A Comprehensive Perspective on the 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 analysis of the evolution, functional properties and implementations of the pilot abstraction, the most common incarnation of which are the many Pilot-Job software systems that have 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

*Corresponding author

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 the proliferation of Pilot-Job systems used in production cyberinfrastructure, the Pilot-Job abstraction is arguably one of the few widely-used in distributed computing.

A variety of Pilot-Job systems have emerged: HTCondor-G/ Glide-in [1], Swift [2], DIANE [3], DIRAC [4], PanDA [5], GWPilot [6], Nimrod/G [7], Falkon [8] and MyCluster [9, 10] to name a few. These systems are for the most part functionally equivalent and motivated by similar objectives; nonetheless, their implementations often serve specific use cases, target specific resources, and lack in interoperability.

The fundamental reason for the proliferation of Pilot-Job systems is that they provide a simple solution to the rigid and static resource management historically found in high-performance and distributed computing. There are two ways in which Pilot-Jobs break free of the rigid resource utilization model: (i) through a process often referred to as late-binding [11, 12, 13], Pilot-Jobs make the selection of heterogeneous and dynamic resources easier and effective; and (ii) Pilot-Jobs decouple the workload specification from the task management. The former results in the ability to utilize resources "dynamically", the latter simplifies the placement of workloads on those resources.

Pilot-Jobs have been almost exclusively developed within pre-existing systems and middleware dedicated to specific scientific requirements. 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 systems or of the user cases justifying their development.

This approach is not problematic in itself and has led to effective implementations that serve many million jobs a year on diverse computing platforms [14, 15]. However, the lack of an conceptual clarity and an explicit appreciation of the computing paradigm underlying Pilot-Jobs has undermined the development of dedicated designs and implementations. This limitation is illustrated not only by the duplication of effort, but also by an overall immaturity of the available systems in terms of functionalities, flexibility, portability, interoperability, and, most often, robustness.

This survey is motivated by the fact that, in spite of the demonstrated potential and proliferation of Pilot-Job systems, there remains significant lack of clarity and understanding about the Pilot-Job abstraction. We believe this results in unfulfilled and lost potential as well as significant overhead and repetition of effort. Ultimately, these also contribute to a high-cost of development and low software sustainability.

This paper offers a critical analysis of the current state of the art providing the conceptual tooling required to appreciate the properties of the Pilot paradigm, i.e. the abstraction and the methodology underlying Pilot-Jobs systems. 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-Job systems and systems with pilot-like characteristics.

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

In §4, the focus shifts from analyzing the design of a Pilot-Job 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-Job 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

At least five features need elucidation to understand the technical origins and motivations of the Pilot abstraction: task-level distribution and parallelism, master-worker pattern, multi-tenancy, multi-level scheduling, and resource placeholding. Even if these features are not unique to the Pilot abstraction, the Pilot abstraction brings them together towards an integrated and collective capability. This section offers an overview of these five features and an analysis of the relationship they hold with the Pilot abstraction. A chronological perspective is taken so to contextualize the inception and evolution of the Pilot abstraction into its multiple and diverse implementations.

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 (WLCG) Data Challenge[1] [16, 17, 18, 19], and then introduced in writing as 'pilot-agent' in a 2005 LHCb report [20, 21]. 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 [22] (Figure 1) to solve systems of differential equations for weather forecasting [23]. This factory required 64,000 'human computers' supervised by a senior clerk. The clerk

[1]Based on private communication.

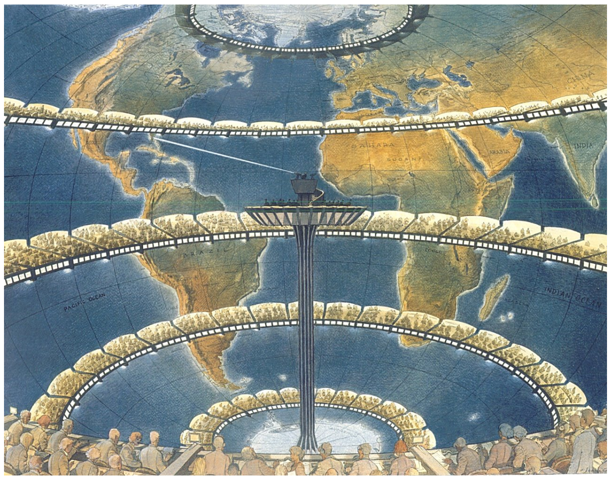

Figure 1: *Forecast Factory* as envisioned by Lewis Fry Richardson. Drawing by François Schuiten.

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 today called "high-performance" task-level parallelism, but also of the coordination pattern for distributed and parallel computation called "master-worker".

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, high-performance machines brought at least three key differences when compared to the carbon-based Forecast Factory devised by Richardson. Most of modern high-performance machines are meant to be used by multiple users, i.e. they support multi-tenancy. Furthermore, diverse high-performance machines were made available to the scientific community, each with both distinctive and homogeneous properties in terms of architecture, capacity, capabilities, and interfaces. Furthermore, high-performance machines 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" [24] and first used in the time of punch cards [25, 26], adopt the 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 policies (e.g. fair usage).

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

The resource provisioning of high-performance machines is limited, irregular, and largely unpredictable [27, 28, 29, 30]. 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 [31, 32]. 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 scheduling is one of the strategies devised to improve resource access across multiple high-performance machines. The idea is to hide the scheduling point of each high-performance machine behind a single scheduler. The users or the applications submit their tasks to the a scheduler that negotiates and orchestrates the distribution of the tasks via the scheduler of each available high-performance machine. While this approach promises an increase in both scale and usability of applications, it also introduces complexities across resources, middleware, and applications.

Several approaches have been devised to manage the complexities associated with multi-level scheduling. For example, the approaches developed under the umbrellas of grid computing [8, 33, 34] or cloud computing [35, 36, 37, 38], target the resource layer; others the application layer as, for example, with workflow systems [39, 40, 36, 41]. 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 high-performance machines 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 resides with the application layer [42, 43, 44]. 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 high-performance machines, and lack of portability and interoperability.

Consider for example a simple distributed application implementing the master-worker pattern. With a single high-performance machine, the application requires the capability of concurrently submitting tasks to the queue of the scheduler of the high-performance machine, and retrieve and aggregate their outputs. When multiple high-performance machines are available, the application requires directly managing submissions to several queues or the capability to leverage a third-party 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 solution to 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 high-performance machine. 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 high-performance machines and then using a dedicated 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 high-performance machine, 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 high-performance machine scheduler.

2.2 Evolution of Pilot System Implementations

The Pilot abstraction has a rich set of properties [45] that have been progressively implemented into multiple Pilot-Job systems. Figure 2 shows the introduction of Pilot-Job systems over time while Figure 3 shows their clustering along the axes of workload management and pilot functionalities. Starting from a set of core functionalities focused on acquiring remote resources and utilizing them independently from the resource management of the remote high-performance machine, Pilot-Job systems progressively evolved to include advanced capabilities like workload and data management.

AppLeS [46] 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 [47].

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 a dedicated abstraction layer between those of the applica-

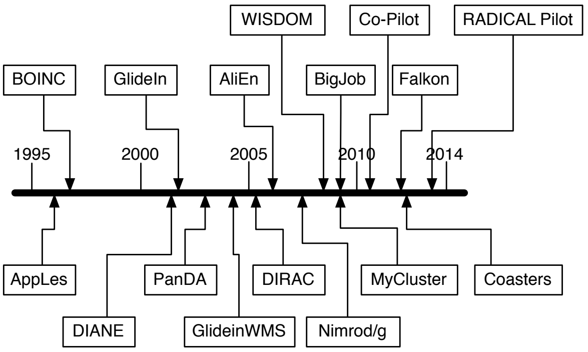

Figure 2: Introduction of systems over time. When available, the date of first mention in a publication or otherwise the release date of software implementation is used.

