordination pattern to achieve high-throughput calculations for a wide range of scientific problems. The workers of these systems could be downloaded and installed on the users workstation. With an installation base distributed across the globe, workers pulled and executed computation tasks when CPU cycles were available.

The volunteer workers were essentially heterogeneous and dynamic as opposed to the homogeneous and static AppLeS workers. The idea of farming out tasks in a dynamic distributed environment including personal computers promised to lower the complexity of distributed applications design and implementation. Each volunteer worker can be seen as an opportunistic resource placeholder and, as such, an implementation of the core functionality of the Pilot abstraction.

The first public volunteer computing projects were The Great Internet Mersenne Prime Search effort[65], shortly followed by distributed.net [35] in 1997 to compete in the RC5-56 secret-key challenge, and the SETI@Home project, which set out to analyze radio telescope data. The generic BOINC distributed master-worker framework grew out of SETI@Home, becoming the *de facto* standard framework for voluntary computing [4].

It should be noted that process of resource acquisition is different in AppLes and voluntary computing. The former has complete knowledge of the available resources while the latter has none. As a consequence, AppLes can request and orchestrate a set of resource, allocate tasks in advance to specific workers (i.e. resources placeholders), and implement load balancing among resources. In voluntary computing tasks are pulled by the clients when they become active and, as such, specific resource availability is unknown in advance. This potential drawback is mitigated by the redundancy offered by the large scale that voluntary computing can reach thanks to its simpler model of worker distribution and installation.

The opportunistic use of geographically distributed resources championed by voluntary computing offers several advantages. The resource landscape available for scientific research is fragmented across multiple institutions, managed with different policies and protocols, and heterogeneous both in quantity and quality. Once aggregated, the sum of otherwise limited resources can support very large distributed computations and a great amount of multi-tenancy. Note that given the required capabilities, this model of resource provisioning can still support the execution of parallel applications on the few resources that offer low-latency network interconnect.

Condor is a high-throughput distributed batch computing system that leverages diverse and possibly geographically distributed resources. Originally, Condor was created for systems within one administrative domain but Flocking [20] made possible to group multiple Condor resource pools in an aggregative manner. However, resource management could not be done on application level by the user: flocking required system level software configurations that had to be made by the administrator of each individual resource of each Condor resource pool.

This limitation was overcame by integrating a resource placeholder mechanism within the Condor system. Condor-G/GlideIn [24] allow users to add remote grid resources to Condor resource pools. In this way, users can uniformly execute jobs on resource pools composed by diverse resources. Thanks to its use of resource placeholders, GlideIn has been one of the systems pioneering the Pilot abstraction implementation, enabling Pilot capabilities also for third parties systems like Bosco [62].

The success of Condor-G/GlideIn shows the relevance of the pilot abstraction to enable scientific computation at scale and on heterogeneous resources. The implementation of GlideIn also highlighted at least two limitations: user/system layer isolation, and application development model. While GlideIn allows for the user to manage resource placeholders directly, daemons must still be running on the re-

mote resources. This means that GlideIn cannot be deployed without involving the resource owners and system administrators. Implemented as a service, GlideIn support integration with distributed application frameworks but does not programmatically support the development of distributed applications by means of dedicated APIs and libraries.

The BigJob Pilot system [ref] was designed to address these limitations, to broaden the type of applications supported by the Pilot-based execution model, and to extend the Pilot abstraction beyond the boundaries of compute tasks. BigJob offers an application-level programmability to provide the end-user with more flexibility and control over the design of distributed application and the isolation of the management of their execution. BigJob is flexible and extensible and uses the Simple API for Grid Applications (SAGA) interoperability library [28, 39] to work on a variety of infrastructures. Additionally, BigJob has also been extended to work with data and, analogous to compute Pilots, to abstract away direct user communication between different storage systems.

BigJob (being a prototype), recently re-implemented as a production-level system named "RADICAL-Pilot" [**?**], represents one of the latest evolutionary stages of the Pilot abstraction. From an initial phase in which Pilot were implemented as ad hoc place holder machinery for a specific application, to the integration of Pilot systems within the middleware of remote resources, RADICAL Pilot implements the Pilot abstraction as an interoperable compute and data management system that can be programmatically integrated into end-user applications.

Another ongoing evolutionary trend of the Pilot abstraction is to be implemented into Pilot-based workload managers, thus moving away from providing simple Pilot capabilities in application space. 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. DIANE [43], GlideInWMS, DIRAC [12], PanDa [66], AliEn [6] and Co-Pilot [1]. Each of these Pilots serves a particular user community and experiment. Interestingly, these Pilots are functionally very similar, work on almost the same underlying infrastructure, and serve applications with very similar (if not identical) characteristics.

GlideinWMS [53] is a higher-level workload management system that is based on the Pilot capabilities of Condor GlideIn. 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. GlideinWMS attempts to hide the Pilot capabilities rather than exposing them to the user. GlideinWMS is currently deployed in production on the Open Science Grid (OSG) [46] and is the recommended mode for new users wanting to access OSG resources.

PanDA (Production and Distributed Analysis) [66] is the workload management system of the ATLAS experiment, used to run managed production and user analysis jobs on the grid. The ATLAS Computing Facility operates the pilot submission systems. This is done using the PanDA 'AutoPilot' scheduler component which submits pilot jobs via Condor-G. 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. AliEn deploys a pull-based model [51] assuming that the resource pool is dynamic and that a pull model doesn't require the broker to keep detailed track of all resources, which leads to a more simple implementation.

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

Another interesting Pilot system that is used in the LHC context is Co-Pilot [1]. Co-Pilot serves as an integration point among different grid Pilot systems (such as AliEn and PanDA), clouds, and volunteer computing resources. Co-Pilot provides components for building a framework for seamless and transparent integration of these resources into existing grid and batch computing infrastructures exploited by the High Energy Physics community.

The Pilot abstraction has also been integrated into scientific workflow systems. Pilot systems have proven an effective tool for managing the workloads executed in the various stages of a workflow. For the Pegasus project [ref], the Corral system [50] was developed to support the requirements of the Pegasus workflow system in particular to optimize the placements of Pilots with respect to their workload. It did this by serving as a front-end to Condor GlideIn. In contrast to GlideinWMS, Corral provides more explicit control over the placement and start of Pilots to the end-user. Corral was later extended to also serve as a possible front end to GlideinWMS.

Swift [63] is a scripting language designed for expressing abstract workflows and computations. The language provides also capabilities for executing external application as well as the implicit management of data flows between application tasks. Swift uses a Pilot implementation called "Coaster" [15], developed to address workload management requirements by supporting various types of infrastructure, including clouds and grids.

Swift has also been used in conjunction with Falkon [48]. Falkon was engineered for executing many small tasks on HPC systems and shows high performance compared to the native queuing systems. Falkon is a paradigmatic example of how the Pilot abstraction can be implemented to support very specific workload.

The proliferation of Pilot systems underlines not only a progressive appreciation for the Pilot abstraction but also the emergence of a Pilot paradigm to support the execution of distributed and increasingly of parallel applications. The brief description of the many Pilot system implementations introduced in this section helps to identify some distinctions in terms of design, usage and operation modes. Figure 3 is a graphical representation of this rough clustering.

The evolutionary trend of the Pilot paradigm has been largely organic and uncoordinated leading to a largely in-

consistent terminology related to the Pilot abstraction, its implementations and usages. Furthermore, a coherent understanding of the Pilot components and functionalities is still missing leading to a blurred definition of Pilot and how it should be distinguished from other abstractions and middleware and application software.

The evolution of Pilots attests to their usefulness across a wide range of deployment environments and application scenarios, but the divergence in specific functionality and inconsistent terminology calls for a standard vocabulary to assist in understanding the varied approaches and their commonalities and differences. This is what is offered in the next section of this paper.

# 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 systems. Implementation details sometimes hide the functional commonalities and differences among Pilot 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 logical components and functionalities shared by every Pilot system. The goal is to offer a paradigmatic description of a Pilot system and a well-defined vocabulary to reason about such a description and, eventually, about its multiple implementations.

## 3.1 Logical Components and Functionalities

All the Pilot systems introduced in §2 are engineered to allow for the execution of (multiple types of) workloads on Distributed Computing Infrastructures (DCIs) such as grid, cloud, or HPC facilities. This is achieved differently, depending on use cases, design and implementation choices, but also on the constraints imposed by the middleware and policies of the targeted DCIs. The common denominators among Pilot systems are defined along multiple dimensions: purpose, logical components, and functionalities.

The purpose shared by every Pilot system is to improve the (performance of) workload execution when compared to executing the same workload directly on one or more DCI. Performance in Pilot systems is usually associated to throughput and execution time to completion (TTC), but other metrics could also be considered, for example, energy efficiency, data transfer minimization, scale of the workload executed, or a mix of them. Purposes that are not (directly) performance related include robustness and ease of application deployment. In order to optimize the chosen set of metrics, each Pilot system exhibits characteristics that are both common or specific to one or more implementations. Discerning these characteristics requires isolating and defining the minimal set of logical components that has to characterize every Pilot system.

At some level, all Pilot systems leverage three separate but coordinated logical components: a **Pilot Manager**, a **Workload Manager**, and a **Task Manager**. The Pilot Manager handles the description, instantiation, and use of one or more resource placeholders (i.e. 'Pilots ') on single or multiple DCIs. The Workload Manager handles the scheduling of one or more given workloads on the available resource placeholders. Finally, the Task Manager takes care of executing the tasks of each workload by means of the resources held by the placeholders.

The implementation details of these three logical components significantly vary across Pilot systems (see §4). One or more logical components may be responsible for specific functionalities, both on application as well as infrastructure level, two or more logical components may be implemented in a single software module, or additional functionalities may be integrated into the Managers and Task Manager. Nevertheless, the Pilot and Workload Managers and the Task Manager can always be distinguished across different Pilot systems.

Each Pilot system supports a minimal set of functionalities that allow for the execution of workloads: **Pilot Provisioning**, **Task Dispatching**, and **Task Execution**. Pilot systems need to schedule resource placeholders on the targeted resources, schedule tasks on the available placeholders, and then use these placeholders to execute the tasks of the given workload.

More functionalities might be needed to implement a production-grade Pilot system: authentication, authorization, accounting, data management, fault-tolerance, or load-balancing. While these functionalities may be critical implementation details, they depend on the specific characteristics of the given use cases, workloads, or targeted resources. As such, these functionalities should not be considered a necessary characteristic of every Pilot system.

Among the core functionalities that characterize every Pilot system, Pilot Provisioning is essential because it allows for the creation of resource placeholders. As seen in §2, this type of placeholder enables tasks to utilize resources without directly depending on the capabilities exposed by the targeted DCI. Resource placeholders are scheduled onto the DCI resources by means of the DCI capabilities, but once scheduled and then executed, these placeholders make their resources directly available for the execution of the tasks of a workload.

The provisioning of resource placeholders depends on the capabilities exposed by the targeted DCI and on the implementation of each Pilot system. Typically, for DCIs adopting queues, batch systems, and schedulers, provisioning a placeholder involves it being submitted as a job. A 'job' on this kind of DCIs is a type of logical container that includes configuration and execution parameters alongside information on the executable that will be executed on the DCIs compute nodes. Conversely, for infrastructures that do not adopt a job-based middleware, a resource placeholder would be executed by means of other types of logical container as, for example, a Virtual Machine (VM) or a Docker Engine [9, 22].

Once resource placeholders are bound to a DCI, tasks need to be dispatched to those placeholders for execution. Task dispatching does not depend on the functionalities of the DCI middleware so it can be implemented as part of the Pilot system. In this way, the control over the execution of a workload is shifted from the DCI to the Pilot system. This shift is a defining characteristic of the Pilot paradigm, as it decouples the execution of a workload from the need to submit its tasks via the DCI middleware. The tasks of a workload will not individually have to wait on the DCI queues, but rather on the availability of the placeholder before being executed. More elaborate execution patterns involving task

and data interdependence can thus be implemented outside the boundaries of the DCI capabilities. Ultimately, this is why Pilot systems allow for the direct control of workload execution and the optimization, for example, of execution throughput.

