A Comprehensive Perspective on Pilot-Abstraction

ABSTRACT

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2015 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

1. INTRODUCTION

The seamless uptake of distributed infrastructures by scientific applications has been limited by the lack of pervasive and simple-to-use abstractions at the development, deployment, and execution level. Of all the abstractions proposed to support effective distributed resource utilization, a survey of actual usage suggested that Pilot-Job ***matteo: Should we use just 'pilot' is arguably one of the most widely-used distributed computing abstractions - as measured by the number and types of applications that use them, as well as the number of production distributed cyberinfrastructures that support them. ***mark: ref?

The fundamental reason for the success of the Pilot-Job abstraction is that Pilot-Jobs facilitate the oterwise challenging mapping of specific tasks onto explicit heterogeneous and dynamic resource pools. Pilot-Jobs decouple the workload specification from the task management improving the efficiency of task assignment while shielding applications from having to manage tasks across such resources. ***Ole: not sure if 'load-balance' is appropriate here ***matteo: Changed into the more generic 'manage' Another concern often addressed by Pilot-Jobs is fault tolerance which commonly refers to the ability of the Pilot-Job system to verify the execution environment before executing jobs. The Pilot-Job abstraction is also a promising route to address specific requirements of distributed scientific applications, such as coupled-execution and application-level scheduling [21, 20].

A variety of PJ frameworks have emerged: Condor-

G/ Glide-in [16], Swift [41], DIANE [25], DIRAC [9], PanDA [10], ToPoS [37], Nimrod/G [8], Falkon [31] and MyCluster [39] to name a few. Although they are all, for the most part, functionally equivalent – they support the decoupling of workload submission from resource assignment – it is often impossible to use them interoperably or even just to compare them functionally or qualitatively. The situation is reminiscent of the proliferation of functionally similar yet incompatible workflow systems, where in spite of significant *a posteriori* effort on workflow system extensibility and interoperability, these objectives remain difficult if not infeasible. ***matteo: Should we have a paragraph explaining the core contribution offered by this paper?

The remainder of this paper is divided into four Sections. §2 offers a critical review of how the concept of Pilot-Jobs has evolved by analyzing existing Pilot systems and systems with pilot-like characteristics. In §3, the hetorogeneity described in §2 is addressed by deriving the minimal set of capabilities and properties that has to characterize the design of a Pilot system. A vocabulary is then defined so that it can be used consistently across different designs and implmentations of a Pilot system. In §4, the focus shifts from analysing the design of a Pilot system to leveraging the terminology defined in §3 to critically reviewing the characteritics and functionalities of a relevant set of Pilot system implementations. Finally, §5 closes the paper by outlining the Pilot paradigm and elaborating on how it impacts and relates to other middleware and the application layer. The generality and breath of the Pilot paradigm is underlined by showing its adoption by the Enterprise, outside the boundaries of scientific research.

2. FUNCTIONAL EVOLUTION OF PILOT-JOBS

The origin and motivations for devising the Pilot abstraction and developing its many implementations can be traced back to five main notions: task-level distribution and parallelism, Master-Worker (M-W) pattern, multi-tenancy, multi-level scheduling, and resource placeholder. This section offers an overview of these five notions and an analysis of their relationship with the Pilot abstraction. A chronological perspective is taken so to contextualize the inception and evolution of the Pilot abstraction.

At the best of the authors' knowledge, the term 'Pilot-Job' was first introduced around 2004 in the context of the Large Hadron Collider (LHC) challenge [ref] and then used in a 2005 LHCb report[**?**]. Despite its relatively recent naming, the Pilot abstraction addresses a problem already well-known at the beginning of the twentieth century: task-level distribution and parallelism on multi-tenant resources.

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

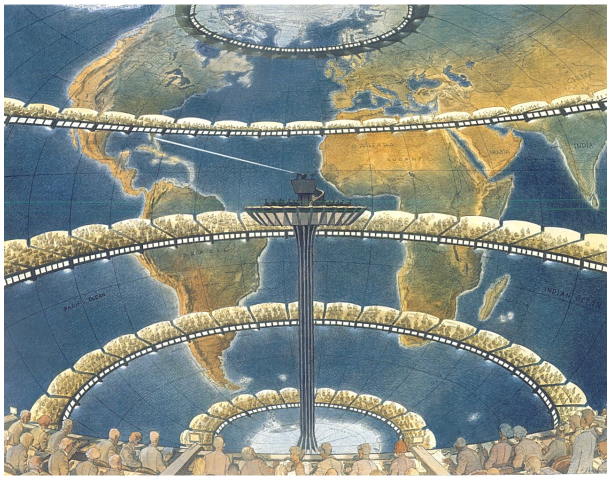


Figure 1: *Forecast Factory* as envisioned by Lewis Fry Richardson. Drawing by François Schuiten.

The clerk of the Forecast Factory is the 'master' while the human computers are her 'workers'. Requests and responses go back and forth between the master and all its workers. Each worker has no information about the overall computation nor about the states of any other worker. The master is the only one possessing a global view both of the overall problem to compute and of its progress towards a solution. As such, the M-W 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 (TTC) of the overall computation when compared to a coordination pattern in which each equation is sequentially solved by a single worker.

***shantenu: Mark: I propose the following: One common use for the M-W scheme is to serve as the coordination substrate for PJs. I know its a bit nebulous, but it connects the two concepts directly, which is the goal here. ***mark: Interesting contrast. Do we see M/W as a communication pattern to implement PJs or do PJs enable the M/W pattern to applicatons? ***matteo: Please see §3.1, 10th paragraph: "As see in..."

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

Multi-tenancy has been shaping the way in which high performance computing resources are exposed to each user. Leveraging the batch processing concept, first used in the time of punch cards [ref], job schedulers, often called "batch queuing systems" [ref], were created to promote efficient and fair resource sharing. This type of system implements a usability model where users submit computational tasks called 'jobs' to a queue managed by the supercomputer scheduler.