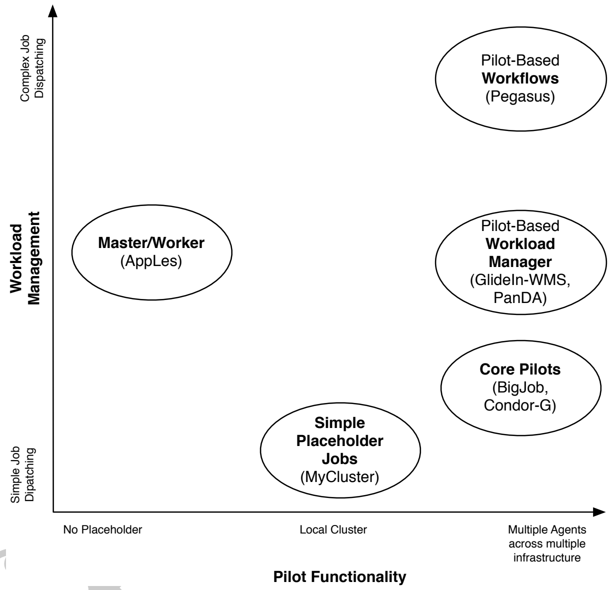

Figure 3: A partial clustering of pilots along functionality.

tion 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[48], shortly followed by distributed.net [49] 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 [50].

It should be noted that process of resource acquisition is different in AppLes and voluntary computing. The former has prior knowledge of the available resources while the latter has none. As a consequence, AppLes can request and orchestrate a set of resources, 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.

HTCondor (formerly known as Condor) is a high-throughput distributed computing framework that uses diverse and possibly geographically distributed resources [51]. Originally, HTCondor was created for systems within one administrative domain but Flocking [52] made it possible to group multiple machines into aggregated resource pools. However, resource management required system level software configurations that had to be made by the administrator of each individual machine of each resource pool.

This limitation was overcame by integrating a resource placeholder mechanism within the HTCondor system. HTCondor-G/GlideIn [1] allowed users to add grid resources to resource pools. In this way, users could uniformly execute jobs on heterogeneous resource pools. Thanks to its use of resource placeholders, GlideIn was one of the systems pioneering the implementation of the Pilot abstraction, enabling some Pilot capabilities also for third parties systems like Bosco [53].

The success of HTCondor-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 remote machines. This means that GlideIn cannot be deployed without involving the machine owners and system administrators. Implemented as a service, GlideIn supports integration with distributed application frameworks but does not programmatically support the development of distributed applications by means of dedicated APIs and libraries.

Concomitant and correlated with developments at LHC there was a "Cambrian Explosion" of Pilot-Jobs in first decade of the millennium, e. g. DIANE [3], GlideInWMS, DIRAC [4], PanDA [54], AliEn [55], and Co-Pilot [56]. Each of these Pilot-Job systems serves a particular user community and experiment at the LHC: DIRAC [4] was developed by the LHCb experiment [21]; AliEn [55] was developed by the ALICE experiment; PanDA (Production and Distributed Analysis) [54] was developed by the ATLAS experiment [57]. Due to socio-technical reasons, the CMS experiment at LHC mostly converged around the HTCondor-Glidein-GlideinWMS [58] ecosystem. Another Pilot-Job system that is used in the LHC context is Co-Pilot [56]. Interestingly, these systems are functionally very similar, work on almost the same underlying infrastructure, and serve applications with very similar (if not identical) characteristics.

It is worth noting that Co-Pilot was set up as an integration point among grid-based Pilot-Job systems (such as AliEn and PanDA), clouds, and volunteer computing resources.

The BigJob Pilot-Job system [59] was designed to support task-level parallelism on distributed HPC resources, to broaden the type of applications supported by the pilot-based execution model, and ultimately to extend the Pilot abstraction beyond the boundaries of compute tasks. BigJob offers 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 [60, 61, 59] to work on a variety of infrastructures.

BigJob was recently re-implemented as a production-level tool named 'RADICAL-Pilot' [62]. BigJob/RADICAL-Pilot represent an evolution of the Pilot abstraction: initially pilots were implemented as ad hoc place holder machinery for a specific application but evolved to be integrated with the middleware of remote resources. BigJob/RADICAL-Pilot implements the Pilot abstraction as an interoperable compute and data management system that can be programmatically integrated into end-user applications and thus provides both features.

Another ongoing evolutionary trend has been to implement the Pilot abstractions 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.

Pilot-Job systems have proven an effective tool for managing the workloads executed in the various stages of a scientific workflow. For example, the Corral system [63] has been developed to serve as a frontend to HTCondor GlideIn and to optimize glides (i.e. pilots) placement for the Pegasus workflow system [64]. 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 [2] is a scripting language designed for expressing abstract workflows and computations. The language also provides capabilities for executing external application as well as the implicit management of data flows between application tasks. Swift uses a Pilot implementation called "Coaster" [65], 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 [8]. Falkon was engineered for executing many small tasks on High Performance Computing (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 workloads.

The proliferation of Pilot-Job systems and their integration within other type of application and middleware 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-Job 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 clustering.

The evolution of the Pilot paradigm has been organic and uncoordinated leading to an inconsistent terminology related to the Pilot abstraction, its implementations and usage. A coherent understanding of the Pilot components and functionalities is still missing leading to a blurred definition of Pilot abstraction and how it should be distinguished from other abstractions, and application or middleware 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 primary motivation of the next section.

3. UNDERSTANDING THE LANDSCAPE: DEVELOPING A VOCABULARY

The overview presented in §2 shows a degree of heterogeneity both in the functionalities and the vocabulary adopted by different Pilot-Job systems. Implementation details sometimes hide the functional commonalities and differences among Pilot-Job systems. 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 a description of the logical components and functionalities shared by every Pilot-Job system. The goal is to offer a paradigmatic description of a Pilot-Job 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 Pilot-Job systems introduced in §2 are engineered to allow for the execution of multiple types of workloads on machines with diverse middleware, e.g. grid, cloud, or HPC. This is achieved in different ways, depending on use cases, design and implementation choices, but also on the constraints imposed by the middleware and policies of the targeted machines. The common denominators among Pilot-Job systems are defined along three dimensions: purpose, logical components, and functionalities.

The purpose shared by every Pilot-Job system is to improve workload execution when compared to executing the same workload directly on one or more machines. Performance of workload execution is usually measured by throughput and execution time to completion, but other metrics could also be considered: energy efficiency, data transfer minimization, scale of the workload executed, or a mix of them. Metrics that are not performance related include reliability, ease of application deployment, and generality of workload. In order to achieve the required metrics under given constraints, each Pilot-Job system exhibits characteristics that are both common or specific to one or more implementations. Discerning these characteristics requires isolating the minimal set of logical components that characterize every Pilot-Job system.

At some level, all Pilot-Job systems employ 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 machines. The Workload Manager handles the scheduling of one or more workloads on the available resource placeholders. 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 vary significantly across Pilot-Job systems (see §4). For example, two or more logical components may be implemented by a single software module or additional functionalities may be integrated into the three managers. Nevertheless, the Pilot, Workload, and Task Managers can always be distinguished across different Pilot-Job systems.

Each Pilot-Job system supports a minimal set of functionalities that allow for the execution of workloads: **Pilot Provisioning**, **Task Dispatching**, and **Task Execution**. Pilot-Job systems need to schedule resource placeholders on the target machines, 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-Job 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 necessary characteristics of a Pilot-Job system.

Among the core functionalities that characterize every Pilot-Job 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 target machines. Resource placeholders are scheduled onto target machines by means of dedicated capabilities, but once scheduled and then executed, these placeholders make their resources directly available for the execution of the tasks of a workload.

The provisioning of resource placeholders depends on the capabilities exposed by the middleware of the targeted machine and on the implementation of each Pilot system. Typically, for middleware adopting queues, batch systems, and schedulers, provisioning a placeholder involves it being submitted as a job. For such middleware, a job is a type of logical container that includes configuration and execution parameters alongside information on the application to be executed on the machine's compute resources. Conversely, for machines without a job-based middleware, a resource placeholder would be executed by means of other types of logical container as, for example, a virtual machine or a Docker Engine [66, 67].

Once resource placeholders are bound to the resources of a machine, tasks need to be dispatched to those placeholders for execution. Task dispatching does not depend on the functionalities of the targeted machine's middleware so it can be implemented as part of the Pilot-Job system. In this way, the control over the execution of a workload is shifted from the machine's middleware 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 machine's scheduler. For example, the execution of tasks of a workload will not individually depend upon the specifics of the targeted machine's state or availability, but rather on those of the placeholder. More elaborate execution patterns involving task and data interdependence can thus be implemented independent of the target machine's middleware capabilities. Ultimately, this is how Pilot-Job 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 systems, and Pilot-Job systems are no exception. The three logical components – Workload Manager, Pilot Manager, and the Task Manager – need to communicate in order to coordinate the execution of the given workload on the instantiated resource placeholders. Nonetheless, Pilot-Job systems are not defined by any specific communication and coordination pattern . The logical components of a Pilot-Job system may communicate using any suitable pattern (e.g. one-to-one, many-to-one, one-to-many **??** with a push or pull model), or coordinate adopting any suitable mechanism (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 used to achieve effective communication and coordination.