Communication and coordination are two distinguishing characteristics of distributed applications and Pilot systems are no exception. The Workload Manager, Pilot Manager, and the Task Manager need to communicate to coordinate the execution of the given workload on the instantiated resource placeholders. Nonetheless, Pilot systems are not defined by any specific communication pattern and coordination strategy. The logical components of a Pilot system may communicate with every suitable pattern (e.g. one-to-one, many-to-one, one-to-many (cit) with a push or pull model) and coordinate adopting any suitable strategy (e.g. time synchronization, static or dynamic coordinator election, local or global information sharing, or master-worker). The same applies to network architectures and protocols: different network architectures and protocols may be leveraged to achieve effective communication and coordination.

As seen in §2, master-worker is a very common coordination pattern among Pilot systems. When the master is identified with the Workload Manager, and the worker with the Task Manager, the functionalities related to task description, scheduling, and monitoring will generally be implemented within the Workload Manager, while the functionalities needed to execute each task will be implemented into the Task Manager. Alternative coordination strategies, for example where a Task Manager directly coordinates the task scheduling, might require a functionally simpler Workload Manager but a comparatively more feature-rich Task Manager. The former would require capabilities for submitting tasks, while the latter would require to coordinate with its neighbor executors leveraging, for example, a dedicated overlay network. Both these systems, adopting different coordination strategies, should be considered Pilot systems.

Data management can play an important role within a Pilot system. For example, functionalities can be provided to support the local or remote data staging required to execute the tasks of a workload, or data might be managed according to the specific capabilities offered by the targeted DCI. Pilot systems can be devised in which tasks do not require any 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 outsourced to third-party systems. Being able to read and write files to a local filesystem should then be considered the minimal capability related to data required by a Pilot system. More advanced and specific data capabilities like, for example, data replication, (concurrent) data transfers, data abstractions other than files and directories, or data placeholders should be considered special-purpose capabilities, not characteristic of every Pilot system.

In the following section, a minimal set of terms related to the logical components and capabilities so far described is defined.

## 3.2 Terms and Definitions

The terms 'pilot' and 'job' are arguably among the most relevant when referring to Pilot systems. It is the case that Pilot systems are commonly referred to as 'Pilot-Job systems', a clear indication of 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 need to be clarified in order to offer a coherent terminology. Both 'job' and 'pilot' need to be understood in the context of DCIs, the infrastructures used by Pilot systems. DCIs offer compute, storage, and network resources and Pilots allow for the users to utilize those resources to execute the tasks of one or more workloads.

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

**Workload.** A set of tasks, possibly related by a set of arbitrarily complex relations.

**Resource.** Finite, typed, and physical quantity utilized when executing the tasks of a workload. Compute cores, data storage space, or network bandwidth between a source and a destination are all examples of resources commonly utilized when executing workloads.

**Infrastructure or DCI.** Structured set of resources, possibly geographically and institutionally separated from the users that utilize those resources to execute the tasks of a workload [ref]. Infrastructures can be logically partitioned, with a direct or indirect mapping onto individual pools of hardware (i.e. clusters, systems, and supercomputers).

As seen in §2, most of the DCIs leveraged by Pilot systems utilize 'queues', 'batch systems' and 'schedulers'. In such DCIs, jobs are scheduled and then executed by a batch system.

**Job.** Functionally defined as a 'task' from the perspective of the DCI, but in the case of a Pilot system indicative of the type of container required to acquire resources on a specific infrastructure.

When considering Pilot systems, jobs and tasks are functionally analogous but qualitatively different. Functionally, both jobs and tasks are containers – i.e. metadata wrappers around one or more executables often called 'kernel', 'application', or 'script'. Qualitatively, 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 schedule tasks on a certain infrastructure. It should be noted that, given their functional equivalence, the two terms can be adopted interchangeably when considered outside the context of Pilot systems. Indeed, workloads are encoded into jobs when they have to be directly executed on infrastructures that support or require that type of container.

As described in §3.1, a resource placeholder needs to be submitted to the target DCI wrapped in the type of container supported by that specific DCI. For example, for a DCI exposing a HPC or grid middleware [cit, cit], a resource placeholder needs to be wrapped within a 'job'. For other type of DCIs, the same resource placeholder will need to be wrapped within a different type of container as, for example,

a VM or a Docker Engine. For this reason, the capabilities exposed by the job submission system of the target DCI determine the submission process of resource placeholders and how or for how long they can be used. For example, when wrapped within a 'job', placeholders are provisioned by submitting a job to the DCI queuing system, become available only once the job is scheduled on the resources out of the queue, and is available only for the duration of the job lifetime.

A Pilot is a resource placeholder. As a resource placeholder, a Pilot holds portion of a DCI's resources for a user or a group of users, depending on implementation details. A Pilot system is a software capable of creating Pilots so to gain exclusive control over a set of resources on one or more DCIs and then to execute the tasks of one or more workloads on those Pilots.

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

It should be noted that the term 'Pilot' as defined here is named differently across Pilot 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 are, in practice, used as synonyms without properly distinguishing between the type of container and the type of executable that compose a Pilot. This is a clear indication of how necessary the minimal and consistent vocabulary offered here is when reasoning analytically about multiple Pilot system implementations.

The term 'Pilot-Job' is often used to identify a Pilot system too. This is an unfortunate choice as the term 'job' identifies just the way in which a Pilot is provisioned on a DCI exposing specific capabilities, not a general property of all the Pilot systems. The use of the term 'Pilot-Job system' should therefore be regarded as a historical artifact, viz., the targeting of a specific class of DCIs in which the term 'job' was, and still is, meaningful. With the development of new types of DCI middleware as, for example, cloud infrastructures, the term 'job' has become too restrictive, a situation that can lead to terminological and conceptual confusion.

We have now defined resources, DCIs and Pilots. We have established that a Pilot is a placeholder for a set of resources. When combined, the resources of multiple Pilots form a resource overlay. The Pilots of a resource overlay can potentially be distributed over multiple resources and/or DCIs.

**Resource Overlay.** The aggregated set of resources of multiple Pilots.

Three more terms associated with Pilot systems (cit, cit, cit) need to be explicitly defined: 'Multi-level scheduling', 'early binding', and 'late binding'.

Pilot systems are said to implement multi-level scheduling because they require the scheduling of two types of entities: Pilots and tasks. A portion of the resources of a DCI is allocated to one or more Pilots in the form of containers supported by that DCI, and the tasks of a workload are dispatched for execution to those Pilots. This is a fundamental feature of Pilot systems because (i) a potentially faster and more flexible execution of workloads is achieved by avoiding the overhead imposed by a centralized job management system shared among multiple users; and (ii) the tasks of a workload can be bound to a set of Pilots before or after it becomes available on a remote resource. (Depending on the implementation of a Pilot system, there can be another level of scheduling, where the Pilot makes scheduling decisions about task placement within the Pilot's resource allocation.)

The simplification obtained as a consequence of bypassing the job submission system of the DCI is one of the main reasons for the success of Pilot systems. As mentioned in §3.1, the tasks of a workload can be executed on a Pilot without waiting in the queuing system of the given infrastructure, and as a result, increasing the throughput of the workload execution. Moreover, Pilots can be reused to execute multiple workloads until the Pilots' walltime expire. It should be noted that how tasks are actually 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 type of binding of tasks to Pilots depends on the state of the Pilot. A Pilot is inactive until it is executed on a DCI, is active thereafter, until it completes (or fails). Early binding indicates the binding of a task to an inactive Pilot; late binding indicates the binding of a task to an active Pilot. Early binding is potentially useful to increase the information about which Pilots can be 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. Additionally, in case of early binding, other type of decisions related to the workload could be made, e.g., the transfer of data to a certain resource while the Pilot is still inactive. Late binding is critical in assuring the aforementioned high throughput of the distributed application by allowing sustained task execution without additional queuing time or container instantiation time.

It should be noted that some aspects of early binding can also be achieved without a Pilot system, but, importantly, the Pilot paradigm allows to do both, even within the confinements of a single workload.

**Multi-level scheduling.** Scheduling Pilots onto resources, and scheduling tasks onto (active or inactive) Pilots.

**Early binding.** Binding one or more tasks to an inactive Pilot.

**Late binding.** Binding one or more tasks to an active Pilot.

Table 1 offers an overview of the defined minimal and consistent vocabulary alongside its mapping into both the logical components of a Pilot system and its required minimal set of functionalities as defined in §3.1. The same mapping is diagrammatically represented in Figure 4.

Relevant to note how scheduling happens both at the infrastructure and at the pilot level. Accordingly, in Table 1 the term 'multi-level scheduling' pertains to both the pilot and workload managers and their respective pilot provisioning and task dispatching functionalities. In Figure 4, scheduling clearly happens at the level of the resource pool of an infrastructure, for example by means of a cluster scheduler, and then of the pilot once it holds its resources.

Table 1 and Figure 4 help also to appreciate the critical distinction between the container of a pilot and the pilot itself. A container, for example a job, is used by the pilot manager to provision the pilot. Once the pilot has been

| Term | Functionality | Logical Component |
|---|---|---|
| Workload | Task Dispatching | Workload Manager |
| Task | Task Dispatching<br>Task Execution | Workload Manager<br>Task Manager |
| Resource | Pilot Provisioning | Pilot Manager |
| Infrastructure or DCI | Pilot Provisioning | Pilot Manager |
| Job | Pilot Provisioning | Pilot Manager |
| Pilot | Pilot Provisioning<br>Task Execution | Pilot Manager<br>Task Manager |
| Multi-level scheduling | Pilot Provisioning<br>Task Dispatching | Pilot Manager<br>Workload Manager |
| Early binding | Task Dispatching<br>Pilot Provisioning | Workload Manager<br>Pilot Manager |
| Late binding | Task Dispatching<br>Pilot Provisioning | Workload Manager<br>Pilot Manager |

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

provisioned, it is the pilot and not the container to be responsible of both holding a set of resources and offering the functionalities of the task manager.

Figure 4 should not be confused with an architectural diagram. No indications are given about the interfaces that should be used, how the logical component should be mapped into software modules, or what type of communication and coordination protocols should be used among such components. This is why no distinction is made diagrammatically between early and late binding. Their difference is temporal and, as such, it can be highlighted only when describing the succession of the states and operations of a Pilot system implementation. In the early case, the binding of the tasks to a pilot will happen before the submission of such a Pilot to the remote infrastructure. In the late case, the binding will happen once the pilot has been instantiated and holds already its resources.

The wide spectrum of available implementations of the logical components of a Pilot system is explored in the next section.

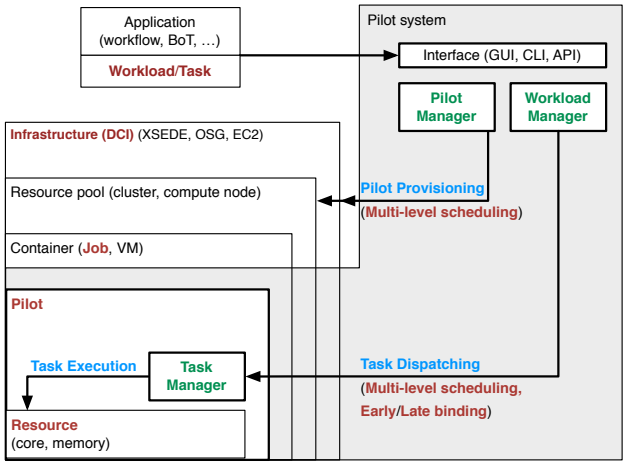


**Figure 4: Diagrammatic representation of the logical components, functionalities, and core vocabulary of a Pilot system. The terms of the core vocabulary are highlighted in red, those of the logical components of a Pilot system in green, and those of their functionalities in blue.**

# 4. PILOT SYSTEMS: ANALYSIS AND IMPLEMENTATIONS

Section §3 offered two main contributions: (i) the minimal sets of logical components and functionalities of Pilot systems [3]; (ii) and a well-defined core terminology to support reasoning about such systems. The former defines the necessary and sufficient requirements for a software system to be a Pilot system, while the latter enables consistency when referring to different Pilot systems. Both these contributions are used in this section to review critically a selection of Pilot systems implemented to execute real-life scientific workloads.

The goal of this section is twofold. Initially, the Pilot functionalities presented in §2 are used as the basis to infer core Pilot implementation properties. Auxiliary properties are also defined when useful for a critical comparison among different Pilot systems. Subsequently, several Pilot systems are analyzed and then clustered around the properties previously defined. In this way, insight is offered about how to choose a Pilot system based on functional requirements, how Pilot systems are designed and engineered, and the design principles that underly such systems.