The execution of these jobs is therefore delayed from the moment of their submission, the amount of delay mostly depending on the size and duration of the submitted job, resource availability, and fair usage policies.

The resource provision of multi-tenant and heterogeneous supercomputers is limited, irregular, and largely unpredictable [ref, red]. 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 [ref]. 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.

Supercomputers present also several types of heterogeneity and diversity. While each supercomputer assumes an analogous, queue-based usability model, its implementation varies from resource to resource. Users are faced with different job description languages, job submission commands, and job configuration options. Furthermore, the number of queues exposed to the users and their properties like walltime, duration, and compute-node sharing policies vary from resource to resource. Finally, each supercomputer may be designed and configured to support only specific types of application.

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

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

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

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

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

Resource placeholders bring together multi-level scheduling and M-W pattern 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 M-W pattern is a natural choice to manage the coordination of tasks execution on the available agent(s).

It should be noted that resource placeholders mitigate the side-effects introduced by multi-tenancy, they do not require special super-user privileges on the remote machines, and have the potential to load balance the execution of distributed applications across multiple supercomputers. Multi-tenancy affects only the scheduling of the agent as it needs to be executed by the batch system of the remote supercomputer. Once the agent is executed, the user – or the master process of the distributed application – may hold total control of the resource placeholder. In this way, tasks are directly scheduled on the agent without competing with other users for the supercomputer scheduler.

Resource placeholders and their scheduling [30] were early Pilot mechanisms. As placeholder scheduling evolved, it came to include dynamic monitoring and throttling of the different placeholders based on the queue times on the remote machines. The progressive definition and implementation of the Pilot abstraction can be seen as the process of evolving both the understanding and implementation complexity of the notion of resource placeholder.

AppLeS [7] 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 enabling the application to acquire resources and to schedule efficiently tasks onto these. Besides M-W, AppLeS provides also different application templates, e. g. for parameter sweep and moldable parallel applications [6].

AppLeS gave 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 translated directly into a change of the application code. The next evolutionary step was to create a dedicated abstraction layer between those of the application and of the various batch queuing systems available at different remote systems.

Around the same time as AppLeS was introduced, volunteer computing projects started using the M-W coordination pattern to achieve high-throughput calculations for a wide

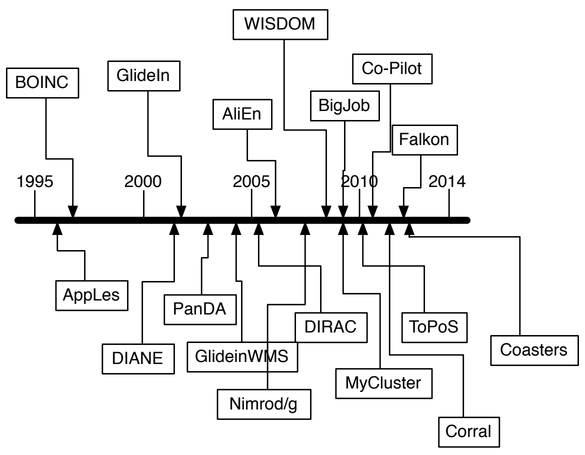
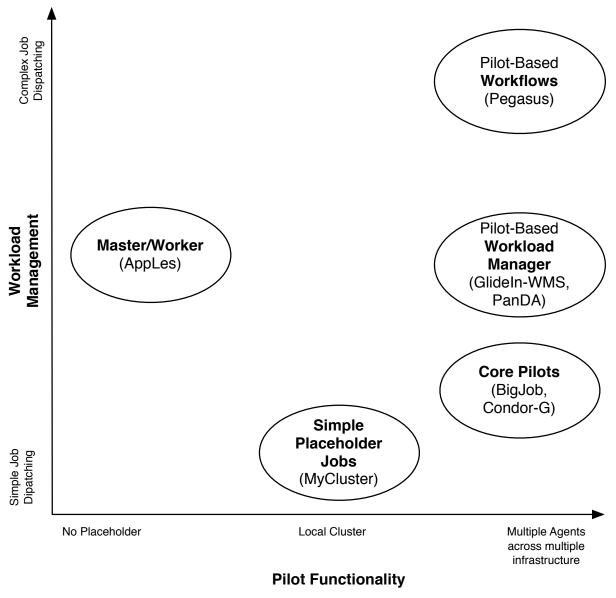


Figure 2: Introduction of systems over time. When available, the date of first mention in a publication or otherwise the release date of software implementation is used.



Complex Job Dispatching
Workload Management
Simple Job Dipatching
Pilot-Based **Workflows** (Pegasus)
Master/Worker (AppLes)
Pilot-Based **Workload Manager** (GlideIn-WMS, PanDA)
Core Pilots (BigJob, Condor-G)
Simple Placeholder Jobs (MyCluster)
No Placeholder
Local Cluster
Multiple Agents across multiple infrastructure
Pilot Functionality

Figure 3: Pilot-Job Clustering

range of scientific problems. These volunteer workers could potentially be distributed at global scale, users only had to download a client program on their laptops and workstations and the workers would have taken care of pulling the computation tasks and executing on the local machine when CPU cycles were available.

The volunteer workers were essentially heterogeneous and dynamic as opposed to the homogeneous and static nature of the AppLeS workers. The idea of farming out tasks in a distributed environment including personal computers promised to lower the complexity barrier associated to distributed applications design and implementation. Furthermore, volunteering computing was consistent with the resource placeholder abstraction, and as such represented an evolutionary step of the notion of Pilot.

The first public volunteer computing projects were The Great Internet Mersenne Prime Search effort[42], shortly followed by distributed.net [22] 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 our of SETI@Home, becoming the *de facto* standard framework for voluntary computing [4].