As seen in §2, master-worker is a very common coordination pattern among Pilot-Job 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 within 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 capabilities to coordinate with its neighbor executors leveraging, for example, a dedicated overlay network. While these systems would adopt different coordination strategies, they could be both be considered Pilot-Job systems.

Data management can play an important role within a Pilot-Job 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 machine's middleware. How these requirements are implemented do not define a core functionality of the Pilot system.

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-Job 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-Job system.

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

3.2 Terms and Definitions

The terms 'pilot' and 'job' are arguably among the most relevant when referring to Pilot-Job systems. 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 machines and middleware used by Pilot-Job systems. These machines 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 and dependences of those operations, and indications on how they should be executed and satisfied. Implementations of a task may include wrappers, scripts, or applications.

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

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.

Distributed Computing Resource (DCR). A machine characterized by a tuple: {a set of possibly heterogeneous resources, a middleware software, and an administrative domain}.

Workloads are characterized by multiple tasks. The tasks could be homogeneous, heterogeneous or one-of-a-kind. There does not exist an established taxonomy for workload description; we propose a workload taxonomy based upon the orthogonal properties of coupling, dependency and similarity of the tasks. Workloads comprised of tasks that are independent and effectively indistinguishable from other tasks; such a workload is commonly referred to as a Bag-of-Tasks (BoT). Ensembles (ENS) represent workloads when the collective outcome of the tasks is relevant (e.g., computing the average property). The tasks that comprise the ensembles (i.e., ensemble-members) in turn can have varying degrees and types of coupling; coupled ensembles might have global (synchronous) or local exchanges (asynchronous), regular or irregular communication. We categorize workloads as coupled ensembles (C-ENS) independent of the specific details of the coupling between the tasks. A workflow (WF) represents a workload with arbitrarily complex relationships among the tasks, ranging from dependencies (e.g., sequential or data) to coupling between the tasks (e.g., frequency or volume of exchange).

A cluster is a typical example of DCR: it offers sets of compute and data resources; it deploys a middleware as, for example, the Torque batch system, the Globus grid middleware, of the OpenStack platform; and enforces policies of an administrative domain like, for example, XSEDE, OSG, CERN, NERSC, or a certain University.

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

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

When considering Pilot-Job systems, jobs and tasks are functionally analogous but qualitatively different. Functionally, both jobs and tasks are containers – i.e. metadata wrappers around one or more executables often called '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 DCR middleware where such a container is submitted. Accordingly, tasks are considered as the functional units of a workload, while jobs as a way to schedule one or more tasks on a DCR with a specific middleware. It should be noted that, given their functional equivalence, the two terms can be adopted interchangeably when considered outside the context of Pilot-Job systems. Indeed, workloads are encoded into jobs when they have to be directly executed on DCRs that support or require that type of container.

As described in §3.1, a resource placeholder needs to be submitted to a DCR wrapped in the type of container supported by the middleware of that specific DCR. For this reason, the capabilities exposed by the job submission system of the target DCR determine the submission process of resource placeholders and its specifics. For example, when wrapped within a 'job', placeholders are provisioned by submitting a job to the DCR queuing system, and become available only once the job has been scheduled, and only for the duration of the job lifetime.

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

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-Job systems. Depending upon context, in addition to the term 'placeholder', a pilot may also be named 'agent' or 'Pilot-Job' [11, 68]. 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-Job system implementations.

Until now, the term Pilot-Job system has been used to indicate those systems capable of executing workloads on pilots. From now on, the term 'Pilot system' will be used instead, as the term 'job' in 'pilot-job' identifies just the way in which a pilot is provisioned on a DCR exposing specific middleware The use of the term 'Pilot-Job system' should therefore be regarded as a historical artifact, viz., the targeting of a specific class of DCRs in which the term 'job' was, and still is, meaningful. With the development of DCR middleware based on new abstractions as, for example, that of virtual machine, the term 'job' has become too restrictive, a situation that can lead to terminological and conceptual confusion.

We have now defined resources, DCRs, 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 and diverse DCRs.

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

As seen in §2.1, three more terms associated with Pilot systems 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 [6, 69, 41]. This definition of 'multi-level scheduling' is problematic because the term 'level' is left undefined. It is not clear what constitutes a level or how its boundaries should be assessed. 'Multi-entity' and 'multi-stage' are better terms to describe the scheduling properties of Pilot systems, as these terms specifically indicate that (at least) two entities are scheduled and that such a scheduling happens at separate moments in time.

In the Pilot systems, a portion of the resources of a DCR is allocated to one or more pilots, and the tasks of a workload are dispatched for execution to those pilots. As alluded to in the previous subsection, this is a fundamental feature of Pilot systems. It leads to the following: (i) more flexible and potentially reduced times to completion of workloads as a consequence of avoiding a centralized job management system multiple times; 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, tasks can be further scheduled by deciding about their placement within the pilot's resource allocation.

The greater control obtained as a consequence of removing the dependence of every task on the job submission system of the DCR is one of the main reasons for the success and early adoption of Pilot systems. As mentioned in §3.1, the tasks of a workload can be executed on a pilot without each task individually waiting in the queuing system of the DCR's middleware. This greater control results in several advantages, including possibly increasing the throughput of the workload execution, and reusing active pilots to execute multiple workloads. How tasks are actually scheduled to pilots is a matter of implementation. For example, a dedicated scheduler could be adopted, or tasks might be directly scheduled 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 DCR, is active thereafter, until it completes or fails. Early binding indicates the binding of a task to an inactive pilot; late binding the binding of a task to an active one. Early binding is 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 instead critical to assure 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, Pilot systems permit both types of binding, even within 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.

Figure 4 offers a diagrammatic overview of the logical components of Pilot systems (green) alongside their functionalities (blue) and the defined vocabulary (red). An application submits a workload composed of tasks to the Pilot system via an interface (a). The Pilot Manager is responsible for pilot provisioning (b), the Workload Manager to dispatch tasks to the Task Manager (c), the Task Manager to executes those tasks once the pilot has become available (d). Figure 4 shows not only the separation between the DCR and the Pilot system but also how the resources on which tasks are executed, are contained within different logical and physical components. Appreciating the characteristics and functionalities of a Pilot system depends upon understanding the levels at which each of its component exposes capabilities.

Note how in Figure 4, scheduling happens at the DCR level, for example by means of a cluster scheduler, and then at the pilot level. This illustrates what has been called here a multi-entity and multi-stage scheduling, a couple of terms replacing the more common but less precise 'multi-level scheduling'. Figure 4 depicts the separation between scheduling at pilot level and scheduling at workload manager level, highlighting the four entities involved: jobs on DCR middleware, and tasks on pilots.

Figure 4 also helps 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 provisioned, it is the pilot and not the container that is responsible of both holding a set of resources and offering the functionalities of the task manager.

Figure 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 adopted among such components. This is why no distinction is made diagrammatically between, for example, early and late binding.

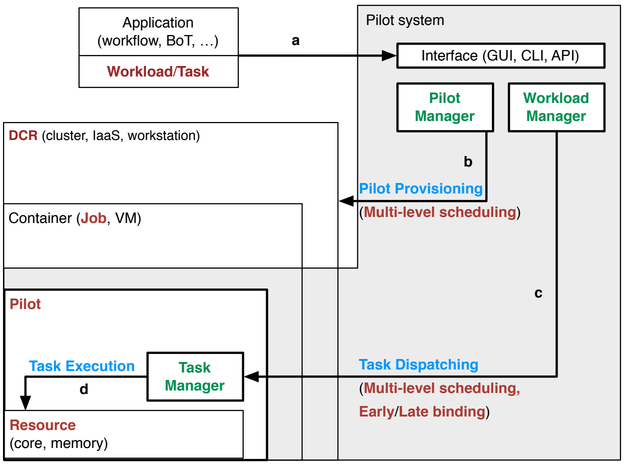

Figure 4: Diagrammatic representation of the logical components, functionalities, and core vocabulary of a Pilot system. The terms of the core vocabulary are highlighted in red, those of the logical components of a Pilot system in green, and those of their functionalities in blue.