[3]From here on we will be internally consistent and use Pilot system instead of Pilot-Job system.

## 4.1 Core and Auxiliary Properties

This section analyzes the properties of diverse implementations of a Pilot system. Two sets of properties are introduced: core and auxiliary (see Table 2). Both sets of properties are chosen by considering the implementation requirements of the Pilot capabilities as defined in §2, Table 1. A property belongs to the set of core properties if it is required by the implementation of one or more of the Pilot functionalities. Auxiliary properties are instead required only when added functionalities needs to be implemented for a specific Pilot system. As such, auxiliary properties are not necessarily shared among all Pilot systems.

Table 2 offers a brief description for each core and auxiliary property alongside their mapping to the required components and functionalities of a Pilot system as described in §3.1.

The core implementation properties of the Pilot Manager and its Pilot Provisioning functionality are Pilot Resources, Pilot Deployment, and Infrastructure Interaction. Pilot Resources identifies the type of resources that the Pilot system exposes, e.g. compute, data, or networking. Pilot Deployment describes the modalities of scheduling, utilization, and aggregation of pilots. Infrastructure Interactions singles out the protocols and policies required for the Pilot system to interact with the remote infrastructure(s). For example, to schedule a pilot onto a specific infrastructure Pilot systems need to know what type of container to use (e.g. job, virtual machine), what type of scheduler the resource exposes, the amount of cores and for how long they can be asked for, what filesystems can be accessed, how to contact the remote resource, but possibly also the type of cores, the amount of available memory and storage and whether a low-latency interconnect is available.

The core implementation properties of a Workload Manager and the Task Dispatching functionality are Workload Semantics and Workload Binding. Semantically, a workload description contains all the information necessary for it to be dispatched to the appropriate resource. For example, type, number, size, and duration of tasks alongside their grouping into stages or their data dependences need to be known 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, the core implementation properties of a Task Manager and its Task Execution functionality are Workload Execution and Infrastructure Interaction. Workload Execution identifies the process of task dispatching and the modalities with which the execution environment is set up for each task. The dispatching process may also depend on the capabilities exposed by the target infrastructure and its policies. As such, the execution of the workload may depend on how the Pilot systems has to interact with the remote infrastructure.

Several auxiliary properties play a fundamental role in distinguishing Pilot systems implementations, as well as in defining their usability. Programming and user interfaces; interoperability across differing middleware and other Pilot systems; multitenancy; strategies and abstractions for data management; security including authentication, authorization, and accounting; 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 implementation but that, in of themselves, would not distinguish a Pilot as a unique system.

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

### 4.1.1 *Core properties*

Following is an in depth description of core properties characterizing each implementation of a Pilot system. This list of properties is minimal and complete. Note that these are the properties of Pilot implementations, and not of individual instantiations of Pilots.

- **Pilot Resources**. Usually, pilots expose compute resources but, depending on the capabilities offered by the infrastructure where the pilot is instantiated, pilots might also expose data and network resources. Some of the typical characteristics of pilots resources are: size (e.g. number of cores), lifespan, intercommunication (e.g. low-latency or inter-domain), computing platforms (e.g. x86, or GPU), file systems (e.g. local, shared, or distributed). The coupling between pilot and the resources that it holds may vary depending on the architecture of the resources in which it is instantiated. For example, a pilot may bind multiple compute nodes, single nodes, or portion of the cores of each node. The same applies to file systems and its partitions or to software defined or physical network resources.

- **Pilot Deployment**. Pilots are scheduled and then bootstrapped on the remote infrastructures. The characteristic of both operations varies depending on both the implementation details of the Pilot systems and the architecture, interfaces, and capabilities offered by the remote infrastructures. For example, Pilot scheduling may be fully automated or directly controlled by applications and end-users. The bootstrapping can offer explicit or implicit localization by allowing to load specific libraries, compilers, and support software. Both scheduling and bootstrapping varies depending on whether the remote infrastructures expose HPC, grid, or cloud interfaces.

- **Workload Semantics**. The tasks of a workload are dispatched to Pilots depending on the workload semantics. Specifically, dispatching decisions depends on the temporal and spatial relationships among tasks, the affinity between data and compute resources required by the tasks, and the type of capabilities needed for their execution. Pilot systems support a varying degree of semantic richness for the workload and its tasks. The (standard) format or language in which the workloads are described may also be relevant.

- **Task Binding**. The Task Dispatching functionality implies the capability of binding tasks to Pilots. Without such a capability, it would not be possible to know

| | Property | Component | Functionality | Description |
|---|---|---|---|---|
| *Core* | Pilot Resources | Pilot Manager | Pilot Provisioning | Types and capabilities of the resources held by the pilot. |
| | Pilot Deployment | Pilot Manager | Pilot Provisioning | Modalities and protocols for pilot scheduling, utilization, and aggregation. |
| | Workload Semantics | Workload Manager | Task Dispatching | The specification of the semantics among tasks captured in the workload description. |
| | Workload Binding | Workload Manager | Task Dispatching | Modalities and policies for binding tasks to pilots. |
| | Workload Execution | Task Manager | Task Execution | Types of tasks and the mechanisms to execute tasks. |
| | Infrastructure Interaction | Pilot Manager<br>Task Manager | Pilot Provisioning<br>Task Execution | Modalities and protocols used to coordinate the pilot system/infrastructure interaction. |
| *Auxiliary* | Architecture | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | Frameworks and architecture that the components and their whole are build with. |
| | Coordination and Communication | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | The interaction between the components of the system. |
| | Interface | Pilot Manager<br>Workload Manager | Pilot Provisioning<br>Task Dispatching | Interface that the user can use to interact with the system. |
| | Interoperability | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | Interoperability between Pilots on multiple DCIs. |
| | Multitenancy | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | The use of components by multiple (simultaneous) users. |
| | Resource Overlay | Pilot Manager<br>Workload Manager | Pilot Provisioning<br>Task Dispatching | The aggregation of resources from multiple pilots into overlays. |
| | Robustness | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | The measures in place to increase the robustness of the components and the whole. |
| | Security | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | AAA considerations for the components and the whole. |
| | Files and Data | Pilot Manager<br>Workload Manager | Pilot Provisioning<br>Task Dispatching | The mechanisms that the system offers to explicitly deal with files and data. |
| | Performance and Scalability | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | A description of scale and limitations and measures to reach that. |
| | Development Model | Pilot Manager<br>Workload Manager<br>Task Manager | Pilot Provisioning<br>Task Dispatching<br>Task Execution | The development and support model for the software. |

**Table 2: Mapping of the Properties of Pilot system implementations onto the components described in §3.1.**

where to dispatch tasks, Pilots could not be used to execute tasks and, as such, the whole Pilot system would not be usable. As seen in §3, Pilot systems may allow for two types of binding between tasks and Pilots: early binding and late binding. Pilot system implementations differ in whether and how they support these two types of binding. Specifically, while there might be implementations that only support a single type of binding, 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. Besides the binary decision between early and late binding, the Pilot system can expose, for example, more detailed application-level scheduling decisions, dispatch policies, or even include more levels of scheduling.

- **Task Execution**. Once the tasks are dispatched to a Pilot, their execution may require for a specific runtime environment to be set up (e.g. MPI). Pilot systems differ in whether and how they offer such a capability. Pilot systems may adopt dedicated components for managing execution environments, or they may rely on ad hoc configuration of the Pilots. Furthermore, execution environments can be of varying complexity, depending on whether the Pilot system allows for data retrieval, software and library installations, communication and coordination among execution environments and Pilots.

- **Infrastructure Interaction**. Pilot systems interact with remote infrastructures at multiple levels. The degree of coupling between the Pilot system and the infrastructure can vary as much as the information shared between them. Depending on the capabilities implemented, Pilot systems have to negotiate the scheduling on Pilots, may be staging data in and out of the infrastructure, and may have to mediate task binding and execution by means of remote interfaces and protocols.

#### *4.1.2 Auxiliary properties*

- **Architecture**. Pilot systems may be implemented by means of different type of architectures (e.g service-oriented, client-server, or peer-to-peer). Architectural choices may depend on multiple factors, including application use cases, deployment strategies, or interoperability requirements. The analysis and comparison of architectural choices is here limited to the trade-offs implied by such a choice, especially when considering how they affect the Core Properties.

- **Communication and Coordination**. Communication and coordination are features of every distributed system. In 3.1 it was suggested that Pilot systems are not defined by any specific communication and coordination pattern or protocol. The details of communication and coordination among the Pilot system components are distinguishing implementation properties.

- **Interface**. Pilot systems may present several types of private and public interfaces: among the components of the Pilot system, between the application and the Pilot system, or between end users and one or more programming language interfaces for the Pilot system.

- **Interoperability**. Interoperability is defined as the capability to deploy Pilots on heterogeneous infrastructures. It allows for a Pilot system to provision pilots and execute workloads on different types of infrastructure (e.g. HTC, HPC, Cloud but also Condor, LSF, Slurm, or Torque).

- **Multitenancy**. Pilot systems 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 system. When available at local level, multiple users may share the same pilot.

- **Resource Overlay**. The resources of multiple Pilots may be aggregated into a resource overlay. Overlays may be directly exposed to the application layer and to the end-users depending on the public interfaces and usability models. Overlays may abstract away the notion of Pilot or offer an explicit semantic for their aggregation, selection, and management.

- **Robustness**. Used to identify those properties that contribute towards the resilience and the reliability of a Pilot system. 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 system implementation, and the type of support offered to the relevant use cases.

- **Security**. The properties of Pilot systems related to security would require a dedicated analysis. The analysis here is limited to authentication, authorization paradigms. The scope of the analysis is further constrained by focusing only on those elements that impact the Core Functionalities as defined in §3.1.

- **Data**. As discussed in Section 3.1, only basic data reading/writing functionalities are minimally necessary for a Pilot system. Nonetheless, most of the use cases [ref] require more advanced data management functionalities that can be implemented within the Pilot system or delegated to third party tools.

- **Performance and scalability**. Pilot systems vary both in terms of overheads they add to the execution of a given workload, and of the size and duration of the workloads a user can expect to be supported. Furthermore, Pilot systems can be designed to optimize one or more performance metrics, depending on the targetted use cases.

- **Development Model**. The model used to develop Pilot systems is a distinguishing element, especially when considering whether the development is supported by a open community or by a specific project. Different development models have an impact on the life span of the Pilot system, its maintainability and, in case, evolution path.

| **Pilot System** | *Pilot Resources* | *Resource Interaction* | *Pilot Deployment* | *Workload Semantics* | *Task Binding* | *Task Execution* |
|---|---|---|---|---|---|---|
| DIANE | HTC | GANGA | Out-of-Band / explicit | Programmable | Late | Serial |
| DIRAC | HTC | Custom | Community Service / implicit | None (Data dependencies?) | Late | Serial, some MPI |
| Falkon | HPC | Unspecified | Web Service | None | Late (mixed push/pull) | Serial |
| HTCondor | HTC (and to some degree HPC) | Condor-G | Explicit in Glidein case | Graph | Late | All |
| MyCluster | HPC | Custom (SGE / PBS / HTCondor) | CLI tools from respective LRMS | Workload semantics from respective LRMS | Agnostic | All |
| PanDA | HTC | Custom, SAGA | Community Service / implicit | Task type, priority | Late | Serial, some MPI |
| RADICAL-Pilot | HPC | SAGA | Programmable / explicit | Programmable | Early & Late | Serial & MPI |

**Table 3: Overview of Pilot systems and a summary their core properties.**

## 4.2 Analysis of Pilot system Implementations

In light of the common vocabulary discussion in §3.2, a representative set of Pilot systems has been chosen for further analysis. Examining these Pilot systems using the common vocabulary exposes their core similarities, and allows a detailed analysis of their differences.

The now following discussion of Pilot systems is ordered alphabetically. To assist the reader, we make use of textual conventions: in **Bold** we express the **Logical Components** and **Functionalities** of our model from §3.1 and the **Terms and Definitions** from §3.2, in *Italic* we refer to the *Properties* from §4.1 and in `Typewriter` we display terminology from the respective Pilot system under discussion.