It should be noted that while in the AppLes the acquisition of resources by means of resource placeholders was centrally planned, the resources in the volunteer computing are acquired dynamically, depending on their unpredictable availability. The master does not know in advance which resources could come available at any given point in time, and therefore there is no advance allocation of tasks to a specific worker. This in contrast with those cases in which an orchestrated set of resources is requested in advance by the master. In these cases, the master can divide the tasks over the workers following a predefined criterion.

***matteo: Connect Condor to static/dynamic resource placeholder allocation

Condor is a high-throughput distributed batch computing system. Originally Condor was created for systems within one administrative domain. With Flocking [14], it was possible to group multiple Condor systems (pools) together so that their resources could be used in an aggregative manner. However, this mode of operation required system level software configuration by the owners of the individual Condor resources and thus required administrator cooperation, and could not simply be done on application level by a user.

Condor-G/GlideIn [16] is one of the pioneers of the Pilot-Job concept. GlideIn is a mechanism by which a user can add remote grid resources to the local condor pool and run their jobs on the added remote resource in the same way that local condor jobs are submitted.

The resources added are available only for the user who added the resource to the pool, thus giving complete control over the resources for managing jobs without any queue waiting time. Various systems that built on the Pilot capabilities of Condor-G/GlideIn have been developed, e. g. Bosco [40].

BigJob was designed as a flexible and extensible Pilot-Job to work on a variety of infrastructures through the use of the Simple API for Grid Applications (SAGA). BigJob was designed to support MPI jobs without adding additional configuration requirements to the end-user. The application-level programmability that BigJob offers was incorporated as a means of giving the end-user more flexibility and control over their job management. BigJob has been extended to work with data and, analogous to Pilot-Jobs, abstract away direct user communication between different storage systems. This work has extended BigJob from being a purely Pilot-Job-based system to a more complete job and data management system.

While the aforementioned systems mainly focus on providing simple Pilot capabilities commonly in application space, many of these systems evolved towards or were extended to Pilot-based workload managers. These higher-level systems which are often centrally hosted, move critical functionality from the client to the server (i.e. a service model). These systems usually deploy Pilot factories that automatically start new Pilots on demand and integrate security mechanisms to support multiple users simultaneously.

Several of these have been developed in the context of the LHC experiment at CERN, which is associated with a major increase in the uptake and availability of Pilots, e. g. GlideInWMS, DIRAC [9], PanDa [43], AliEn [5] and Co-Pilot [1]. Each of these Pilots serves a particular user community and experiment. Interestingly, we observe that these Pilots are functionally very similar, work on almost the same underlying infrastructure, and serve applications with very similar (if not identical) characteristics.

GlideinWMS [35] is a higher-level workload management system that is based on the Pilot capabilities of Condor Glide-in. The system can, based on the current and expected number of jobs in the pool, automatically increase or decrease the number of active Glide-ins (Pilots) available to the pool. GlideinWMS is a multi-user Pilot-Job system commonly deployed as a hosted service. GlideinWMS attempts to hide from rather than expose the Pilot capabilities to the user. GlideinWMS is currently deployed in production on the Open Science Grid (OSG) [28] and is the recommended mode for new users wanting to access OSG resources.

PanDA (Production and Distributed Analysis) [43] is the workload management system of the ATLAS experiment, used to run managed production and user analysis jobs on the grid. The ATLAS Computing Facility operates the pilot submission systems. This is done using the PanDA "AutoPilot" scheduler component which submits pilot jobs via Condor-G. PanDA also provides the ability to manage data associated the jobs managed by the PanDA workload manager.

In addition to processing, AliEn [5] also provides the ability to tightly integrate storage and compute resources and is also able to manage file replicas. While all data can be accessed from anywhere, the scheduler is aware of data localities and attempts to schedule compute close to the data. AliEn deploys a pull-based model [34] with the argument that the resource pool is dynamic and that a pull model doesn't require the broker to keep detailed track of all resources, which leads to a more simple implementation.

DIRAC [9] is another comprehensive workload management system built on top of Pilots. It supports the management of data, which can be placed in different kinds of storage elements (e. g. based on SRM).

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

In the context of scientific workflows, Pilot-Job systems have proven an effective tool for managing the workload. For the Pegasus project, the Corral system [33] was developed to support the requirements of the Pegasus workflow system in particular to optimize the placements of Pilots with respect to their workload. It did this by serving as a front-end to Condor Glidein. In contrast to GlideinWMS, Corral provides more explicit control over the placement and start of Pilots to the end-user. Corral was later extended to also serve as a possible frontend to GlideinWMS.

Swift [41] is a scripting language designed for expressing abstract workflows and computations. The language provides among many things capabilities for executing external application as well as the implicit management of data flows between application tasks. The Coaster system [11] has been developed to address the workload management requirements of Swift by supporting various infrastructures, including clouds and grids. Swift has also been used in conjunction with Falkon [31]. Falkon was engineered for executing many small tasks on HPC systems and shows high performance compared to the native queuing systems. Falkon is further a good example of the use of the pilot abstraction for a very specific workload.

Based on the descriptions in this section we can identify some distinctions in terms of design, usage and operation of pilot(-based) systems. Figure 3 is a graphical representation of this rough clustering.

The evolution of Pilot-Jobs 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.

3. UNDERSTANDING THE LANDSCAPE: DEVELOPING A VOCABULARY

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

This section offers an analysis of the logical components and functionalities shared by every Pilot system. The goal is to offer a paradigmatic description of a Pilot system and a well-defined vocabulary to reason about such a description and, eventually, about its multiple implementations.