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

4. PILOT SYSTEMS: ANALYSIS AND IMPLEMENTATIONS

Section §3 offered two main contributions: (i) the minimal sets of logical components and functionalities of Pilot systems; and (ii) 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 analyze a selection of Pilot systems.

The goal of this section is twofold. Initially, the Pilot functionalities presented in §3.1 are used as the basis to infer core and auxiliary Pilot implementation properties. Subsequently, a selection of Pilot systems are described and then analyzed. In this way, insight is offered about the commonalities underlying these systems alongside their distinguishing differences.

4.1 Core and Auxiliary Properties

This subsection analyzes the properties of diverse implementations of a Pilot system. Two sets of properties are introduced: core and auxiliary (see Tables 1 and 2). Both sets of properties are chosen by considering the implementation requirements of the Pilot capabilities as defined in §2. Core properties are necessary for every Pilot system implementation to provide the minimal set of components and functionalities as described in §3.1. Auxiliary properties are instead required only as a support to the implementation of core properties. For example, while every Pilot system has to implement a mechanism for pilot instantiation, a specific authentication and authorization protocol may be required to support the instantiation of pilots for a particular middleware of a DCR. As such, auxiliary properties are not necessarily shared among all Pilot systems.

Every Pilot systems has five core properties, two pertaining to the components and functionalities related to pilots, three to those of workloads: Pilot Resources, Pilot Deployment, Workload Semantics, Workload Binding, and Workload Execution.

The core properties 'Pilot Resources' and 'Pilot Deployment' characterize the implementation of a Pilot Manager and its Pilot Provisioning functionality. Pilot Resources identifies the type of resources that the Pilot system exposes, e.g. compute, data, or networking while Pilot Deployment describes the modalities of scheduling, utilization, and aggregation of pilots. For example, to schedule a pilot onto a specific DCR, Pilot systems need to acquire and process information about what type of container to use (e.g. job, virtual machine), what type of scheduler the resource exposes, the amount of cores and duration they can be requisitioned for, what filesystems can be accessed, or how to communicate with the DCR.

The core properties 'Workload Semantics' and 'Workload Binding' characterize the implementation of a Workload Manager and its Task Dispatching functionality. Semantically, a workload description contains all the information necessary for it to be dispatched to an appropriate DCR and to be executed. 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 property 'Workload Execution' characterizes the implementation of a Task Manager and its Task Execution functionality. 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 DCR and its policies. As such, the execution of the workload may depend on how the Pilot systems implement their interactions with the remote DCR.

4.1.1 *Core properties*

Following is an in depth description of core properties by which any Pilot system can be characterized (see Table 1). This list of properties is minimal and complete. Note that these are the properties of Pilot systems and not of individual pilot instances executed on DCRs.

- **Pilot Resources**. Usually, pilots expose compute resources but 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

Property	Description	Component	Functionality
Pilot Resources	Types and capabilities of the resources held by the pilot	Pilot Manager	Pilot Provisioning
Pilot Deployment	Modalities and protocols for pilot scheduling, utilization, and aggregation	Pilot Manager	Pilot Provisioning
Workload Semantics	The specification of the semantics among tasks captured in the workload description	Workload Manager	Task Dispatching
Workload Binding	Modalities and policies for binding tasks to pilots	Workload Manager	Task Dispatching
Workload Execution	Types of tasks and the mechanisms to execute tasks	Task Manager	Task Execution

Table 1: Mapping of the Core Properties of Pilot system implementations onto the components and functionalities described in §3.1. Core properties are necessary for every Pilot system to implement these components so to provide those functionalities.

depending on the architecture of the DCR in which it is instantiated. For example, a pilot may hold multiple compute nodes, single nodes, or portion of the cores of each node. The same applies to file systems and their partitions, or to software defined or physical networks.

- **Pilot Deployment**. Pilots are scheduled and then bootstrapped on a DCR. The characteristic of both operations varies depending on the design choices of the Pilot systems. For example, pilot scheduling may be fully automated or directly controlled by applications and end-users, and pilots can be bootstrapped from code downloaded at every instantiation or from code that is bundled by the DCR. Both scheduling and bootstrapping varies depending on whether the Pilot system targets one or more DCR with heterogeneous or homogeneous middleware. For example, a design based on connectors will be required when targeting DCRs with different middleware, optional when targeting a single DCR or multiple DCRs with the same middleware.

- **Workload Semantics**. The tasks of a workload are dispatched to pilots depending on the workload semantics. Dispatching decisions depend on the temporal and spatial relationships among tasks and the type of capabilities needed for their execution. Pilot systems support varying degrees of semantic richness of workloads and its tasks.

- **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 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.

4.1.2 Auxiliary properties

Several auxiliary properties play an important role in supporting the implementation and usability of the core properties (see Table 2). While these properties might be necessary for Pilot systems deployment and usability, in of themselves they do not distinguish a Pilot as a unique system. For example, programming and user interfaces; interoperability across differing middleware and other Pilot systems; multitenancy of pilots as opposed to that of DCRs; 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.

- **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 limited to the trade-offs implied by each choice, especially when considering how they affect the Core Properties.

- **Communication and Coordination**. Communication and coordination are features of every distributed

Property	Description
Architecture	Frameworks and architecture that the components and their whole are build with
Coordination and Communication	The interaction between the components of the system
Interface	Interface that the user can use to interact with the system
Interoperability	Interoperability between Pilots on multiple DCIs
Multitenancy	The use of components by multiple (simultaneous) users
Resource Overlay	The aggregation of resources from multiple pilots into overlays
Robustness	The measures in place to increase the robustness of the components and the whole
Security	AAA considerations for the components and the whole
Files and Data	The mechanisms that the system offers to explicitly deal with files and data
Performance and Scalability	A description of scale and limitations and measures to reach that
Development Model	The development and support model for the software
DCR Interaction	Modalities and protocols used to coordinate the pilot system/DCR interaction

Table 2: Summary of Auxiliary Properties and their descriptions. Auxiliary properties are required as a support to the implementation of core properties.

system, but 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 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 DCRs characterized by heterogeneous middleware. It allows for a Pilot system to provision pilots and execute workloads on different types of DCR middleware (e.g. HTC, HPC, Cloud but also HTCondor, 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 usage and applicability of Pilot systems' Core Functionalities are influenced by security protocols and mechanisms. The analysis here focuses on authentication and authorization, describing their main characteristics and the differences among Pilot systems implementations. An in depth analysis of the security protocols for Pilot systems is outside the scope of this paper.

- **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 targeted 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.

- **DCR Interaction**. Pilot systems interact with DCRs at multiple levels. The degree of coupling between the Pilot system and the DCR 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 DCR, and may have to mediate task binding and execution by means of remote interfaces and protocols.

Both core and auxiliary properties have a direct impact on the use cases for which Pilot systems are 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 for specific workloads varies across implementations. Some Pilot systems support Virtual Organizations (VO) [70] and running tasks from multiple users on a single pilot while others support jobs using a Message Passing Interface (MPI). Furthermore, 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).

4.2 Pilot System Implementations

A set of Pilot systems has been chosen for further analysis to show how the core properties just described are implemented under different requirements both in terms of target DCRs and application workloads. Examining these Pilot systems using the common vocabulary defined in §3.2 exposes similarities and differences allowing a detailed analysis. Critically assessing these differences will bring to the fore the generality of the pilot abstraction, its independence from specific software systems and technological environments, and the more relevant challenges of Pilot systems implementation. Table 3 offers a summary of the core properties for each analyzed Pilot system[2].

4.2.1 DIANE

DIANE [3] (DIstributed ANalysis Environment) has been developed at CERN to support the execution of workloads for the DCRs associated with EGI/WLCG. DIANE has since been used also in the Life Sciences [74, 75, 76] and in few other scientific domains [77, 78].

DIANE is an application task coordination framework for distributed applications that can be executed by means of the master-worker pattern [3]. DIANE consists of four logical components: a TaskScheduler, an ApplicationManager, a SubmitterScript, and a set of ApplicationWorkers [79]. The first two components – TaskScheduler and the ApplicationManager, are implemented as a RunMaster service, while the ApplicationWorkers as a WorkerAgent service. Submitter Scripts deploy ApplicationWorkers on DCRs.

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

The execution model of DAINE can be summarized in four steps [80]: 1. the user submits one or more jobs to DCR by means of SubmitScript(s) to bootstrap one or more WorkerAgent; 2. When ready, the WorkAgent(s) reports back to the ApplicationManager; 3. tasks are scheduled by the TaskScheduler on the available WorkerAgent(s); 4. after execution, WorkerAgents send the output of the computation back to the ApplicationManager.