### 4.2.1 DIANE

DIANE [43] is a task coordination framework, which follows the master-worker pattern. It was developed at CERN for data analysis in the LHC experiment, but has since been used in various other domains, though mainly in the Life Sciences. As DIANE is a software framework, some of its semantics are not predetermined, as they can be implemented in various ways depending on the user requirements. In this way, DIANE also provides Pilot functionality for job-style executions.

#### *Resource Interaction.*

The Pilot provisioning with GANGA [21] provides a unified interface for job submissions to various resource types. In this way, DCI specifics are hidden from other components of DIANE as well as from the user.

#### *Overlay Management.*

The *Overlay Management* of Pilots is done out-of-band by means of the `diane-submitter` script which can launch a Pilot on any supported infrastructure.

#### *Workload Semantics.*

DIANE is mainly a master-worker framework and, as such, the **Tasks** are fully independent, consistently with a Bag-of-Tasks **Workload Semantics**. The **Workload** that is managed by the application-specific component of DIANE can freely structured, as long as it can be translated to independent tasks[**?**, **?**]. Plugins for other types of workload (e. g. DAGs or for data-intensive applications) exist or are under development. The framework is extensible: for example, applications can implement a custom application-level scheduler.

#### *Coordination and Communication.*

The *Coordination & Communication* in DIANE is based on CORBA [45] and uses TCP/IP. The CORBA layer is invisible to the application layer. Networking-wise, the workers are clients of the master server. On TCP/IP level, communication is always unidirectional from the `WorkerAgent` to the `RunMaster`. *Security* is provided by a secret token that the `WorkerAgent` needs to communicate back to the `RunMaster`. This implies that the `WorkerAgent` needs to be able to reach the `RunMaster` via TCP/IP but not the other way around. Bidirectional communication is achieved by periodic polling through heartbeats by the `WorkerAgent`, where the `RunMaster` responds with feedback.

#### *Task Binding.*

DIANE is primarily designed for HTC environments (such as EGI [19]), i. e. one Pilot consists of a single worker agent with the size of 1 core. Although the semantics of the binding are ultimately controllable by the user-programmable scheduler, the general architecture is consistent with a pull model. The pull model naturally implements the late-binding paradigm as every worker pulls a new task once it is available and has free resources.

#### *Task Execution Modes.*

DIANE is primarily designed for HTC environments (such as EGI [19]), i. e. one Pilot consists of a single worker agent with the typical size of 1 core/node. As such, DIANA is not

able by default to run (widely) parallel applications as those based, for example, on MPI.

*Pilot Resource Capabilities.*
DIANE implements the notion of "capacity", a property that applications can use to diverge from the "1 worker = 1 core" mode.

*Architecture.*
The central component of DIANE is the `RunMaster`, consisting of a `TaskScheduler` and an `ApplicationManager` module. Both modules are abstract classes that need to be implemented for the specific purpose at hand. Examples and/or default implementations are provided to the user as guidelines and initial code stubs. Together the `TaskScheduler` and `ApplicationManager` implement the **Workload Manager** component.

The `TaskScheduler` keeps track of the task entries and is responsible for **Binding Tasks** to the `ApplicationWorkers`. The implementation of the `ApplicationManager` is responsible for defining the application **Workload** and for describing the **Tasks** that are passed to the **TaskScheduler**. As the `ApplicationWorker` asks the `TaskScheduler` for new **Tasks**, the natural way of implementing the scheduler is by following a **Late Binding** approach.

**Pilots** in DIANE are `WorkerAgents`. The core component of a `WorkerAgent` is the `ApplicationWorker`, an abstract class that defines three methods that need to be implemented by every implementation. Two of these methods are for initialization and cleanup while the third method, (`do_work()`) receives the task description and executes the work. `do_work()` is therefore an implementation of the **Task Manager** component.

*Interface.*
DIANE's *Architecture* is based on the `Inversion of Control` design pattern. Implemented in Python, it exposes well-defined hooks for **Application** programming. The aforementioned abstract classes need to be implemented to provide the application-specific semantics. The *Interface* to these "DIANE-**Applications**" is to start them through the `diane-run` command. As discussed in the *Overlay Management*, the creation of Pilots is done out-of-band.

*Performance and Scalability.*
The authors report that in first instance they had implemented full bi-directional communication, but that turned out to be difficult to correctly implement and created scalability limitations.

*Robustness.*
The *Robustness* in DIANE comes from the mature CORBA communication layer, and from the custom task-failure policies implemented in the `TaskScheduler`. *Robustness* is achieved in DIANE by: (i) offering mechanisms to supports fault tolerance: basic error detection and propagation mechanisms are in place; (ii) implementing an automatic re-execution of tasks; and (iii) leaving room for application specific strategies.

*Files and Data.*
The CORBA communication channel between the Master and the Worker also allows for file transfers, meaning that the Pilot not only exposes cores, but also the file system on the worker nodes.

*Interoperability and Security.*
*Interoperability* to various middleware security mechanisms (e. g. GSI, X509 authentication) and backends is achieved through its integration with GANGA.

*Multitenancy.*
DIANE is a single-user Pilot system, i. e. each Pilot is executed with the privileges of the respective user. Also, only workload of this respective user can be executed by DIANE. However, *Multitenancy* with DIANE does happen when it is used as a backend for Science Gateways, where it serves multiple users on the same installation, but with a single credential.

*Development Model.*
DIANE is developed, maintained, and used by the CERN community with external contributions and users.

### 4.2.2 DIRAC

DIRAC (Distributed Infrastructure with Remote Agent Control) is a software product developed by the CERN LHCb project[**?**]. DIRAC implements a `Workload Management System` (WMS) to manage the computational payload of its community members, and the DIRAC Pilot framework to execute that payload.

DIRAC's claim to fame is that "[. . . ] the LHCb data production run in summer 2004 was the first successful demonstration of the massive usage of the LCG grid resources. This was largely due to the use of the DIRAC Workload Management System, which boosted significantly the efficiency of the production jobs." [**?**]

*Pilot Resource Capabilities.*
The bootstrapping of the `DIRAC Agent` performs a full DIRAC installation including a download of the most current version of the configuration files. After completion of the bootstrapping, the installer checks for the working conditions (e.g. where execution takes place, available resources, etc.) and then a `DIRAC Agent` is started.

*Resource Interaction.*
Pilots are sent to the resource by the `Pilot Director` and all resource interaction is hidden behind that abstraction . The Pilot Director is reported to be able to interact with different batch systems like PBS/Torque, LSF, SGE and BQS.

*Overlay Management.*
Pilots are being launched by the `Pilot Directors` to the supported DCIs. The DIRAC Pilots create a resource overlay that presents a homogeneous layer to the other components. The user has no control over where Pilots are started.

*Workload Semantics.*
The user **Workload** (in DIRAC terminology: `payloads`), consisting of independent **Tasks** have varying levels of priorities for LHCb as a whole, as well as a variety of requirements for them to execute. Additionally, users can specify require-

ments like needed input data for their tasks and this will be taken into account when placing the task on a resource.

*Task Binding Characteristics.*
Pilots implement a pull scheduling paradigm. Once they are running on the computing resource, they contact central WMS servers for a late binding of the resource to the payload.

When **Tasks** are assigned to the `WMS`, a consistency check is performed and optional input data requirements are translated to resource candidates.

Once this is done, **Tasks** are placed into `TaskQueues`. `TaskQueues` are sorted groups of `Payloads` with identical requirements (e.g., user identity, number of cores, etc.) that are ready to be executed.

`TasqQueue Directors` direct the workload from the `TaskQueues` for execution to the Job Agents.

*Task Execution Modes.*
Once the **Task** has been pulled by a pilot, it is executed by means of a `Job Wrapper`. Given the nature of the resources DIRAC is aimed at, **Task** are generally single or few-core tasks. Work on supporting MPI is ongoing[].

*Coordination and Communication.*
The `DIRAC Agent` is responsible for sending the `payload` request to the central DIRAC WMS server and for the subsequent execution of the received `payload`. The client/service communication is performed with a light-weight protocol (XML-RPC) with additional standard-based authentication mechanisms.

*Architecture.*
The design of DIRAC relies on three main components: the `DIRAC Agent` `TaskQueue`, and `TaskQueue Director`.

*Interface.*
The DIRAC project provides many command line tools as well as a comprehensive Python API to be used in a scripting environment. In DIRAC3 a full-featured graphical Web Portal was developed.

*Interoperability.*
DIRAC is not inter-operable with other Pilot systems but supports execution of Pilots on a variety of computing resources by means of the `Pilot Director`.

*Multitenancy.*
The community nature of the `DIRAC WMS` makes it an intrinsically multi-user system. The WMS holds the workload for multiple users and the pilot agents are shared amongst the workload of multiple user.

*Robustness.*
The `Job Wrapper` creates a uniform environment to execute **Tasks** independent of the **DCI** where they run. Furthermore, it retrieves the input sandbox, checks availability of required input data and software, executes the payload, reports success or failure of the execution, and finally uploads output sandbox and output data if required.

At the same time it also instantiates a `Watchdog` to monitor the proper behavior of the `Job Wrapper`. The watchdog checks periodically the situation of the `Job Wrapper`, takes actions in case the disk or available CPU is about to be exhausted or the payload stalls, and reports to the central `WMS`. It can also execute management commands received from the central service, e.g. to terminate the `Payload` execution.

*Security.*
The DIRAC Secure client/service framework called `DISET` (DIRAC SEcure Transport) is full-featured to implement efficient distributed systems in the grid environment[]. It provides X509 based authentication and a fine-grained authorization scheme. Additionally, the security framework includes logging service that provides history up to 90 days to investigate security incidents.

*Files and Data.*
Tasks within DIRAC that specify their input data are specifically managed. The WMS ensures that tasks are only started on resources where that data is available and it makes sure that that data becomes available.

*Performance and Scalability.*
DIRAC has well documented numbers about scaling in supporting many users, on many resources, for a variety of applications over a long period of time.

*Development Model.*
DIRAC is actively development and used by the LHCb community[]. More recently, they have been reaching out to other communities too[]. From a development perspective, there is ongoing work on enriching the data capabilities[] and to be able to execute MPI applications[].

### 4.2.3 Falkon

The Fast and Light-weight tasK executiON framework (Falkon) [48] was created with the primary objective of enabling many independent tasks to run on large computer clusters (an objective shared by most Pilot systems). 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.

*Pilot Resource Capabilities.*
Falkon exposes resource as a set of cores as tasks are by definition single core only.

*Resource Interaction.*
Falkon was originally developed for use on large computer clusters in a grid environment, but has since been expanded to work on other types of infrastructures. Falkon has been shown to run on TeraGrid (now XSEDE), TeraPort, Amazon EC2, IBM Blue Gene/L, SiCortex, and Workspace Service [48].

*Overlay Management.*
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.

*Workload Semantics.*
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.

The Swift parallel programming system [63] was integrated with Falkon for the purpose of task dispatching.

*Task Binding Characteristics.*
Cores and tasks are considered homogeneous. Tasks are pulled to the nodes and are thereby of late-binding nature.

*Task Execution Modes.*
Falkon does not support MPI jobs, a limiting factor in its adoption for certain scientific applications.

*Coordination and Communication.*
Interaction between the components is by use of the Globus Web Services model.

*Architecture.*
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 terminology defined in Section 3.2, 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.

Further, in order to process more complex workflows, Falkon has been integrated with the Karajan workflow execution engine [60]. This integration allows Falkon to accept more complex workflow-based scientific applications as input to its Pilot-like job execution mechanism.

*Interface.*
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.

*Interoperability.*
For launching to resources Falkon relies on the availability of GRAM4. This abstracts the resource details but also limits the use of other resources.

*Multitenancy.*
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.

*Robustness.*
Falkon supports a fault tolerance mechanism which suspends and dynamically readjusts for host failures.

*Security.*
Thanks to the use of Globus transport channels, Falkon allows for both encrypted and non-encrypted operation. The non encrypted version was used to gain most throughput, obviously lowering the security of real-world use.

*Files and Data.*
The data management capabilities of Falkon extend beyond the core Pilot-Job functionalities as described in Section 4.1.1. Falkon encompasses advanced data-scheduling and caching but also 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 [49].

*Performance and Scalability.*
As previously stated, the design of Falkon was centered around the goal of providing support to run efficiently large numbers of jobs 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 developed 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 [48].