3.1 Logical Components and Functionalities

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

The purpose shared by every Pilot system is to improve the (performance of) workload execution when compared to executing the same workload directly on one or more DCI. Performance in Pilot systems is usually associated to throughput and execution time-to-completion (cit), but other metrics could be also considered as, for example, energy efficiency, data transfer, scale of the workload executed or a mix of them. In order to maximize or minimize the chosen set of metrics, each Pilot system exhibits characteristics that are both common or specific to one or more implementations. Discerning these characteristics requires isolating and defining the minimal set of logical components that has to characterize every Pilot system.

At some level, all Pilot systems leverage three separate but coordinated logical components: a **Pilot Manager**, a

Workload Manager, and a **Task Executor** (see Table 1). The Pilot Manager handles the definition, instantiation, and use of one or more resource placeholders – called 'pilot' – on single or multiple DCIs. The Workload Manager handles the inspection and submission of one or more workload on the available resource placeholders. Finally, the Task Executor takes care of executing the tasks of each workload by means of the resources held by the placeholders.

The implementation details of these three logical components significantly vary across Pilot system implementations (see §4). One or more components may be responsible for specific functionalities, both on application as well as infrastructure level, two or more components may be implemented in a single module, or additional components may be integrated into the managers and executor. Nevertheless, the Pilot and Workload Managers and the Task Executor can always be distinguished across different Pilot systems. For example, looking at the systems mentioned in §2, ... ***matteo: I will wait for S2 and S4 to be final before picking the right examples/vocabulary.

The logical components that characterize a Pilot system support a minimal set of functionalities that allow for the execution of workloads: **Pilot Provisioning**, **Task Dispatching**, and **Task Execution** (see Table 1). Every Pilot system needs to schedule resource placeholders on the targeted resources, schedule tasks on the available placeholders and then leverage these placeholders to execute the tasks of the given workload. Clearly, many more functionalities are needed in order to implement a production-grade Pilot system. For example, authentication, authorization, and accounting (AAA) are fundamental when exposing a Pilot system on a public network infrastructure, complex data management is required by specific type of workloads, and fault-tolerance and load-balancing play an important role on specific type of DCIs. While these functionalities are critical implementation details, they depend on the specific characteristics of the given user, workload, or targeted resources and, as such, they should not be considered a necessary trait of all the Pilot systems.

Among the core functionalities that characterize every Pilot system, Pilot Provisioning is essential because it allows for the creation of resource placeholders. This type of placeholder enables tasks to utilize resources without directly depending on the capabilities exposed by the targeted DCI. Resource placeholders are scheduled to the DCI resources by means of the DCI capabilities but, once scheduled and then executed, these placeholders make their resources directly available for the execution of the tasks of a workload. Resource placeholders belong to the user(s) that created them and they can be tailored and used so to satisfy the user's specific requirements. Furthermore, resource placeholders are logical partitions of resources that do not need to leverage trade-offs among competing user requirements as needed instead with large pools of resources adopting multi-tenancy.

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

Once resource placeholders are bound to a DCI, tasks need to be dispatched to those placeholders for execution. Task dispatching does not depend on the functionalities of the DCI middleware so it can be implemented as part of the Pilot system. In this way, the control over the execution of a workload is shifted from the DCI to the Pilot system. This shift is fundamental because it decouples the execution of a workload from the need to submit its tasks via the DCI middleware. The tasks of a workload will not wait on the DCI queues before being executed, and complex execution patterns involving task and data interdependence will be available outside the boundaries of the DCI. Ultimately, this is why Pilot systems allow for the parametrization of workload execution and the maximization, for example, of execution throughput.

Communication and coordination are two distinguishing characteristics of distributed software tools and Pilot systems make no exception. The Workload Manager, Pilot Manager, and the Task Executor need to communicate and coordinate in order to instantiate resource placeholders, place tasks, and, in case, move data. For example, the Workload Manager needs to communicate with the Task Executor to coordinate the execution of the given workload. Analogously, the Workload and Pilot Managers need to communicate in order to coordinate the instantiation and utilization of resource placeholders. Nonetheless, Pilot systems are not defined by any specific communication pattern and coordination strategy. The logical components of a Pilot system may communicate with every suitable pattern – point-to-point (i.e. one-to-one), many-to-one, or one-to-many (cit) – and coordinate adopting any suitable strategy – time synchronization, static or dynamic coordinator election, local or global information sharing, or Master-Worker. The same applies to specific network architectures and protocols. Different network architectures and protocols may be leveraged within a Pilot implementation in order to achieve effective communication and coordination. ***shantenu: maybe this is a good place to introduce pull vs push models

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

Data management may have an important role within Pilot systems. For example, functionalities can be provided to support the local or remote data staging required to execute the tasks of a workload, or data might be managed according to the specific capabilities offered by the targeted DCI. Pilot systems can be devised in which tasks do not require any data management because they (i) do not necessitate input files, (ii) do not produce output files, (iii) data is already locally available or (iv) data management is left to the application itself. Being able to read and write files to a local filesystem should then be considered the minimal capability related to data required by a Pilot system. More advanced and specific data capabilities like, for example, data replication, concurrent data transfers, data abstractions other than files and folders, or data placeholders should be considered special-purpose capabilities, not characteristic of every Pilot system.

In the following Subsection, a minimal set of terms related to the logical components and capabilities just described is defined.

3.2 Terms and Definitions

The terms 'pilot' and 'job' are arguably among the most prominent when referring to Pilot systems. It is the case that Pilot systems are commonly referred to as 'Pilot-Job systems' (cit, cit, cit, cit), a clear indication of the primary role played by the concepts of 'pilot' and 'job' in this type of system. The definition of both concepts is context-dependent and several other terms need to be clarified in order to offer a coherent terminology. Both 'job' and 'pilot' need to be understood in the context of DCIs, the infrastructures leveraged by Pilot systems. DCIs offer compute, storage, and network resources and Pilot allow for the users to utilize those resources to execute the tasks of one or more workload.