[2]Pilot systems are ordered alphabetically in the table & text.

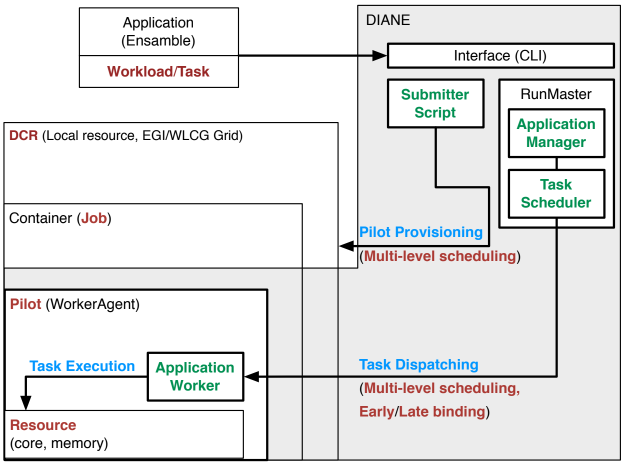

Figure 5: Diagrammatic representation of DIANE components, functionalities, and core vocabulary mapped on Figure 4.

The pilots used by DIANE hold compute resources on the target DCRs. ApplicationAgents are executed by the DCR middleware as jobs with mostly one core but possibly more. DIANE also offers a data service with a dedicated API and CLI that allows for staging files in and out of ApplicationAgents. This service represents an abstraction of the data resources and capabilities offered by the DCR, and it is designed to handle data only in the form of files stored into a file system. Network resources are assumed to be available and no abstractions are offered.

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

A computational task-management tool called GANGA [81, 82] is available to support the development of SubmitterScripts. GANGA offers a unified interface for job submission to DCRs with Globus, HTCondor, UNICORE, or gLite middleware. The main goal of GANGA is to facilitate the submission of WorkerAgents (i.e. pilots) to diverse DCRs by means of a uniform interface and abstraction.

DIANE has been designed to execute workloads that can be partitioned into ensembles of parametric tasks on multiple ApplicationAgents. Each task can consist of an executable invocation but also of a set of instructions, OpenMP threads, or MPI processes **??**. Relations among tasks and group of tasks can be specified before or during runtime enabling DIANE to execute articulated workflows. Plugins have been written to manage DAGs [83] and data-oriented workflows [84].

DIANE is primarily designed for HTC/Grid environments and to execute pilots with a single core. Nonetheless, the notion of "capacity" is exposed to the user to allow for the specification of pilots with multiple cores. Although the workload binding is controllable by the user-programmable TaskScheduler, the general architecture is consistent with a

Pilot System	Pilot Resources	Pilot Deployment	Workload Semantics	Workload Binding	Workload Execution
DIANE	Compute	Explicit	WF (MOTOUR [71])	Late	Serial
GlideinWMS	Compute	Implicit	WF (Pegaus [64], DAGMan [72])	Late	Serial, MPI
PanDA	Compute	Implicit	BoT	Late	Serial, MPI
RADICAL-Pilot	Compute, data	Explicit	ENS (EnsembleMD Toolkit [73])	Early, Late	Serial, MPI

Table 3: Overview of Pilot systems and a summary of the values of their core properties. Based on the tooling currently available for each Pilot system, the types of workload supported as defined in §3.2 are: BoT = Bag of Tasks; ENS = Ensembles; WF = workflows.

pull model. The pull model naturally implements the late-binding paradigm where every ApplicationAgent pulls a new task once it is available and has free resources.

4.2.2 *HTCondor Glidein and GlideinWMS*

The HTCondor Glidein system has been designed as part of the software ecosystem of HTCondor. The HTCondor Glidein system implements pilots within regular Condor pools. It was developed by the Center for High Throughput Computing at the University of Wisconsin-Madison (UW-Madison). HTCondor Glidein's original goal was to aggregate DCRs with heterogeneous middleware into HTCondor resource pools [85].

The logical components of HTCondor relevant to the Glidein system are: a set of Schedd and Startd daemons, a Collector, and a Negotiator [86]. Schedd is a queuing system that holds workload tasks and Startd handles the DCR resources. The Collector holds references to all the active Schedd/Startd daemons, and the Negotiator matches tasks queued in a Schedd to resources handled by a Startd.

HTCondor Glidein has been complemented by GlideinWMS [58], a Glidein-based workload management system that automates deployment and management of glideins on multiple types of DCR middleware. GlideinWMS builds upon the HTCondor Glidein system by adding the following logical components: a set of Glidein Factory daemons, a set of VO Frontend daemons, and a Collector dedicated to the WMS [87, 88]. Glidein Factories submits tasks to the DCRs middleware, each VO Frontend matches the tasks on one or more Schedd to the resource attributes advertised by a specific Glidein Factory, and the WMS Collector holds references to all the active Glidein Factories and VO Frontend daemons.

Figure 6 shows the mapping of the HTCondor Glidein Service and GlideinWMS elements to the components and functionalities of a pilot system as described in §3. The set of VO Frontends and Glidein Factories alongside the WMS collector implement a Pilot Manager and its pilot provisioning functionality. The set of Schedds, the Collector, and the Negotiator implement a Workload Manager and its task dispatching functionality. The Startd daemon implements a Task Manager alongside its task execution functionality. A glidein is a job submitted to a DCR middleware that, once instantiated, configures and executes a Startd daemon. As such, a glidein is a pilot.

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

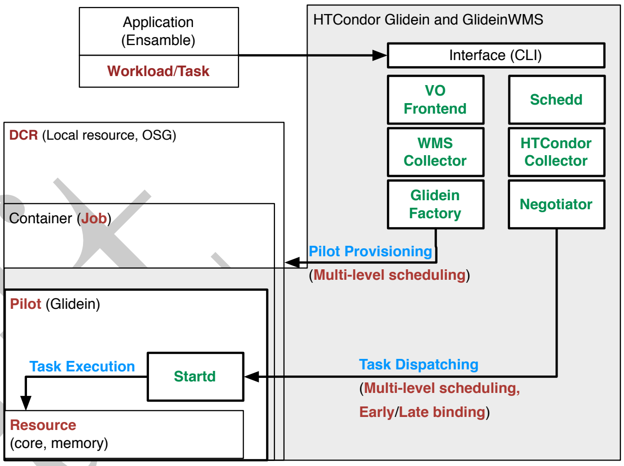

Figure 6: Diagrammatic representation of Glidein components, functionalities, and core vocabulary mapped on Figure 4.

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

The resources managed by a single glidein (i.e. pilot) are limited to compute resources. Glideins may bind one or more cores, depending on the target DCRs. For example, heterogeneous HTCondor pools with resources for desktops, workstations, small campus clusters, and some larger clusters will run mostly single core glideins. More specialized pools that hold, for example, only DCRs with HTC/-

grid/cloud middleware may instantiate glideins with a larger number of cores. Both HTCondor Glidein and GlideinWMS provide abstractions for file staging but no pilot abstraction is offered for data or network resources.

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

HTCondor Glidein and GlideWMS expose the interfaces of HTCondor to the application layer and no theoretical limitations are posed on the type and complexity of the workloads that can be executed [89]. For example, DAGman (Directed Acyclic Graph Manager) [72] has been designed to execute workflows by submitting tasks to Schedds, and a master-worker tool is available to design applications with a master-worker execution pattern.

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

Both HTCondor Glidein and GlideWMS rely on one or more HTCondor Collectors to match task requirements and resource properties, represented in form of ClassAds [90]. This matching can be evaluated right before the execution of the task. As such, both pilot systems allow for late binding. Early binding is not available.

4.2.3 PANDA

PanDA (Production and Distributed Analysis) [54] was developed to provide a multi-user workload management system (WMS) for ATLAS [57]. ATLAS is a particle detector at the Large Hadron Collider (LHC) at CERN that requires a WMS to handle large numbers of tasks for their data-driven processing workloads. In addition to the logistics of handling large-scale task execution, ATLAS also needs integrated monitoring for analysis of system state and a high degree of automation to reduce user and administrative intervention.

PanDA has been initially deployed as an HTC-oriented, multi-user WMS system for ATLAS, consisting of 100 heterogeneous computing sites [91]. Recent improvements to PanDA have extended the range of deployment scenarios to HPC and Cloud-based DCRs making PanDA a general-purpose Pilot system [92].