Falkon has been tested for throughput and performance with multiple applications: fMRI (medical imaging), Montage (astronomy workflows), and MolDyn (molecular dynamics simulation). Falkon shown favorable results in terms of overall execution time when compared to GRAM and GRAM/Clustering methods [48].

The per task overhead of Falkon execution has been shown to be in the millisecond range. Furthermore, Falkon has been demonstrated to achieve throughput in the range of hundreds to thousands of tasks per second for very fine-grained tasks.

*Development Model.*
The Falkon project ran from 2006 to 2011 and the source code is not publicly available.

### 4.2.4 HTCondor

HTCondor can be considered one of the most prevalent distributed computing projects of all time in terms of its pervasiveness and size of its user community. Is often cited as the project that introduced the Pilot concept[]. Similar in many respects to other batch queuing systems, HTCondor puts special emphasis on high-throughput computing (HTC) and opportunistic computing. HTC is defined as providing large scale computational power on a long term scale. Opportunistic computing is about making pragmatic use of computing resources whenever they are available, without requiring full availability. The former can be can be achieved by applying the latter.

*Architecture.*

HTCondor is a high-throughput distributed batch computing system.

The core components of HTCondor are collectively named the `Condor Kernel`: `Agent` (also referred to as `schedd`), `Resource` (also referred to as `startd`), `Shadow`, `Sandbox` (also referred to as `starter`), and `Matchmaker` (also referred to as `central manager`).

The user submits **Tasks** (in HTCondor called `Jobs`) to the `Agent`. `Agents` provide persistent storage for the user's `Jobs` and tries to find computing **resources** `Resource` to execute them.

Both `Agents` and `Resources` advertise themselves to the `Matchmaker`, which is then responsible to match the latter to the former. The `Matchmaker` notifies the `Agent` of a potential match, and the `Agent` is in turn responsible to contact the `Resource` and validate the suggested match.

Once the match is confirmed by the `Resource`, both `Resource` and `Agent` have to start a new process to make the execution of the `Job` happen. The `Agent` starts a `Shadow` which is responsible for providing all the details required to execute a `Job` (the executable, the arguments, the environment, the input files).

The `Resource` creates a `Sandbox` which is an appropriate execution environment (the `Sand` for the **Task** while at the same time protecting the `Resource` from a misbehaving `Job` (the `Box`).

Following on from this, an `Agent`, `Matchmaker` and a (set of) `Resources` together form a `Pool`. **Resources** in a pool can span a wide spectrum of systems. While some `Pools` are comprised of regular desktop PCs (sometimes collectively called a campus grid), other `Pools` incorporate large HPC clusters and cloud resources. Hybrid pools with heterogeneous types of resources are also possible.

Within the `Pool` the `Matchmaker` is responsible for enforcing the policies defined by the `Community`.

While technically a `Pool` could grow as big as needed, reality was that (which was also only natural) many independent `Pools` were created. As a user (or `Agent`) could only belong to one `Matchmaker`, the `Resources` that a user could access were therefore limited to one `Pool`. This lead to the concept of `gateway flocking` [20]. With this, an individual `Pool` would be interfaced to another `Pool` by letting associated `gateway` machines expose information about the `Pool's` participants to another `Pool`, e.g. about idle `Resources` or `Agents`. These `gateway` nodes can be configured to adhere to policies and are not necessarily bi-directional.

In this model, however, `flocking` is still bounded to single `organizations/communities`, and a user (or `Agent`) could not join multiple of them at the same time. This limit was overcome by introducing `direct flocking` in which an `Agent` can report itself to multiple `Matchmakers` While both forms of `flocking` had their *raison d'être*, and they could even be mixed, `gateway` flocking did not stand the test of time as with scaling up, both technical overhead and organizational burden became too high.

Around that time (1998) the vision of the grid began to surface. One of the initial activities was a uniform interface for batch execution. This was achieved by the Globus project with the design of the Grid Resource Access and Management (GRAM) protocol.

The HTCondor `Agent` was extended to speak GRAM and Condor-G [24] was born. GRAM services are deployed as remote job submission endpoints on top of HPC cluster batch queuing systems. Condor-G allows a user (an `Agent`) to incorporate those HPC resources temporarily to an HTCondor pool.

Although Condor-G opened up the execution of Condor `Jobs` at any GRAM-enabled computing resource, the drawback of this approach was that resource allocation and job execution became coupled again, i.e. the `Agent` had to direct the `Job` to a specific GRAM endpoint without (potentially) knowing about the availability of resources on that endpoint. Because GRAM was an abstraction of so many batch queuing systems, it could only implement the greatest common denominator of all the functionalities of these various implementations. This implied that in this mode of operation there was no `Matchmaker` as the required functionality at the endpoints for that was not provided by GRAM.

`Gliding In` was engineered to undo this coupling by offering a personal Condor `Pool` out of GRAM enabled resources. `Gliding In` consist of a three-step procedure: (i) a Condor-G agent uses the GRAM protocol to launch HTCondor servers ad hoc via a GRAM endpoint service on a remote system; (ii) the Condor servers, once gotten through the batch queue, connect to the `Matchmaker` started by the `Agent` and thereby form a personal Condor `Pool`; and (iii) `Jobs` submitted by the user to the `Agent` which are then matched to and executed on the `Resources` within the `Pool`.

### *Pilot Resource Capabilities.*

The resources managed by a single Pilot are generally limited to compute resources only and more specifically to single compute nodes.

### *Resource Interaction.*

In its native mode as a batch queuing system, HTCondor is a middleware, and therefore the discussion of resource interaction does not apply. With the advent of Condor-G, Condor can be deployed to any site that offers a GRAM interface.

### *Overlay Management.*

GlideinWMS, a workload management system (WMS) [53] based on Condor GlideIns, introduces advanced Pilot capabilities to HTCondor by providing automated Pilot (`startd`) provisioning based on the state of an HTCondor pool.

### *Workload Semantics.*

The upper most layer of the Condor stack is the `problem solver`. A `problem solver` is a application layer structure built on top of the Condor agent. Two problem solvers are supplied in the distribution: master-worker and the directed acyclic graph manager (DAGMan). Other problem solvers can be built using the public interfaces of the agent. The task dependencies that are defined e.g. with DAGMAN are managed outside of the Agent and the lower layers are not aware of the workload semantics.

### *Task Binding Characteristics.*

None of the details of the `Job` are made known outside of the `Agent` (`Shadow`) until the actual moment of execution. This allows for the `agent` to defer placement decisions until the last possible moment. If the `Agent` submits requests for resources to several matchmakers, it may award the highest priority job to the first resource that becomes available,

without breaking any previous commitments.

### *Task Execution Modes.*

A `universe` in HTCondor defines an execution environment. There are distinct implementations of these universes available: the `Standard Universe` provides checkpointing (relevant because of the opportunistic mode of Condor) and remote system calls, which requires special compilation of the application though; the `Java Universe` hides the details of setting up the remote JVM; the `Vanilla Universe` is for applications that can't be modified; the `Grid Universe` is the mode for Condor-G; the `VM Universe` for exploiting VMware and XEN virtual machines; and the `Parallel Universe` for executing MPI applications. It should be noted that these `Universes` don't mix and that there is no (real) support for MPI jobs in the Condor-G mode.

### *Coordination and Communication.*

The various kernel elements can be deployed in multiple configurations and can all be run on a single machine but also fully distributed. The HTCondor system has their history in institutional environments and assume not too many network restrictions. The communication between the kernel components consist of both UDP and TCP based proprietary protocols that uses an array of dynamically allocated ports. With Condor-G the requirements were significantly reduced and the main communication is with the GRAM service.

### *Interface.*

In a pool, user tasks are represented through job description files and submitted to a (user-)`agent` via command-line tools. The main HTCondor distribution provided command line utils to setup these Glideins. These are now replaced by 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 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 it explicitly allows for *ad hoc* user-space deployment. BoSCO provides a Pilot 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 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).

CorralWMS is an alternative frontend for GlideinWMS-based infrastructures. It replaces or complements the regular GlideinWMS frontend 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.

### *Multitenancy.*

The main differences between the GlideinWMS and the CorralWMS front-ends lie with identity management and resource sharing. While GlideinWMS pilots (glidins) are provisioned on a per-VO base and shared/re-used among 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).

### *Robustness.*

As stated before, one of the initial drivers for Condor was the heterogenuous environments and the need to provide a robust runtime environment for the application by means of the sandbox. Additionally, for applications that support it, Condor provides checkpointing which allows applications to be restarted in case of failure or migrated to another resource.

### *Security.*

HTCondor inter-component communication is based on the home grown CEDAR message-based communication library. It allows client and servers to negotiate an appropriate security protocol. Regarding the execution of tasks, there is a level of trust assumed between resource and application.

### *Files and Data.*

The handling of files and data was initially not a primary design goal and did not extend well beyond staging relatively small data into the sandbox. The `Standard Universe` offers the ability to re-route file I/O back to the `Shadow` of the `Agent`. Over the years, HTCondor has been integrated with various data tools. Out of the same stable as HTCondor comes NeST[]. NeST is a software-only storage appliance that supports a myriad of storage protocols and integrates with the `Matchmaker` to announce its availability. With storage systems like NeST being available, one can then use a system like Stork[] to manage the transfers to and from this storage resources. Stork can make educated decisions about data placement by coordinating with the `Matchmaker`. Finally, Parrot[] is a mechanism that offers a POSIX I/O interface to applications for files that are otherwise only available remotely.

*Development Model.*
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 for the past 20 years.

### 4.2.5 MyCluster

MyCluster [61] was developed to allow users to submit and manage jobs across heterogeneous NSF TeraGrid resources in a uniform, on-demand manner. TeraGrid, the predecessor of XSEDE, existed as a group of compute clusters connected by high-bandwidth links facilitating the movement of data, but with many different middlewares requiring cluster-specific submission scripts. MyCluster allowed all cluster resources to be aggregated into one personal cluster with a unified interface, being either SGE, OpenPBS or Condor. This enhancement to user control was envisioned as a means of allowing users to submit and manage thousands of jobs at once and across heterogeneous distributed computing infrastructures, while providing a familiar and homogeneous interface for submission.

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 capabilities which users can then use to schedule their applications.

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 implicit. Rather than directly *binding* tasks 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.

*Task Execution Modes.*
MyCluster's *Task Execution Modes* are explicitly limited to "serial" tasks although it is unspecified whether this also confines it to single core **tasks** only.

*Pilot Resource Capabilities.*
As per the earlier description, MyCluster was specifically targeted to TeraGrid resources and thereby the *Pilot Resource Capabilities* are those of the TeraGrid, being HPC resources. As the **Task Execution Modes** are limited to serial tasks, the only relevant resource that is exposed is the "CPU" or "core".

*Resource Interaction.*
The *Resource Interaction* is exclusively by the `Agent Manager` through Globus GRAM to start a `Proxy Manager` at each site.

*Workload Semantics.*
All **tasks** are considered independent and no other *Workload Semantic* is described.

*Task Binding Characteristics.*
As the Pilots are not exposed (and the **tasks** are considered homogeneous) there are no explicitly controllable **Task Binding Characteristics**. This in effect makes it a **Late Binding** mechanism.

*Overlay Management.*
With respect to **Overlay Management** MyCluster allows the user to configure the number of `Proxies` per site, the number of CPUs per `Proxy` and the list of sites to submit to. All resources that are acquired through the `Proxy Managers` are pooled together. These resource specifications are just requests, it depends on the behavior of the queue whether (and when) these resources are actually acquired. In addition to these static resource specifications, MyCluster includes a mode of operation where `Proxies` can be `Migrated` to other sites. The rationale for `migration` is that the weight of the resource requests can be moved to the site with the shortest queuing time.

*Architecture.*
When the user starts a session via the `vo-login` command a `Agent Manager` is instantiated. The `Agent Manager` in turn starts a `Proxy Manager` at every resource site gateway host through Globus GRAM and a `Master Node Manager` at the local client machine. The `Proxy Manager` has an accompanying `Submission Agent` that also runs on the resource gateway. The `Submission Agent` interacts with the local queuing systems and submits the `Job Proxy` that launches a `Task Manager` process on a compute host. The `Task Manager` starts one or more `Slave Node Manager` processes on all the allocates compute hosts. Ultimately, the `Slave Node Manager` in control of the worker node. Depending on the configuration, the `Slave Node Manager` starts the Condor, SGE, or OpenPBS job-starter daemons, which in turn connect back to their master daemon started earlier on the local client machine by the `Master Node Manager`.