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

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

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

Infrastructure or DCI. Structured set of resources, possibly geographically and institutionally separated from the users that utilize those resources to execute the tasks of a workload (cit, cit). Infrastructures can be logically partitioned, with a direct or indirect mapping onto individual pools of hardware. In the context of Pilot systems, an infrastructure is usually to be assumed to be a distributed computed infrastructure, i.e., a DCI.

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

Job. Functionally defined as a 'task' from the perspective of the DCI, but indicating the type of container required to acquire resources on a specific infrastructure.

When considering Pilot systems, jobs and tasks are functionally analogous but qualitatively different. Functionally, both jobs and tasks are containers – i.e. metadata wrappers around one or more executable often called 'kernel', 'application', or 'script'. Qualitatively, the term 'task' is used when reasoning about workloads while 'job' is used in relation to a specific type of infrastructure where such a container can be executed. Accordingly, tasks are considered as the functional units of a workload, while jobs as a way to execute executables on a certain infrastructure. It should be noted that, given their functional equivalence, the two terms can be adopted interchangeably when considered outside the context of Pilot systems. Indeed, workloads are encoded into jobs when they have to be directly executed on infrastructures that support or require that type of container.

As described in §3.1, a resource placeholder needs to be submitted to the target DCI wrapped in the type of container supported by that specific DCI. For example, for a DCI exposing a HPC or Grid middleware [cit, cit], a resource placeholder needs to be wrapped within a 'job'. For other type of DCIs, the same resource placeholder will need to be wrapped within a different type of container as, for example, a VM or a Docker Engine. For this reason, the capabilities exposed by the job submission system of the target DCI determine the submission process of resource placeholders and how or for how long they can be used. For example, when wrapped within a 'job', placeholders are provisioned by submitting a job to the DCI queuing system, become available only once the job is scheduled on the resources out of the queue, and is available only for the duration of the job walltime.

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

Pilot. The sum of a container – e.g., a job – plus a set of executables functioning as a resource placeholder on a given infrastructure and capable of executing tasks of a workload.

It should be noted that the term 'pilot' as defined here is named differently across Pilot systems. Depending upon context, in addition to the term 'placeholder', pilot is also named 'agent' and, in some cases, 'Pilot-Job' [cit]. All these terms are, in practice, used as synonyms without properly distinguishing between the type of container and the type of executable that compose a Pilot. This is a clear indication of how necessary the minimal and consistent vocabulary offered in this Subsection is when reasoning analytically about multiple implementations of a Pilot-Job system.

The term 'pilotjob' is often used to identify a Pilot system. This is an unfortunate choice as the term 'job' identifies just the way in which a pilot is provisioned on a DCI exposing specific capabilities, not a general properties of all the Pilot systems. The use of the term 'Pilot-Job system' should therefore be regarded as a historical artifact, viz., the targeting of a specific class of DCIs in which the term 'job' was, and still is, meaningful. With the development of new types of DCI middleware as, for example, cloud-based infrastruc-

Term	**Functionality**	**Logical Component**
Workload	Task Dispatching	Workload Manager
Task	Task Dispatching Task Execution	Workload Manager Task Executor
Resource	Pilot Provisioning	Pilot Manager
DCI	Pilot Provisioning	Pilot Manager
Job	Pilot Provisioning	Pilot Manager
Pilot	Pilot Provisioning Task Execution	Pilot Manager Task Executor
Multi-level scheduling	Pilot Provisioning Task Dispatching (Task Execution?)	Pilot Manager Workload Manager (Task Executor?)
Early binding	Task Dispatching Pilot Provisioning	Workload Manager Pilot Manager
Late binding	Task Dispatching Pilot Provisioning	Workload Manager Pilot Manager

Table 1: Mapping of the core terminology of Pilot systems into the functionalities and logical components described in §??. ***mark: What are the considerations for the mapping?

tures, the term 'job' has become too restrictive, a situation that can lead to terminological and conceptual confusion.

We have now defined resources, DCIs and Pilots. We have established that Pilots are a placeholder for a set of resources. The combined resources represented by Pilots that are available for the execution of a workload together form an resource overlay. This set of Pilots can potentially be distributed over multiple resources and/or DCIs. The details of this aggregation or federation is outside the scope of this paper.

Resource Overlay. The set of resources that are represented by one or more Pilots.

To promote internal consistency, having clarified the meaning of 'job' and 'pilot', it is imperative that three more terms fundamentally associated with Pilot systems (cit, cit, cit) be explicitly defined: 'Multi-level scheduling' and 'late/early binding'.

Pilot systems are said to implement multi-level scheduling because they require the scheduling of two types of entities: pilots and tasks. A portion of the resources of a DCI is scheduled to one or more pilots in the form of containers supported by that DCI, and the tasks of a workload are dispatched to those pilots. This is a fundamental feature of Pilot systems because (i) a faster and more flexible execution of workloads is achieved by avoiding the overhead and lack of control imposed by a centralized job management system shared among multiple users; and (ii) the tasks of a workload can be bound to a set of pilots before of after it becomes available on a remote resource. ***mark: There is another (sub)level arguably, where the 'agent' makes scheduling decisions about task placement within the pilot's resource pool. Although this is an implementation detail, it might be worth mentioning?

The simplification obtained by bypassing the job submission system of the DCI is one of the main reasons for the success of the Pilot systems. As mentioned in §3.1, the tasks of a workload can be executed on a pilot without waiting in the queue system of the infrastructure increasing, as a result, the throughput of the workload execution. Moreover, pilots can be reused to execute multiple workloads until the pilots' walltime expire. It should be noted that how tasks are actually assigned to pilots is a matter of implementation. For example, a dedicated scheduler could be adopted, or tasks might be directly assigned to a pilot by the user.