PanDA architecture consists of a Grid Scheduler and a PanDA Server [93, 94]. The Grid Scheduler is implemented by a component called 'AutoPilot' that submits jobs to diverse DCRs. The PanDA server is implemented by four main components: a Task Buffer, a Broker, a Job Dispatcher, and a Data Service. The Task Buffer collects all the submitted tasks into a global queue and the Broker prioritizes and binds those tasks to DCRs on the basis of multiple criteria. The Data Service stages the tasks' input file(s) to

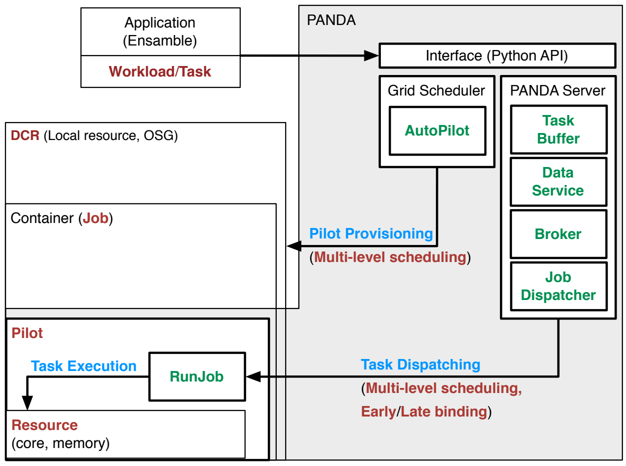

Figure 7: Diagrammatic representation of PANDA components, functionalities, and core vocabulary mapped on Figure 4.

the DCR to which the tasks have been bound using the data transfer technologies exposed by the DCR middleware (e.g. uberftp, gridftp, or lcg-cp). The Job Dispatcher delivers the tasks to the RunJob(s) running on the bound DCR.

Figure 7 shows how PANDA implements the components and functionalities of a pilot system as described in §3: the Grid Scheduler is a Workload Manager implementing Pilot Provision while the PanDA Server is a Task Manager implementing Task Dispatching. The jobs submitted by the Grid Scheduler are called 'Pilots' and act as pilots once instantiated on the DCR by contacting the Job Dispatcher component to request for tasks to execute.

The execution model of PANDA can be summarized in seven steps [95, 96]: 1. the user submits tasks to the PanDA server; 2. the tasks are queued within the Task Buffer; 3. the tasks requirements are evaluated by the Broker and bound to a DCR; 4. the tasks' input files are staged to the bound DCR by the Data Service; 5. the required pilot(s) are submitted as jobs to the target DCR; 6. the submitted pilot(s) becomes available and reports back to the Job Dispatcher; 7. tasks are dispatched to the available pilots for execution.

PanDA pilots expose computational resources. Pilots are designed to expose mainly a single core but extensions have been developed to instantiate pilots with multiple cores [97]. The Data Service of PanDA allows to integrate and automate data staging within the task execution process but no pilot-based data abstractions are offered [91]. Network resources are assumed to be available but no network-specific abstractions are made available.

The AutoPilot component of PanDA's Grid Scheduler has been designed to use multiple methods to submit pilots to DCRs. The PanDA installations of the US ATLAS infrastructure uses the HTCondor-G [1] system to submit pilots to the US production sites. Other schedulers enable AutoPilot to submit to local and remote batch systems or to the GlideinWMS frontend. Submissions via the canonical tools offered by HTCondor have also been used to submit tasks to cloud resources via PanDA.

PanDA was initially designed to serve specifically the AT-

LAS use case and, as such, to execute mostly single-core tasks with input and output files. Since its initial design, the ATLAS analysis and simulation tools have started to investigate multi-core task execution with AthenaMP [97] and PanDA has been evolving towards a general purpose workload manager [14]. As a consequence, PanDA is starting to offer experimental support for multi-core pilots and tasks with or without data dependences. PanDA also now supports applications from a variety of science domains.[**?**, **?**].

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

4.2.4 *RADICAL-Pilot*

The authors of this paper have been engaged in theoretical and practical aspects of Pilot systems for the past several years. In addition to formulating the P* Model [98] which by most accounts is the first complete conceptual model of a pilot system, the RADICAL group is responsible for the development and maintenance of RADICAL-Pilot[62, 99]. RADICAL-Pilot is the group's long-term effort for creating a production level Pilot system. The effort is built upon the experience gained from developing and deploying BigJob [59], and and integrating it with many applications on different DCRs [**?**, **?**, **?**].

RADICAL-Pilot consists of three main logical components: a Pilot Manager, a Compute Unit (CU) Manager, and a set of Agents. The Pilot Manager describes pilots and then submit them to DCR, while the CU manager describes tasks (i.e. CU) and schedules them to one or more pilots. Agents are instantiated on DCRs and execute the CUs pushed by the CU manager.

RADICAL-Pilot has been specifically designed to be a Pilot system and, as seen for example with DIANE, it closely resembles the description offered in §3 (see Figure 7). The Pilot Manager and the Workload Manager are implemented by the CU Manager. The Agent is deployed on the DCR to expose its resources and execute the tasks pushed by the CU Manager. As such, the Agent is a pilot.

RADICAL-pilot is implemented as a python module that a user can use to design and code a distributed application. The execution model of RADICAL-Pilot is as follows: 1. the user describes tasks as a set of CUs with or without data and DCR dependences; 2. the user also describes one or more pilots choosing the DCR(s) where they should be submitted to; 3. The Pilot Manager submits each pilot that has been described to the indicated DCR; 4. The CU Manager schedules each CU either to the pilot indicated in the CU description or on the first pilot with free and available resources; 5. when required, the CU Manager stages also the CU's input file(s) to the target DCR; and 6. the Agent executes the CU. Eventually, if needed the output files of the CU can be staged to a user-defined destination.

The Agent component of RADICAL-Pilot offers abstractions for both compute and data resources. Every Agent can expose between one and all the cores of the compute node where it is executed, and it can also expose a data handle that abstracts away specific storage properties and capabilities [59]. In this way, the CUs running on a Agent can benefit from unified interfaces both to core and data resources. Network resources are assumed to be available and no pilot abstraction is offered for their abstraction.

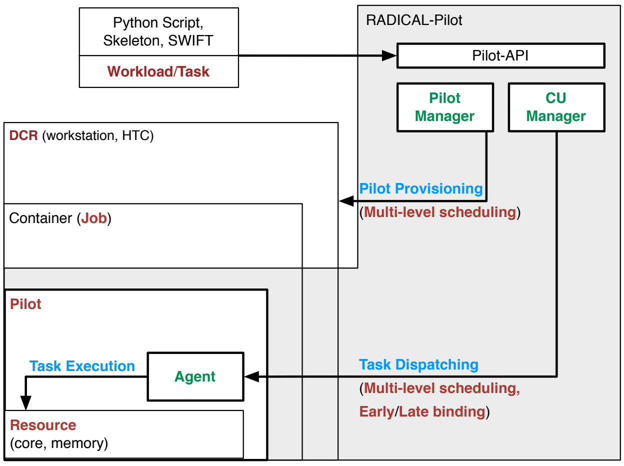

Figure 8: Diagrammatic representation of RADICAL Pilot components, functionalities, and core vocabulary mapped on Figure 4.

The Pilot Manager deploys the Agents of RADICAL-Pilot by means of the SAGA-python API [60]. SAGA provides access to diverse DCR middleware via an unified and coherent API, and thus RADICAL-Pilot can submit pilots to resources exposed by XSEDE and NERSC [100], by the OSG HTCondor pools, and many "leadership" class systems like those managed by OLCF [101], or NCSA [102].

The resulting separation of agent deployment from DCR architecture makes the addition of support for a DCR relatively simple **??**. This is demonstrated by the relative ease with which RADICAL-Pilot is extended to support (i) a new DCR, such as IaaS, and (ii) DCR that have different middleware but essentially similar architecture, for example the Cray supercomputers operated in the US and Europe.

RADICAL-Pilot can execute tasks with varying degree of coupling and communication requirements. Tasks can be completely independent, single or multi-threaded; they may be loosely coupled requiring input and output files dependences, or they might be tightly coupled requiring low-latency runtime communication. As such, RADICAL-Pilot supports MPI applications, workflows, and diverse execution patterns such as simulation/analysis, Replica Exchange simulations, or pipelines [73].