*Coordination and Communication.*
The communication between `Proxy Manager` and `Agent Manager` is over TCP.

*Security.*
The TCP communication channels are not encrypted but do use GSI-based authentication and has measures in place to prevent replay attacks.

*Robustness.*
The `Agent Manager` maintains little state. It is assumed that "lost" `Proxy Managers` will re-connect back when they recover. The `Proxy Manager` is restartable and will try to re-establish the `Proxies` based on a last-known state stored on disk when they got lost due to exceeding wallclock limitations or node reboots;

*Interoperability.*
The primary goal of MyCluster was *Interoperability* over multiple TeraGrid resources.

*Interface.*
The *Interface* of MyCluster depends on the choice of the underlying system as that one is exposed through the `Vir-`

`tual Login Session`. The interface also allows for users to interactively monitor the status of their **Pilots** and **tasks**.

*Multitenancy.*
The created cluster 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. The clusters can be created per user, per experiment or to contribute to the resources of a local cluster.

*Files and Data.*
MyCluster doest not offer and file staging capabilities other than those of the underlying systems provide. This means that it exposes the file capabilities of Condor, but doesn't offer any file staging capabilities when using SGE.

*Development Model.*
MyCluster is no longer developed or supported after the TeraGrid project came to a conclusion in 2011.

*Conclusion.*
MyCluster illustrates how an approach aimed at **multi-level scheduling** by marshaling multiple heterogeneous resources lends itself to a Pilot-based approach. The fact that the researchers behind it developed a complete Pilot system while working toward interoperability/uniform access is a testament to the usefulness of Pilots in addressing these problems. The end result is a complete Pilot system, despite the authors of the system being constructed not having used the word "pilot" once in their main publication.

While not an official product, a recent and similar approach has been adopted at NERSC. The operators of the Hopper cluster provide a tool called MySGE which allows for the user to provision a personal SGE cluster on Hopper.

### 4.2.6 PanDA

PanDA (Production and Distributed Analysis) [66] was developed to provide a multi-user workload management system (WMS) for ATLAS [**?**]. ATLAS is a particle detector at the Large Hadron Collider at CERN that requires a WMS to handle large numbers of jobs for their data-driven processing workloads. In addition to the logistics of handling large-scale job execution, ATLAS needed also integrated monitoring for analysis of system state and a high degree of automation to reduce the need for user/administrative intervention.

PanDA has been initially deployed as an HTC-oriented, multi-user WMS system for ATLAS, consisting of 100 heterogeneous computing sites [41]. Recent improvements to PanDA have extended the range of deployment scenarios to HPC and Cloud infrastructures making PanDA a general-use Pilot system [44].

*Architecture.*
PanDA's central component is called the `PanDA server`. It keeps track of jobs, matches jobs with pilots and resources, dispatches jobs to resources, and coordinates data management. The PanDA server is implemented as a stateless multi-process REST web service, with a database backend, to which it communicates through the `Bamboo` interface. A central job queue allows users to submit jobs to distributed resources in a uniform manner. PanDA's **Pilot** is called, appropriately enough, a `Pilot`, and handles the execution environment. When pilots are launched, they collects information about their worker nodes that is sent back to the job dispatcher. The main functionality of the Pilot is the so-called multi-job loop. It connects to the PanDA server to ask for jobs to run. If a matching job does not exist, the pilot ends. If a job does exist, the pilot forks a separate thread (`runJob`) for the job and starts monitoring its execution from the `monitor`.

*Pilot Deployment.*
An independent component, AutoPyFactory[**?**], manages the delivery of Pilots to worker nodes via a number of schedulers (`pilot factories`) serving the sites at which PanDA operates. This component handles Pilot submission, management, and monitoring. AutoPyFactory supersedes the first generation component called `PandaJobScheduler`, as well the second generation one called `AutoPilot`).

*Task Binding.*
Workload jobs are assigned to activated and validated pilots by the PanDA server based on brokerage criteria like data locality and resource characteristics.

*Workload Semantics.*
On the Pilot level there are no rich workload semantics. Workload semantics exists only in the `ProdSys` component that feeds jobs into the database from which the PanDA server retrieves its tasks.

*Task Execution Modes.*
Given the nature of the application workload and the targetted infrastructure, PanDA is targetted (exclusively?) towards sequential program execution.

*Pilot Resources.*
PanDA began as a specialized Pilot for the LCG Grid, and has been extended into a generalized Pilot which is capable of working across other grids as well as HPC[] and cloud[] resources.

*Resource Interaction.*
PanDA traditionally relies on Condor-G for its interaction with DCIs. For recent endeavors in HPC PanDA uses SAGA to interface with the queuing systems.

*Coordination and Communication.*
Communication between the client and the server is based on HTTP/S RESTful communication. Coordination of the workflow is maximally asynchronous.

*Interface.*
Jobs are submitted to PanDA via a simple Python client. The client needs to define job sets, their associated datasets as well as the input and output files at submission. The client API has been used to implement the PanDA front ends for ATLAS production, distributed analysis and US regional production.

*Multitenancy.*
PanDA is a *true* multi-user system, as both the WMS

manages workloads for multiple users (the whole ATLAS community), but also the Pilots are capable of executing tasks on the resources for multiple users by means of glExec[]. The latter enables the re-use of Pilots amonst users which has a positive effect on latency from a user perspective.

*Files and Data.*
`PanDA Dynamic Data Placement` [41] 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.

*Robustness.*
The wrapper of the Pilot takes care of disk space monitoring, and enables job recovery of failed jobs and restarting from crash whenever possible.

For production jobs the majority of the errors are site or system related, while for user analysis jobs the most common issues are related to application software. Pilot mechanisms like job recovery contribute to the robustness against site related failures.

A comprehensive monitoring system supporting production and analysis operations, including site and data management information for problem diagnostics, usage and quota accounting, and health and performance monitoring of PanDA subsystems and the computing facilities being utilized.

*Interoperability.*
PanDA enables the concurrent execution of workload on heterogeneous resources.

*Security.*
Job specifications from the client are sent to the PanDA server via a secure HTTPS connection authenticated with a GSI certificate proxy. gLExec is used to use the job user's identity instead of Pilot credentials for executing tasks.

*Performance and Scalability.*
PanDA has been used to process approximately a million jobs a day [17] which handle simulation, analysis, and other work [41]. The ATLAS experiment itself produces several petabytes of data a year which must be processed and analyzed.

The PanDA system is serving over 100k production jobs and 35k user analysis jobs concurrently.

*Development Model.*
PanDA is a multi-stakeholder project that has been active for many years. As such, it went through many iterations. We try to refer to the current state as much as possible. Current funding for PanDA enables the project to reach out to new user communities.

### 4.2.7 RADICAL-Pilot

The authors of this paper (the RADICAL group) have been engaged in theoretical and practical aspects of Pilot systems for the past several years. In addition to formulating the P* Model[] which by most accounts is the first complete conceptual model of Pilots, the RADICAL group is responsible for the development and maintenance of RADICAL-Pilot[]. RADICAL-Pilot is the groups long-term effort for creating a production level Pilot system. The effort is build upon the experience gained from developing, maintaining and using BigJob[], an initial prototype of Pilot system.

*Pilot Resource Capabilities.*
RADICAL-Pilot is mainly tailored towards HPC environments such as the resources of XSEDE and NERSC[]. The primary **resource** that is exposed is the compute node, or more specifically the cores within such a node.

*Resource Interaction.*
RADICAL-Pilot uses the Simple API for Grid Applications (SAGA) [27, 29] in order to interface with different DCIs. Thanks to SAGA, RADICAL-Pilot submits the Pilot Agent as a **job** through the reservation or queuing system of the DCI. Once the `Agent` is started, there is no direct *resource interaction* with the DCI except for monitoring of the state of the `Agent` process. Adopting SAGA has enabled RADICAL-Pilot to expand to many of the changing and evolving architectures and middleware.

*Overlay Management.*
The programming interface of RADICAL-Pilot enables (and requires) the explicit configuration of the **overlay**. The user is expected to specify the size, type and destination of the pilot(s) he wants to add to the **overlay**. Through the concept of `UnitManagers` on the client side, the user can group Pilots together and foster them under a single scheduler or have multiple independent `UnitManagers` and Pilots, all with their own `scheduler`. On HPC style systems there is typically one Pilot that manages all the resources that are allocated to that specific run, but there can be as many Pilots per resource as the policies allows concurrently running jobs.

*Workload Semantics.*
The **workload** within RADICAL-Pilot consists of `ComputeUnits` that represent **tasks**. From the perspective of RADICAL-Pilot there are no dependencies between these `ComputeUnits` and once a `ComputeUnit` is given to the control of RADICAL-Pilot it is assumed to be ready to be executed. RADICAL-Pilot includes the concept of `kernel abstractions`. These are generic application descriptions that can be configured on a per-source basis. Once an `application kernel` is configured for a specific resource and available in the repository, the user only needs to refer to the `application kernel`. RADICAL-Pilot then takes care of setting up the right environment and executing the right executable. This facility is especially useful in the case of **late-binding**, when it is unknown on which Pilot (and thereby resource) a task will run at the time of its submission.

*Task Binding Characteristics.*
**Task** to **resource binding** can be done either implicitly or explicitly. A user can bind a task explicitly to a pilot, and thereby to the resource the pilot is scheduled to, making this a form of early binding. The user can also submit a task to a unit manager that schedules units to multiple pilots. In this case it is the semantics of the scheduler that decides on the task binding. RADICAL-Pilot supports mul-

tiple schedulers that can be selected at runtime. Two schedulers are currently implemented: (i) a round-robin scheduler that binds incoming tasks to associated pilots in a rotating fashion, irrespective of their state, and thereby performing **early binding**. (ii) a BackFilling scheduler that binds tasks to pilots once the pilot is active and has available resources, thereby making it a **late binding** scheduler.

*Task Execution Modes.*

RADICAL-Pilot supports two type of **tasks**. One is the generic single node task, that can be single core, or multi-core threaded/OpenMP applications. In addition, RADICAL-Pilot has extensive support for MPI applications, where it supports a large variety of launch methods that are required to successfully run MPI applications on a wide range of HPC systems. The `launch methods` are a modular system and new `launch methods` can be added when required.

*Architecture.*

From a high level perspective RADICAL-Pilot consist of two components. A client side python module that is programmable through an API and is used by scripts/applications, and the agent (or multiple agents) that runs on a resource and executes tasks.

One of the primary features of RADICAL-Pilot is that it completely lives in user-space and thereby requires no collaboration from the resource owner / administrator. Other than an online database that maintains state during the lifetime of a session, there are no other (persistent) service components that RADICAL-Pilot relies on.

*Coordination and Communication.*

MongoDB is the bridge between the client side and agent. MongoDB is a document-oriented database that needs to run in a location that is accessible for both type of components. The client side publishes the **workload** in the MongoDB which is picked up and then executed by the agent. The agent in turn publishes the status and the output of the `ComputeUnit` back to the MongoDB which can be retrieved by the client side. The main advantage of this model is that it works in situations where there is no direct communication channel between the client host and the compute resource, which is a very common scenario. The MongoDB database also offers (some) persistency which allows the client side to re-connect to active sessions stored in the database. The main drawback of the use of a database for communication is that all communication is by definition indirect with potential performance bottlenecks as a consequence.

*Interface.*

The client side RADICAL-Pilot is a Python library that is programmable through the so called `Pilot-API`. The application-level programmability that RADICAL-Pilot offers was incorporated as a means of giving to the end-user control over their job management. Users define their pilots and their tasks and submit them through the `Unit Manager`. They can query the state of pilots and units or rely on a `callback` mechanism to get notified of any state change that they are interested in. Users define their pilots and tasks in a declarative way, but the flow of activity is more of an imperative style.

*Interoperability.*

RADICAL-Pilot allows for *interoperability* with multiples types of resources (heterogeneous schedulers and clouds/-grids/clusters) at one time. It does not have *interoperability* across different Pilot systems.

*Multitenancy.*

RADICAL-Pilot 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.

*Robustness.*

For fault-tolerance RADICAL-Pilot relies on the user to write the application logic to achieve *Robustness*, to assist in that process, it does report task failures.