The binding of tasks to pilots depends on the state of the pilot. A pilot is inactive until it is executed on a DCI, active after that. Early binding indicates the binding of a task to an inactive pilot; late binding the binding of a task to an active pilot. Early binding is useful to increase the flexibility with which pilots are deployed. By knowing in advance the properties of the tasks that are bound to a pilot, specific deployment decisions can be made for that pilot. ***mark: This can also be done without pilots. We probably need a better (if any) argument for early binding on pilots. Late binding is critical in assuring the aforementioned high throughput by allowing task execution without additional queuing time or container instantiation time.

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

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

Multi-level scheduling. Scheduling pilots onto resources and tasks onto active or inactive pilots.

Depending on the specific capabilities implemented in the Workload Manager component, some Pilot systems allow for pilots to be specified by taking into consideration the properties of (the tasks of) a workload. This type of specification should not be confused with early binding as the latter requires for a pilot to have been already bound to a resource.

Figure 4 offers a diagrammatic representation of how the minimal and consistent vocabulary proposed in this Subsection should be used to describe the logical components and core capabilities of a Pilot system illustrated in §3.1.

***matteo: For S3, the TODO list is: 1. Argue explicitly that the offered 'model' is sufficient and necessary in order to discriminate between Pilot and not-Pilot systems; [. . .], and possibly another one relative to overlay enacting as distinguished from task dispatching; 3. Clean up Table 1 removing 'Infrastructure'.

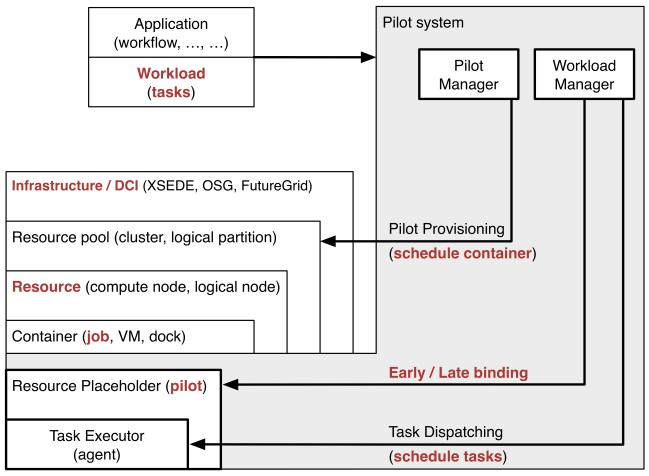


Figure 4: Diagrammatic representation of the logical components, capabilities, and core vocabulary of a Pilot system. The terms of the core vocabulary are highlighted in red.

4. PILOT SYSTEMS: ANALYSIS AND IMPLEMENTATIONS

Section §3 offered two main contributions: a minimal description of the logical components and the functionalities of Pilot systems [1], and a well-defined core terminology to support reasoning about such systems. The former sets the necessary and sufficient requirements for a distributed system to be a Pilot system, while the latter enables consistency when referring to different Pilot systems. Both these contributions are leveraged in this Section in order to review critically a set of relevant Pilot system implementations.

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

4.1 Core and Auxiliary Properties

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

As shown in Table **??**, the set of Core Properties is derived consistently with the set of functionalities presented in §**??** - Pilot Provisioning, Task Dispatching, and Task Execution. ***mark: I dont see how it is derived or consistent ;-) As such, Core Properties are both necessary and sufficient for an implementation of a distributed system to be classified as a Pilot system. On the contrary, auxiliary properties are not defining of a Pilot system but they are implementation details that further characterize a Pilot system. The set of auxillary properties we discuss is not closed, i.e., the complete set of auxillary properties includes, but is not limited to the set of auxillary properties we discuss. ***mark: todo: Fix/merge the two non-compatible definitions of aux props

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

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

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

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

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

	Property	Component	Description
Core	Pilot Resource Capabilities	Pilot Manager	The resources that the Pilot represents.
	Resource Interaction	Pilot Manager	The software and protocols of the interaction with the DCI.
	Overlay Management	Workload Manager	The construction of the Overlay.
	Workload Semantics	Workload Manager	The specification of the semantics (between tasks) captured in the workload description.
	Task Binding Characteristics	Workload Manager	The techniques and policies of binding tasks to Pilots.
	Task Execution Modes	Task Executor	The types of tasks and the mechanisms to execute tasks.
Auxiliary	Architecture	Pilot Manager Workload Manager Task Executor	Frameworks and archicture that the components and their whole are build with.
	Coordination and Communicatuon	Pilot Manager Workload Manager Task Executor	The interaction between the components of the system.
	Interface	Pilot Manager Workload Manager	Interface that the user can use to interact with the system.
	Interoperability	Pilot Manager Workload Manager Task Executor	Interopability between Pilots on multiple DCIs.
	Multitenancy	Pilot Manager Workload Manager Task Executor	The usage of components by multiple (simultaneous) users.
	Robustness	Pilot Manager Workload Manager Task Executor	The measures in place to increase the robustness of the components and the whole.
	Security	Pilot Manager Workload Manager Task Executor	AAA considerations for the components and the whole.
	Files and Data	Pilot Manager Workload Manager	The mechanisms that the system offers to explicitly deal with files and data.
	Performance and Scalability	Pilot Manager Workload Manager Task Executor	A description of scale and limitations and measures to reach that.
	Development Model	Pilot Manager Workload Manager Task Executor	The development- and support model for the software.

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

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

4.1.1 Core properties

Following is the list of core properties. This list of properties is minimal and complete. Note that these are the properties of Pilot implementations, and not of (individual) instantiations of Pilots.