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

4.3 Implementation Discussion

The analysis offered in the previous subsection shows how diverse Pilot system implementations conform to the set of components and functionalities defined in §3. The analysis of DIANE, HTCondor Glidein (and GlideinWMS), PANDA, and RADICAL-Pilot also highlighted implementation differences, especially concerning the auxiliary properties of these Pilot systems. For example, based on the description of the auxiliary properties offered in §4.1.2, the described Pilot systems showed not only evident variations in their architectures and data management capabilities, but also in their DCR interaction model, their interfaces, and interoperability.

Clarifying the differences among Pilot systems implementations offers insight into how and when a specific implementation should be adopted, or possibly adapted. Understanding the differences will also help appreciate the challenges in implementing them.

Here these differences are described, widening the scope of the analysis to other relevant auxiliary properties, and to thereby highlight the challenges in implementing them.

The Pilot systems described in the previous subsection offer a varying degree of interoperability across diverse DCRs. For example, DIANE and PANDA were initially designed to support mostly grid-based DCRs used by the LHC experiments, while Glidein to execute tasks on the HTCondor-based DCRs. With the progressive development of both the Pilot systems and DCR middleware, more types of DCR are being supported. GlideIn, PANDA, and RADICAL-Pilot support diverse DCR middleware including typical HPC, grid, and cloud batch systems. This in part reinforces the importance of well-defined and robust interoperability layer between the Pilot system and heterogeneous target DCRs. As seen in §4, this flexibility is possible due to the generality of the notion of resource placeholder and its independence from the specificities of the target infrastructures and their middleware. Placeholders abstract the notions of resources, scheduling, and task execution, creating a well-defined and isolated logical space for the management of task execution. It remains to be understood under what conditions and for what type of application each implementation of a placeholder is more appropriate.

Each Pilot system described in the previous subsection exposes a different interface. DIANE, Glidein, and PANDA offer command line tools tailored to specific use cases, applications, and DCRs. RADICAL-Pilot exposes an API; DIANE and PANDA command line tools are built on API that users may directly access and use to develop distributed applications. This degree of heterogeneity can be justified with special-purpose Pilot systems but it may lead to fragmentation and duplication of effort with general-purpose systems.

Open APIs like the one specified in [45] and implemented by RADICAL-Pilot offer the missing layer on which workflow and, more generally, distributed applications could be built upon. Pilot systems 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 and tooling landscape.

Finally, DIANE, Glidein, PANDA, and RADICAL-Pilot implement different types of authentication, authorization, and accounting processes (AAA). Nonetheless, the AAA requirements specific to Pilot system implementations can be limited to: (i) the single or multitenancy of the instantiated pilots; and (ii) the credentials used to submit pilots to the DCRs.

The AAA required by the user to access her own pilots varies depending on the pilot's tenancy. With single tenancy, the pilot can be accessed only by the user that submitted it. As such, AAA can be based on inherited privileges. With multitenancy, the Pilot system has to evaluate whether a user requesting access to a pilot is part of the group of allowed users. Among the described Pilot systems, HTCondor Glidein and GlideinWMS are the only two implementing pilot multitenancy and using advanced abstractions like VOs and federated certificate authorities [103].

The credential used for pilot deployment depends on the target DCR. The AAA requirements of DCRs are a diverse and often inconsistent array of mechanisms and policies. Pilot systems are gregarious in the face of such a diversity as they only need to present the credentials provided by the application layer (or directly by the user) to the DCR. As such, the requirements for AAA implementation within Pilot systems are minimal.

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 them. 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 the crux of the concept 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 special purpose solutions to speed up the execution of a certain type of workload. The section closes with a look into the future of Pilot systems moving from the current state of the art and discussing the engineering and sociotechnical challenges that are being faced both by developers and target users.

5.1 The Pilot Paradigm

The generality of the Pilot paradigm may come as a surprise when considering the requirement that has motivated most implementations, viz., to increase the execution throughput of large workloads made of short running tasks. 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 DCR interoperability, support for multiple types of workloads, or flexibility in the optimization of execution are introduced. Satisfying those requirements demands abstracting the specificity of application patterns, of the middleware and architectures of DCRs, and of the types of resources.

The generality of the Pilot paradigm and the independence of its implementations from other types of software, clarify also a potential misunderstanding: Pilots are not just an alternative implementation of a scheduler, a workload manager, a batch system, or any other special-purpose software component. Pilot systems represents a category of software in itself.

The Pilot paradigm is general because it does not strictly depend on a single type of workload, a specific DCR, or a unique performance metric. In principle, systems implementing the Pilot paradigm can execute workloads composed of 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 types of DCR and of resource on which a Pilot system can execute workloads. The descriptions presented in §4.2, showed how Pilot systems already operate on diverse DCRs. Originally devised for HTC grid infrastructures, Pilot systems have been (re-)engineered to operate also on HPC and Cloud infrastructures.

As seen in §3, the Pilot paradigm demands resource placeholders but does not specify the type of resource that the placeholder should expose. In principle, pilots can be placeholders also and exclusively for data or network resources. For example, in Ref. [104, 105] the concept of Pilot-Data was conceived to be fundamental to dynamic data placement and scheduling as Pilot is to computational tasks.

Traditionally, Pilots have been thought of as a means to optimize the throughput of single-core (or at least single-node), short-lived, uncoupled tasks execution [106, 107, 108]. The analysis presented in §4 showed that such a view is restrictive: Pilot systems can be used to optimize diverse types of workloads along multiple performance dimensions. For example, Pilot systems have been successfully integrated within workflow systems to support optimal execution of workloads with articulated data and single or multi-core task dependencies. As such, not only throughput can be optimized for multi-core, long-lived, coupled tasks executions but optimal data/compute placement, and dynamic resource sizing can be achieved.

Thanks to the generality of the Pilot paradigm in respect of types of workload, target DCR, and resource, Pilot systems 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, namely the main sources of duplication and fragmentation in the current distributed application and tooling landscape.

The generality of the pilot paradigm across workload, DCR, and resource types was first discussed in Ref. [98], wherein an initial conceptual model for Pilot systems was proposed. This paper significantly enhances and extends that preliminary analysis.

5.2 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 can 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 were developed to serve a specific use case, e.g., they emerged as a pragmatic solution for improving the throughput of distributed applications, and designed as local and point solutions. Pilot systems were not thought from their inception as an independent and well-defined system, but, at best, as a module within a specific framework. Pilot systems also inherited the development model of the scientific projects within which they were initially developed. As a consequence, these systems were not engineered to promote (re)usability, modularity, well-defined interfaces, or long-term sustainability. Collectively, 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.

The analysis offered in this paper indicates that the number of Pilot systems actively developed can be reduced so to avoid duplication while promoting consolidation, robustness, and overall capabilities. Nonetheless, this conclusion should not be taken to an extreme. A single Pilot system should not be elected as the only implementation worthy of development effort or adoption. As with other software systems and middleware [109] 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 with an unique performance objective, a special purpose system might be warranted, but otherwise a conservative approach would be more efficient and 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.

As argued in the previous section, Pilot systems should be agnostic towards the type of application that is executed. They can be engineered so as to support different types of applications and application objectives. Pilots hold resources and the properties of such resources should also be left open to arbitrarily specification. Thus the the type of matching between the requirements of tasks and the capabilities offered by the resources held by the pilot should not be mandated. For this reason, adopting the Pilot paradigm for the execution of increasingly diverse applications on new infrastructure or different resources should be seen as a set of often challenging implementation details more than a foundational issue requiring new paradigms.

This can be evidenced by the emergence of frameworks that encapsulate the Pilot paradigm. Hadoop 2 [110], the second version of the affirmed infrastructure for data-intensive computing, introduced the YARN [111] resource manager to heterogeneous workloads. YARN supports multi-stage scheduling: Applications need to initialize their so-called 'Application-Master' 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. TEZ [112] 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. Independent of the Hadoop developments, Google's Kubernetes [113] is emerging as an important container management approach, which not completely coincidently is Greek for the English term 'Pilot'.

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 [105].

5.3 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 abstraction 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 development process 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. Tables 1 and 2 summarize these contributions and 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.

The current state of the workflow systems [39] is a paradigmatic example of the consequence of a lack of conceptual clarity: Many workflow 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 workflow 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.