*Security.*

The compute resource *Security* aspects of RADICAL-Pilot are that it follows security measures of the respective scheduler systems (i.e. policies like allocations, etc).

*Files and Data.*

RADICAL-Pilot supports many modes of `data staging` although all of them are exclusively file based. Files can be staged at both Pilot and `ComputeUnit` level, and they can be transferred to and from the staging area of the pilot and of the compute unit `sandbox` from and to the client machine. RADICAL-Pilot also allows the transfer of files from third party locations onto the compute resource. In addition, the ComputeUnits can be instructed to use files from the pilot sandbox. In this way, the pilot sandbox is turned into a shared file storage for all its compute units. The user has the option to Move, Copy or Link data based on the performance and usage criteria.

*Performance and Scalability.*

RADICAL-Pilot is one of the largest pilot based consumers of resources on XSEDE[]. Many thousands concurrent ComputeUnits can be managed by each Pilot Agent and scalability can be achieved in many dimensions as multiple pilots can easily be aggregated over multiple distinct DCIs.

*Development Model.*

RADICAL-Pilot is an Open Source project released under the MIT license. Its development is public and takes place on a public Github repository. The project is under active funding and development and has a number of external projects that use it as a foundation layer for job execution.

## 4.3 Overall observations

Some of systems we have discussed, like DIRAC, X, are inherently service oriented, or exposed as a service. On the other side of the spectrum, systems like DIANE and RADICAL-Pilot take the form of a library or a framework. From a formal standpoint, services can be abstracted by a library and a library can be abstracted by a service, so the main distinction between these architectures if the default form they come in and how they are meant to be deployed.

# 5. DISCUSSION AND CONCLUSION

Section 3 offered a description of the minimal capabilities and properties of a Pilot system alongside a vocabulary defining 'Pilot' and its cognate concepts. Section 4 offered a classification of the core and auxiliary properties of Pilot system implementations, and the analysis of an exemplar set of Pilot system implementations. Considered altogether, these contributions outline the characteristics and properties of a paradigm for the execution of tasks on distributed resources by means of resource placeholders. This is here referred to as 'Pilot paradigm'.

In this section, the properties of the Pilot paradigm are critically assessed. The goal is to show the generality of this paradigm and how Pilot systems go beyond the implementation of a special purpose trickery to speed up the execution of a certain type of workload. Once understood the breadth of the Pilot paradigm, it is contextualized by describing its relation with relevant domains such as middleware, applications, security, and the enterprise. Finally, the section closes with a look into the future of Pilot systems moving from the current state of the art and discussing the sociotechnical and engineering challenges that are being faced both by developers and target users.

## 5.1 The Pilot Paradigm

The Pilot paradigm identifies a type of software system with both general and unique characteristics. This paradigm is general because, in principle, it does not depend on a single type of workload, a specific infrastructure, or a unique performance metric. In principle, systems implementing the Pilot paradigm can execute workloads composed by an arbitrary number of tasks with disparate requirements. For example, as seen in §4, Pilot systems can execute homogeneous or heterogeneous bags of independent or intercommunicating tasks with arbitrary duration, data, or computation requirements.

The same generality applies to both the type of resource and of infrastructure on which a Pilot system can execute given workloads. As seen in §3, the Pilot paradigm demands resource placeholders but does not specify the type of resource that the placeholder should expose. Nonetheless, most of the Pilot systems illustrated in §4 are designed to expose only compute resources, i.e. the cores of the compute nodes for which the pilot is a placeholder. Tasks executed on these Pilots may also use data or network resources but mostly as accessories to their execution on computational resources.

It is important to note that the focus on computational resources is not mandated by the Pilot paradigm. In principle, pilots can be placeholders also and exclusively for data or network resources, depending on the capabilities exposed by the middleware of the target infrastructure. For example, in Ref. [pilot data], the notion of Pilot-Data has been conceived using the power of symmetry, i.e., Pilot-Data taken to be a notion as fundamental to dynamic data placement and scheduling as Pilot is to computational tasks.

Currently, no Pilot system exposes networking resources by means of placeholders but there is no theoretical limitation to the implementation of what may be called 'Pilot networks'. With the advent of Software-Defined Networking and User-Schedulable Network paths in mind, the concept was already hinted at in [52].

Pilot systems already operate on multiple platforms. Originally devised for HTC grid infrastructures, Pilot systems have been (re-)engineered to operate also on HPC and Cloud infrastructures. Important to note that many of the Pilot systems described in §4 have been adapted to work with different infrastructures without undergoing major re-architecturing. This is further indication of the independence of the Pilot paradigm from infrastructural details, especially those concerning the type of container used to instantiate a placeholder.

Usually, Pilots are thought as a means to optimize the throughput of single-core (or at least single-node), short-lived, uncoupled tasks execution [47, 54, 31]. The analysis presented in §4 showed how restrictive such a view is. The Pilot paradigm has been successfully integrated within workflow systems to support execution of workloads with articulated data and task dependencies. Furthermore, both single and multi-core tasks can be executed by many existing Pilot systems, with some of them also offering resource-specific MPI support. As such, depending on the type of the given workload, the Pilot paradigm can be implemented to optimize different metrics of task execution other than throughput.

Characterized and defined primarily by resource placeholding, multi-level scheduling, and late binding, Pilot systems represent a category of software in itself. Pilot systems are not just an alternative implementation of a scheduler, of a workload manager, of a batch system, or of any other special-purpose software component. This specific system is used when abstracting one or more type of physical resources with well-defined spatial and time properties can lead to the optimization of one or more parameters of the workload execution.

Appreciating the generality of the Pilot paradigm and the independence of its implementations from other types of software, helps also to avoid a potential misunderstanding involving Pilots and how multi-level scheduling works. Pilot systems are not circumventing the infrastructure middleware but simply abstracting away some of its properties in order to optimize one or more user-defined (performance) metrics.

Multi-level scheduling is not replacing the infrastructure-level scheduler either. As clearly illustrated in §3, resource containers are still created on the target infrastructure by means of that infrastructure's capabilities and the second level of scheduling is completely contained within the boundaries of the resource container. Optimizing the usage of the resource containers, for example by minimizing idling time, is an implementation issue, not an intrinsic limitation of the Pilot paradigm.

It should be noted that shifting the control over tasks scheduling away from the infrastructure middleware does not imply that the end-user will become necessarily responsible for an efficient resource utilization. A Pilot system can be implemented as a middleware component, as seen with HTCondor and its role as an integral part of the infrastructure middleware stack on some DCIs.

The generality of the Pilot paradigm may come as a surprise when considering the most common requirement that motivates its implementations. Leveraging multi-level scheduling so to increase the execution throughput of large workloads made of short running tasks has been achieved without having to devise a new paradigm. For example, as seen in §4 PanDA, Falkon, or DIRAC were initially developed as single-point solutions, focusing on either a type of

workload, a specific infrastructure, or the optimization of a single performance metric.

Appreciating the properties of the Pilot paradigm becomes necessary once requirements of infrastructure interoperability, support for multiple types of workloads, or flexibility in the optimization of execution are introduced. Satisfying those requirements requires abstracting the specificity of middleware, infrastructure architectures, and application patterns. As shown in this paper, this process of abstraction means to develop an understanding of the Pilot paradigm.

## 5.2 Pilot Paradigm in Context

The adoption of the Pilot paradigm in a scientific computing environment brings to the fore a rich set of details about its specification, requirements, and functionalities.

The analysis offered in §4 shows a widespread adoption of the Pilot paradigm with multiple implementations serving diverse use cases. While the paradigm remains the same, its application varies depending on the characteristics of the type of computation considered. These differences are well illustrated by looking at how Pilot systems support the execution of parallel and distributed applications.

Both types of applications may have similar Pilot and task durations but their spatial dimension varies sensibly. Typically, parallel applications require tightly coupled computations [ref] that, in turn, need Pilots to be instantiated on infrastructures offering potentially large-core, large-memory compute nodes. These pilots are usually submitted by means of the infrastructure headnode, each pilot having at least as many core as those offered by a node of the target infrastructure.

Differently from parallel applications, distributed applications usually require loosely coupled or fully independent computations. Consequently, Pilots supporting their execution depend less from the size of the nodes of the target infrastructure and more on the overall number of possibly independent cores that they can hold. These Pilots can be instantiated also by means of decentralized services (e.g. WMS) aggregating resources from possibly heterogeneous infrastructures.

The notion of task as defined in §3 and then utilized in §4 shapes both the conceptualization and implementation of the Pilot paradigm. Conceptually, adding a wrapper to the executables of a distributed or parallel application means creating a well-defined space for their localization. This may involve not only the specification of program switches, data locations, or library dependences but also loading modules, downloading supporting code, or on-the-fly compilations and execution of supporting applications. Implementation-wise, the process of localization requires tight coupling with the target infrastructure, its capabilities, policies, and specific configurations.

The implementation of the localization process becomes more challenging in the presence of heterogeneous infrastructures. The same executable may require different localization procedures depending on where it will be executed. More importantly, different types of localizations may or may not be available or allowed on all the target infrastructures. This imposes decision capabilities on the task scheduler and the development of a robust interoperability layer between the Pilot system and the heterogeneous target infrastructures.

Interoperable localization offers the opportunity to execute heterogeneous executables via Pilot systems. As seen in §2, the abstraction of Pilot was progressively developed in a distributed context leaving under-explored the execution of parallel applications by means of Pilot systems. This is merely a historical byproduct, not an intrinsic limitation of the Pilot paradigm. For example, by leveraging localization, different flavors of MPI libraries can be loaded and specific execution commands may be used to run MPI applications via a task wrapper and a Pilot agent.

The benefits of executing tightly-coupled parallel code via Pilot systems are especially evident when considering workloads involving homogeneous tasks and possibly diverse computational phases. For example, applications with separated simulation and analysis phases [ref], or applications integrating multiple types of analysis [ref], ....

The definition of the term 'Pilot' as a resource placeholder in §3 and the review of its implementations in §4 helps also to explore the relation between Pilot systems and middleware. The notion of Pilot as a resource placeholder is more general than that of a job as defined in the context of HPC and grid middleware. This is exemplified when considering cloud-based DCIs.

Within a IaaS [cit], the instantiation of virtual machines (VMs) introduces overheads due to the bootstrapping of an operating system, including data and networking subsystems. Nonetheless, queuing of VMs that need to be instantiated has a more predictable and consistent behavior compared to that of HPC queues [42, 26].

Pilots can benefit from the more predictable VM instantiation time because VMs can be used as resource placeholders. A pilot agent can be executed within a VM, retrieving and executing tasks on the resources of (one or more) VM. Conceptually, the role and functionalities of the Pilot agent are the same when used in a IaaS, HPC, or grid infrastructures. The implementation details are isolated within the agent code itself and have no bearing on the overall design of the Pilot system.

This analysis highlights not only the generality of the notion of resource placeholder but also the independence of the Pilot paradigm from the specificities of the target infrastructures and their middleware. The Pilot paradigm abstracts the notion of resources, alongside that of scheduling and task execution, creating a well-defined and isolated logical space for the management of the application execution.

Importantly, an analogous degree of isolation is offered by the Pilot paradigm to the application layer. The scientific application landscape is diverse and fragmented, especially when considering distributed applications [cit]. The current state of the workflow (WF) systems (a type of distributed application) is a paradigmatic example of this situation.

Many WF systems have been implemented with significant duplication of effort and limited means for extensibility and interoperability. One important contributing factor to these limitations is the lack of suitable, open, and possibly standard-based interfaces for the resource layer. Most WF engines are developed with proprietary solutions to access the resource layer, solutions that cannot be shared with other engines and that often serve specific requirements, use cases, and infrastructures.

The Pilot paradigm and an open Pilot API like the one specified in [ref Pilot-API] offer the missing layer on which WF and, more generally, distributed applications could be built upon. As with the resource layer, the Pilot paradigm

offers a well-defined and well-isolated layer between applications and resources. This fosters extensibility, interoperability, and modularity by separating the application description and logic from the management of its execution, and from the provisioning and aggregation of resources. In turn, this avoids the need to develop special-purpose, vertical, and end-to-end applications, the main source of duplication and fragmentation in the current distributed application landscape.