- **Pilot Resource Capabilities**: In §**??** pilots have been defined as placeholders for resources. As such, the resource capabilities of each pilot depends upon the type and capabilities of the resource it exposes. Pilots usually expose computing resources but, depending on the capabilities offered by the infrastructure where the pilot is instantiated, they might also expose data and network capabilities. Within the domain of each type of resource and infrastructure, some of the typical characteristics of pilots are: Size (e.g. number of cores), lifespan, intercommunication (e.g. low-latency or inter-domain), computing platforms (e.g. x86, or GPU), file systems (e.g. local, shared, or distributed). Another consideration in discussing Pilot resource capabilities is the notion of granularity, e.g. is there one Pilot for a set of resources within on infrastructure or is there a more fine grained coupling between Pilot and resource and how does this effect the suitability for HTC and/or HPC? ***mark: This should not overlap with Task Execution Modes

- **Resource Interaction**: The provisioning of pilots depends on how the Pilot system interfaces with one or more targeted infrastructure(s). In this context, the degree of coupling between the Pilot system and the infrastructure can vary, depending on how the Pilot system is deployed, how much it is integrated with the middleware used within the targeted infrastructure, whether the Pilot system is interoperable across multiple types of middleware, and how much information the Pilot system needs about the states and capabilities of the targeted infrastructure. ***mark: This should not overlap with Task Execution Modes

- **Overlay Management**: The total set of resources fostered by Pilots together form the overlay onto which a workload can be dispatched. Whether this overlay is left to chance or careful crafting by the Pilot system is a major implication.

- **Workload Semantics**: The tasks of a workload are dispatched to pilots depending on their semantics. Specifically, dispatching decisions depends on the relationship held by a task with the other tasks belonging to the workload, the affinity they require between data and compute resources, and the type of capabilities they require in order to be executed. Pilot implementations support a varying degree of semantic richness for the workload and its tasks. Additionally of interest is the (standard) format or language in which the workloads are described.

- **Task Binding Characteristics**: One of the core functionalities implied by Task Dispatching is the binding of tasks to pilots. Without such capability, it would not be possible to know where to dispatch tasks, Pilots could not be used to execute tasks and, as such, the whole Pilot system would not be usable. As seen in §**??**, 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 behavior, they might also differ in whether they allow for the users to control directly what type of binding is performed, and in whether both types of binding are available on an heterogeneous pool of resources. Besides the binary decision between early and late binding, the Pilot system can expose more detailed application-level scheduling decisions, dispatch policies, etc., and might even include more levels of scheduling.

- **Task Execution Modes**: Once the tasks are dispatched to a pilot, their execution may require for a specific environment to be set up. Pilot system implementations differ in whether and how they offer such a capability. Pilot system implementations 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 implementation allows for data retrieval, dedicated software and library installations, communication and coordination among multiple execution environment and, in case, pilots.

4.1.2 Auxiliary properties

- **Architecture**: Pilot systems may be implemented by means of different type of architectures (e.g service-oriented, client-server, or peer- to-peer). For example, it is conceivable that architectural choices influence if not preclude certain deployment strategies. The analysis and comparison of architectural choices is limited to the trade-offs implied by such a choice, especially when considering how they affect the Core Properties.

- **Coordination and Communication**: Earlier in 3.1 we discussed extensively the importance of Coordination and Communication, but also suggested it as a non-defining function of a Pilot system. The details of Communication and Coordination between its components do distinguish various Pilot systems from each other.

- **Interface**: The implementations of Pilot systems may present several types of interfaces. For example, there are interface considerations between the main components of the Pilot system, between the application and Pilot system, between end users and the Pilot system, and for one or more programming languages. Some of this interface implications might be a consequence of the Architecture of the Pilot system. Here we will focus on the external interface.

- **Interoperability**: Two types of interoperability are relevant when analyzing different Pilot system implementations: Interoperability with multiple type of resources and interoperability across diverse Pilot systems. The former allows for the Pilot systems to provision pilots and execute workloads on different type of resources and systems (e.g. HTC, HPC, Cloud but also Condor, LSF, Slurm or Torque), while the latter becomes relevant when considering a landscape where multiple Pilot system implementations are available with diversified characteristics and capabilities. ***melissa: Note from meeting with SJ: May want to change to discuss one first (including descriptive text), then change tracks to discuss the second - they are two very different concepts to grasp in the first sentence (I know this detracts from the mathematical/proof language of it, but I think its a good idea). Especially since the second type of interoperability is very much less-common to a single PJ system (except in cases like PanDA)
- **Multitenancy**: A Pilot system may offer multitenancy at both system and local level. When offered at system level, multiple users are allow to utilize the same instance of a Pilot system, while when available at local level, multiple users may use the same pilot instance or any other component implemented within the Pilot system. ***mark: Alternative names: deployment, service model
- **Robustness**: Used to identify those properties that contribute towards the resilience and the reliability of a Pilot system implementation. In this Section, the analysis focuses on fault-tolerance, high-availability and state persistence. These properties are considered indicators of both the maturity of the development stage of the Pilot system implementation, and the type of support offered to the paradigmatic use cases introduced in Section 2.
- **Security**: While the properties of Pilot system implementations related to security would require a dedicated analysis, we limit the discussion to authentication, authorization and policies. The scope of the analysis is further constrained by focusing the analysis only on those elements of these properties that impact the Core Functionalities as defined in §**??**.
- **Files and Data:** Does the framework provide data management capabilities or should that be done out of band? If so, what does it offer?
- **Performance and scalability:** Performance and scalability define the response times and size of the workload a user can expect to be supported.
- **Development Model:** Is this a community effort?

4.2 Analysis of Pilot system Implementations

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

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

4.2.1 DIANE

DIANE [25] is a task coordination framework, which follows the Master/Worker pattern. It was developed at CERN for data analysis in the LHC experiment, but has since been used in various other domains, though mainly in the Life Sciences. As DIANE is a software framework, some of its semantics are not pre-determined, as they can be implemented in various ways as the user sees them. In that way it also provides Pilot functionality for job-style executions.