This paper establishes the the generality of Pilot paradigm and shows that a more structured approach is needed to the conceptualization and design of its software systems. The generality of this paradigm along the types of workload, resource, and performance indicates the fundamental role that the it can play to support task-level parallelism at higher scales and on possibly multiple and diverse DCRs. If appreciated, the contributions of this paper offer an analytical base for the improvement of exiting Pilot systems implementations, improving their interoperability across specific DCR and curtailing the need to create unsustainable partial implementations.

.

Acknowledgements

This work is funded by the Department of Energy Award (ASCR) DE-FG02-12ER26115 and NSF CAREER ACI-1253644. We thank the many members of the RADICAL group – former and current, for helpful discussions, comments and criticisms.

6. REFERENCES

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

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

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

[4] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, *et al.*, "DIRAC pilot framework and the DIRAC Workload Management System," in *Journal of Physics: Conference Series*, vol. 219(6), p. 062049, IOP Publishing, 2010.

[5] P.-H. Chiu and M. Potekhin, "Pilot factory – a Condor-based system for scalable Pilot Job generation in the Panda WMS framework," in *Journal of Physics: Conference Series*, vol. 219(6), p. 062041, IOP Publishing, 2010.

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

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

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

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

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

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

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

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

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

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

[16] Large Hadron Collider (LHC), homepage, `http://home.web.cern.ch/topics/large-hadron-collider`.

[17] T. LHC Study Group *et al.*, "The large hadron collider, conceptual design," tech. rep., CERN/AC/95-05 (LHC) Geneva, 1995.

[18] Worldwide LHC Computing Grid (WLCG), homepage, `http://wlcg.web.cern.ch/`.

[19] D. Bonacorsi, T. Ferrari, *et al.*, "Wlcg service challenges and tiered architecture in the lhc era," *IFAE 2006*, pp. 365–368, 2007.

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

[21] The LHCb experiment, `http://lhcb-public.web.cern.ch/lhcb-public/`.

[22] P. Lynch, "Richardson's marvelous forecast," in *The life cycles of extratropical cyclones*, pp. 61–73, Springer, 1999.

[23] L. Richardson, "Weather prediction by numerical process cambridge university press," *CambridgeRichardsonWeather prediction by numerical process1922*, 1922.

[24] 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*, pp. 62–82, Springer, 1998.

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

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

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

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

[29] 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*, pp. 301–308, 2004.

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

[31] 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, (New York, NY, USA), pp. 46:1–46:8, ACM, 2013.

[32] 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, (New York, NY, USA), pp. 50:1–50:8, ACM, 2013.

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

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

[35] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE'08*, pp. 1–10, Ieee, 2008.

[36] G. Juve and E. Deelman, "Resource provisioning options for large-scale scientific workflows," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pp. 608–613, IEEE, 2008.

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

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

[39] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: scientific workflows for grids*. Springer Publishing Company, Incorporated, 2014.

[40] V. Curcin and M. Ghanem, "Scientific workflow systems-can one size fit all?," in *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pp. 1–9, IEEE, 2008.

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

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

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

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

[45] A. Luckow, M. Santcroos, O. Weidner, A. Merzky, S. Maddineni, and S. Jha, "Towards a common model for pilot-jobs," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 123–124, ACM, 2012.

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

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

[48] G. Woltman, S. Kurowski, *et al.*, "The great Internet Mersenne prime search," *Online],(1997, March 23) available* `http://www.mersenne.org`, 2004.

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

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

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

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

[53] D. Weitzel, D. Fraser, B. Bockelman, and D. Swanson, "Campus grids: Bringing additional computational resources to HEP researchers," in *Journal of Physics: Conference Series*, vol. 396(3), p. 032116, IOP Publishing, 2012.

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

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

[56] P. Buncic and A. Harutyunyan, "Co-Pilot: The distributed job execution framework," tech. rep., CERN, 03 2011.

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

[58] I. Sfiligoi, "glideinWMS—a generic pilot-based workload management system," in *Journal of Physics: Conference Series*, vol. 119(6), p. 062044, IOP Publishing, 2008.

[59] 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*, pp. 135–144, IEEE, 2010.

[60] A. Merzky, O. Weidner, and S. Jha, "Saga: A standardized access layer to heterogeneous distributed computing infrastructure," *SoftwareX*, 2015.

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

[62] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Radical-pilot: A scalable implementation of the pilot abstraction," 2015. Under review. Draft available at: `http://radical.rutgers.edu/publications/rp-paper-2015`.

[63] M. Rynge, G. Juve, G. Mehta, E. Deelman, G. B.

Berriman, K. Larson, B. Holzman, S. Callaghan, I. Sfiligoi, and F. Würthwein, “Experiences using glideinWMS and the corral frontend across cyberinfrastructures,” in *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pp. 311–318, IEEE, 2011.

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

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

[66] D. Bernstein, “Containers and cloud: From LXC to docker to Kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[67] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” *technology*, vol. 28, p. 32, 2014.

[68] C. Pinchak, P. Lu, and M. Goldenberg, “Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences,” in *Job Scheduling Strategies for Parallel Processing*, pp. 205–228, Springer, 2002.

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

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

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

[72] J. Frey, “Condor DAGMan: Handling inter-job dependencies,” *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

[73] Ensemble MD Toolkit, `http://radical-cybertools.github.io/ensemble-md/index.html`.

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

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

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

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

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

[79] DIANE API and reference documentation, `http://it-proj-diane.web.cern.ch/it-proj-diane/install/2.4/doc/reference/html/index.html`.

[80] J. T. Mościcki, “Understanding and mastering dynamics in computing grids: processing moldable tasks with user-level overlay,” *UvA-DARE (Digital Academic Repository)*, 2011.

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

[82] GANGA homepage, `https://ganga.web.cern.ch/`.

[83] G. Grzeslo, T. Szepieniec, and M. Bubak, “DAG4DIANE-enabling DAG-based applications on DIANE framework,” *CGW Book of Abstracts*, 2009.

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

[85] HTCondor Manual v7.6.10, Glidein, `http://research.cs.wisc.edu/htcondor/manual/v7.6.10/5_4Glidein.html`.

[86] G. Juve, “The Glidein service.” Presentation, `http://www.slideserve.com/embed/5100433`.

[87] GlideinWMS homepage, `http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/doc.prd/index.html`.

[88] Glidein based WMS, manual, `http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/doc.v1_0/manual/index.html`.

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

[90] HTCondor ClassAd, `http://research.cs.wisc.edu/htcondor/manual/v7.8/4_1HTCondor_s_ClassAd.html`.

[91] T. Maeno, K. De, and S. Panitkin, “PD2P: PanDA dynamic data placement for ATLAS,” in *Journal of Physics: Conference Series*, vol. 396(3), p. 032070, IOP Publishing, 2012.

[92] 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*, vol. 396(3), p. 032080, IOP Publishing, 2012.

[93] PanDA Architecture,

https://twiki.cern.ch/twiki/bin/view/PanDA/PanDA#Architecture_and_workflow.

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

[95] P. Nilsson, J. Caballero, K. De, T. Maeno, A. Stradling, T. Wenaus, *et al.*, "The ATLAS PanDA pilot in operation," in *Journal of Physics: Conference Series*, vol. 331(6), p. 062040, IOP Publishing, 2011.

[96] PandaRun, https://twiki.cern.ch/twiki/bin/view/PanDA/PandaRun.

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

[98] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, "$P*$: a model of pilot-abstractions," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pp. 1–10, IEEE, 2012.

[99] RADICAL-Pilot, http://radical-cybertools.github.io/radical-pilot/index.html.

[100] National Energy Research Scientific Computing Center (NERSC), https://www.nersc.gov/.

[101] Oak Ridge National Laboratory (OLCF), https://www.olcf.ornl.gov/.

[102] National Centre for Supercompunting Applications (NCSA), http://www.ncsa.illinois.edu/.

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

[104] A. Luckow, M. Santcroos, A. Zebrowski, and S. Jha, "Pilot-data: an abstraction for distributed data," *Journal of Parallel and Distributed Computing*, 2014.

[105] 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*, pp. 1–2, IEEE, 2012.

[106] 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*, vol. 78(1), p. 012057, IOP Publishing, 2007.

[107] 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*, vol. 2, pp. 428–432, IEEE, 2009.

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

[109] P. A. Bernstein, "Middleware: a model for distributed system services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996.

[110] PandaRun, http://hadoop.apache.org/.

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

[112] Apache Tez, https://tez.apache.org/.

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