The multi-level scheduling and the heterogeneity of the infrastructures used by Pilot systems present several challenges for the authentication, authorization, and accounting processes (AAA). As seen in §4, AAA requirements for Pilot systems are limited to: (i) the multitenancy of the instantiated Pilots; and (ii) to the credentials used to submit Pilots to the remote infrastructure and to use them to execute the application tasks.

The authentication and authorization used by the user to access her own Pilots is a well-isolated implementation detail while the credential used for Pilot deployment depends on the target infrastructure. Pilot systems that do not implement multitenancy for each pilot instance have little need for pilot-level AAA while those implementing it may have to use advanced abstractions like those of virtual organizations [23] and federated certificate authorities [30].

The AAA requirements of the target infrastructures are a diverse and often inconsistent array of mechanisms and policies. This affect the deployment of Pilots and, possibly, the binding of tasks to pilots. This is especially the case when HPC, grid, and cloud systems are targeted by the same Pilot system implementation. The Pilot paradigm is gregarious in the face of such a diversity as no AAA requirements are imposed on the resource layer. Pilot systems need to present the credentials required by the resource layer consistently with the infrastructure policies. These credentials are supposed to be provided by the application layer therefore reducing to a minimum the requirements for AAA implementation within Pilot systems.

Moving away from the relationship between the Pilot paradigm and critical implementation details associated with its use in a scientific, production-grade computing environment, the Pilot paradigm has relevant implications also for the enterprise sector, which has seen an increased adoption of distributed computing.

Over the past years, Hadoop [2] emerged as distributed computing for data-intensive tasks in the enterprise sector. While early versions of Hadoop were monolithic, tightly coupling the Map Reduce programming framework to the underlying infrastructure resource management, Hadoop 2 introduced the YARN [59] resource manager to support not only Map Reduce but also heterogeneous workloads.

YARN supports multi-level scheduling enabling the application to deploy their own application-level scheduling routines on top of Hadoop-managed storage and compute resources. With YARN managing the lower resources, the higher-level runtimes typically use an application-level scheduler to optimize resource usage for the application. Applications need to initialize their so-called ApplicationMaster via YARN; the Application Master is then responsible for allocating resources in form of so called containers for the applications. YARN then can execute tasks in these containers.

As with the HPC environment, the support for different application workloads and job types as long-lived vs. short-lived applications, or homogeneous vs. heterogeneous tasks is challenging. Pilot-like frameworks are emerging for YARN to address such a challenge. Llama [37] offers a long-running application master for YARN designed for the Impala SQL engine. TEZ [5] is a DAG processing engine primarily designed to support the Hive SQL engine allowing the application to hold containers across multiple phases of the DAG execution without the need to de/reallocate resources. REEF [14] is a similar runtime environment that provides applications a higher-level abstractions to YARN resources allowing it to retain memory and cores supporting heterogeneous workloads.

## 5.3 Future Directions and Challenges

The Pilot landscape is currently fragmented with a high degree of duplicated effort and capabilities. The reasons for such a balkanization are to be traced back mainly to two factors: (i) the relatively recent discovery of the importance of the Pilot paradigm; and (ii) the development model fostered within academic institutions.

As seen in §2 and §4, Pilot systems emerged as a pragmatic solution for improving the throughput of distributed applications. Pilot systems were not thought from their inception as an independent class of middleware but, at best, as a module within a specific framework. This not only promoted duplication of development effort across frameworks and projects but also hindered the appreciation for the generality of the Pilot abstraction, the theoretical framework underlying the Pilot systems, and the application execution paradigm they enable.

Pilot systems inherited the development model of the scientific projects within which they were initially developed. Most of the early Pilot systems were developed to serve a specific use case, often within the remit of a specific research project. As a consequence, these systems were not engineered to promote (re)usability, modularity, well-defined interfaces, or long-term sustainability. Furthermore, for the most part code was not build around a wider community, following recognized and widely-accepted coding standards.

All this is likely to change with the progressive appreciation of the generality, richness, and versatility of the Pilot paradigm. The ongoing re-engineering of PANDA and DIRAC and the development of RADICAL-Pilot indicate that the Pilot paradigm is becoming the leading approach for large scale scientific applications. For example, PANDA supports the execution of 5 million jobs weekly for the ATLAS project [3] and in the past year RADICAL-pilot has be used to execute [...] million jobs on XSEDE.

It should be noted that 'large scale scientific applications' includes both distributed and parallel applications executed on diverse types of infrastructure. This shows how historical distinctions between, for example, HPC and HTC are progressively loosing their meaning when considering resource provisioning and application execution by means of Pilot systems.

As argued in the previous section, Pilot systems are agnostic towards the type of application that is executed. They need to be engineered so to support different types of applications but the Pilot paradigm and its underlying notions of resource placeholder and multi-level scheduling do not require for the application to be distributed, parallel, MPI, closed-coupled, or communication independent.

Pilots holds resources and the properties of such resources can be arbitrarily specified. Analogously, Pilots allow for tasks to be scheduled but the type of matching between the requirements of these tasks and the capabilities offered by the resources held by the pilot is not mandated by the Pilot paradigm itself. For this reason, adopting the Pilot paradigm for the execution of increasingly diverse applications should be seen as a set of often challenging implementation details more than a foundational issue requiring new middleware paradigms.

Scalability and performance are two important elements that should see further development in the Pilot system landscape. The ongoing increase of the scale required by scientific applications [cit] poses new challenges to the development of effective Pilot systems. Increase in scale brings the need to focus on federation of resources, integration of leadership machines, and scaling-out to cloud infrastructures. Furthermore, large scale executions make fault-tolerance and advanced mechanisms for fault-recovery essential. For example, Pilot systems need to support the rerun of those few tasks that failed among the millions that have been successfully executed. Analogously, the failure of a Pilot among the many instantiated must not be an unrecoverable event. In this context, resource discovery and dynamic handling of application execution will probably play an increasingly important role.

With scale, performance becomes an increasingly relevant issue. Performance not only as in overall application throughput but, more subtly and yet not less relevant, also as in overheads imposed by the Pilot system on the application execution. Such overheads need to be quantifiable and accountable for. Especially in a context of heterogeneous types of applications, Pilot overheads may be relevant to decide what type of resource to target, how many pilots to instantiate, or whether a Pilot system should be used at all.

The analysis offered in this paper indicates that the number of Pilot systems actively developed should be reduced so to avoid duplication while promoting consolidation, robustness, and overall capabilities. Nonetheless, this conclusion should not be brought to an extreme. A single Pilot system should not be elected as the only implementation worthy of development effort or adoption. As with other middleware [10] the problem is not to eliminate special purpose systems in favor of a single encompassing solution but it is, instead, having both of them, depending on the application and use case requirements. A multi-purpose, functionally encompassing Pilot system is desirable for all those use cases in which applications are heterogeneous in the type of computation, time, or space. For specific applications designed to run on a single type of infrastructure, a special purpose system might still be more effective. Another rationale against rigid consolidation is the diversity in programming languages used, the different deployment models required, and what that means for the interaction with existing applications.

## 5.4 Contributions

*Contributions.* This paper offers several contributions to support the understanding, design, and adoption of Pilot systems. §2 provided an overview of both the motivations that led to the development of the Pilot abstractions and its early implementations, and an analysis of the many Pilot systems that have been and still are used to support scientific computing. These systems were clustered on the base of their capabilities to show the progressive process of implementations of the Pilot abstraction.

The analysis provided in §2 also showed the heterogeneity of the Pilot landscape and the need for a clarification of the basic components and functionalities that distinguish a Pilot system. These were described in §3 offering a way to identify Pilot systems and discriminate them from other type of middleware. §3 contributed also a well-defined vocabulary that can be used to reason consistently about different implementation of the Pilot abstraction.

Both contributions offered in §3 were then leveraged in §4 to analyze a set of paradigmatic Pilot system implementations. The shift from understanding the minimal set of components and functionalities characterizing the Pilot abstraction to the comparison of actual Pilot implementations required to outline core and auxiliary implementation properties. Table 2 summarize these contributions by mapping both types of property into the Pilot system components and functionalities. This table can be used to analyze any middleware software, decide whether it is a Pilot system, and assess the richness of its functionalities.

The work done in §2, §3, and §4.1 supported the comparative analysis of Pilot system implementations offered in §4.2. This contribution outlined differences and similarities among implementations, showing how they impact on the overall Pilot systems capabilities and their target use cases. Thanks to these insights, it was possible in §5 to highlight the properties of the Pilot paradigm.

Generality along the types of workload, resource, and performance indicates the fundamental role that the Pilot paradigm can play to support distributed computing to higher scales and on diverse infrastructures. Among all the contributions of this paper, this indication is one of the most important. When understood, it would show to the user communities, DCIs managers, and developers that converging towards a robust, interoperable, openly available, community developed and maintained Pilot system is a privileged way to progress the computational-intense research of many scientific fields.

## Acknowledgements

This work is funded by the Department of Energy Award (ASCR) DE-FG02-12ER26115 and NSF CAREER ACI-1253644. 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".

## 6. REFERENCES

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

[2] Apache Hadoop. `http://hadoop.apache.org/`, 2014.

[3] G. Aad et al. The atlas experiment at the cern large hadron collider. *JINST*, 3:S08003, 2008.

[4] 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.

[5] Apache TEZ. `http://hortonworks.com/hadoop/tez/`, 2014.

[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] 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.

[8] 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.

[9] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, (3):81–84, 2014.

[10] P. A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[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] B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. *Proc. VLDB Endow.*, 6(12):1370–1373, Aug. 2013.

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

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

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

[18] A. B. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings 11th International Parallel Processing Symposium*, pages 209–218, 1997.

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

[20] 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.

[21] J. M. *et al.* Ganga: A tool for computational-task management and easy access to grid resources. *Computer Physics Communications*, 180(11):2303 – 2316, 2009.

[22] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[23] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[24] 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.

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

[26] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[27] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). OGF Recommendation Document, GFD.90, Open Grid Forum, 2007. `http://ogf.org/documents/GFD.90.pdf`.

[28] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. Saga: A simple api for grid applications. high-level application programming on the grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[29] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. K. leijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid applications, High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[30] J. Horwitz and B. Lynn. Toward hierarchical identity-based encryption. In *Advances in Cryptology—EUROCRYPT 2002*, pages 466–481. Springer, 2002.

[31] G. Juve, E. Deelman, K. Vahi, and G. Mehta. Experiences with resource provisioning for scientific workflows using corral. *Scientific Programming*, 18(2):77–92, 2010.

[32] J. H. Katz. Simulation of a multiprocessor computer system. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 127–139. ACM, 1966.

[33] 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.

[34] 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.

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

[36] H. Li, D. Groep, J. Templon, and L. Wolters. Predicting job start times on clusters. In *IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2004*, pages 301–308, 2004.

[37] Llama. `http://cloudera.github.io/llama/`, 2014.

[38] C.-D. Lu, J. Browne, R. L. DeLeon, J. Hammond, W. Barth, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra. Comprehensive job level resource usage measurement and analysis for xsede hpc systems. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 50:1–50:8, New York, NY, USA, 2013. ACM.

[39] A. Luckow, L. Lacinski, and S. Jha. Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 135–144. IEEE, 2010.

[40] A. Luckow, M. Santcroos, O. Weider, A. Merzky, S. Maddineni, and S. Jha. Towards a common model for pilot-jobs. In *Proceedings of The International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[41] 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.

[42] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

[43] 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.

[44] 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.

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

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

[47] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, et al. The open science grid. In *Journal of Physics: Conference Series*, volume 78, page 012057. IOP Publishing, 2007.

[48] 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.

[49] 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.

[50] 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.

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

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

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

[54] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Würthwein. The pilot way to grid resources using glideinwms. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 2, pages 428–432. IEEE, 2009.

[55] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[56] The LHCb collaboration. LHCb computing: Technical Design Report. Technical report, June 2005.

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

[58] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[59] V. K. Vavilapalli. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC*, 2013.

[60] G. Von Laszewski, M. Hategan, and D. Kodeboyina. Java cog kit workflow. In *Workflows for e-Science*, pages 340–356. Springer, 2007.

[61] 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.

[62] D. Weitzel, D. Fraser, B. Bockelman, and D. Swanson. Campus grids: Bringing additional computational resources to hep researchers. *Journal of Physics: Conference Series*, 396(3):032116, 2012.