Resource Interaction.

The Pilot provisioning with GANGA [15] provides a unified interface for job submissions to various resource types. In this way, DCI specifics are hidden from other parts of DIANE as well as the from user.

Overlay Management.

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

Workload Semantics.

As expected from the fact that DIANE is in essence a Master/Worker framework, the **Tasks** have no relationships inside the framework, thereby the **Workload Semantics** are that of a Bag-of-Tasks. The **Workload** that is managed by the application specific part can maintain any type of workload structure, as long as it can be translated to individual tasks[**?**, **?**]. Plugins for other workloads, e. g. DAGs or for data-intensive application, exist or are under development. The framework is extensible: applications can implement a custom application-level scheduler.

Coordination and Communication.

The *Coordination & Communication* in DIANE is based on CORBA [27] using TCP/IP. The CORBA layer itself is invisible to the application. Networking-wise, the workers are clients of the master server. On TCP/IP level, communication is always uni-directionally from the `WorkerAgent` to the `RunMaster`. *Security* is provided by a secret token that the `WorkerAgent` needs to communicate back to the `RunMaster`. This implies that the network requirements are such that the `WorkerAgent` needs to be able to reach the `RunMaster` through TCP/IP but not the other way around. Bi-directional communication is achieved by periodic polling through heartbeats by the `WorkerAgent`, where the `RunMaster` responds with feedback.

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

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

Task Binding Characteristics.

DIANE is primarily designed with respect to HTC environments (such as EGI [13]), i. e. one Pilot consists of a single worker agent with the size of 1 core. Although the semantics of the binding are ultimately controllable by the user-programmable scheduler, the general architecture is that of a pull model. The pull model naturally implements the late-binding paradigm as the worker will only pull a new task once it is available and has free resources.

Task Execution Modes.

DIANE is primarily designed with respect to HTC environments (such as EGI [13]), i. e. one Pilot consists of a single worker agent with the typical size of 1 core/node, and thus, by default is not able to run (widely) parallel applications (e. g. based on MPI).

Pilot Resource Capabilities.

One of the DIANE *Pilot Resource Capabilities* is that it has the notition of "capacity" and applications can make use of that property do diverge from the "1 worker = 1 core" mode.

Architecture.

The central component of DIANE is the `RunMaster`. It consists of a `TaskScheduler` and an `ApplicationManager`. Both are abstract classes that need to be implemented for the specific purpose at hand. Example and/or default implementations are provided for the user to use or build upon. Together the `TaskScheduler` and `ApplicationManager` implement the **Workload Manager** component.

The `TaskScheduler` keeps track of the task entries and is responsible for **Binding Tasks** to the `ApplicationWorkers`. The implementation of the `ApplicationManager` is as its name suggest, responsible for defining the application **Workload** and the creating **Tasks** that are passed to the **TaskScheduler**. As the `ApplicationWorker` asks for **Tasks** from the `TaskScheduler` the natural way of implementing the scheduler is in a **Late Binding** approach.

The **Pilot** in DIANE is the `WorkerAgent`. The core component of the `WorkerAgent` is the `ApplicationWorker`, which is an abstract class that defines three methods that every implementation needs to implement. Two of these methods are for initialization and cleanup and the last, `do_work()`, is the method that actually receives the task description and executes the work, thereby being a close resemblance of the **Task Executor**.

Interface.

DIANE's *Architecture* is based on the `Inversion of Control` design pattern. It is a Python framework, that formulates certain hooks that an **Application** can be programmed against. The aforementioned abstract classes need to be implement to provide the application-specific semantics. The *Interface* to these "DIANE-**Applications** " is to start them through the `diane-run` command. As discussed in the *Overlay Management* section, the creation of Pilots is done out-of-band.

Performance and Scalability.

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

Robustness.

The *Robustness* in DIANE comes from the mature CORBA communication layer, and from custom task-failure policies in the `TaskScheduler`. Dealing with *Robustness* in DIANE is twofold. At one hand DIANE offers the mechanisms to supports fault tolerance: basic error detection and propagation mechanisms are in place. Further, an automatic re-execution of tasks is possible. On the other hand, DIANE also leaves room for application specific strategies.

Files and Data.

The CORBA communiation channel between the Master and the Worker also allows for file transfers, meaning that the Pilot not only exposes cores, but also the file system on the worker node.

Interoperability.

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

Security.

Access to various middleware security mechanisms (e. g. GSI, X509 authentication) is achieved through its integration with GANGA.

Multitenancy.

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

Development Model.

Used, maintained and developed by CERN with external contributions and users.

4.2.2 DIRAC

DIRAC (Distributed Infrastructure with Remote Agent Control) is a software product that grew out of the CERN LHCb project[?]. DIRAC implements a community-wide `Workload Management System` (WMS) to manage the computational payload of its community members. The complete DIRAC ecosystem consists of the DIRAC Pilot framework and the WMS.

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

Pilot Resource Capabilities.

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

Resource Interaction.

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

Overlay Management.

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

Workload Semantics.

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

Additionally, users can specify requirements like needed input data for their tasks and this will be taken into account when placing the task on a respource.

Task Binding Characteristics.

Pilots allow an effective implementation of the pull scheduling paradigm. Once they are safely running in the final computing resource, they contact central WMS servers for a late binding of the resource to the payload.

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

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

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

Task Execution Modes.

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

Coordination and Communication.

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

Architecture.

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

Interface.

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

Interoperability.

DIRAC is not inter-operable with other Pilot systems. Through the `Pilot Director` abstraction it does support execution of Pilots on a variety of computing resources.

Multitenancy.

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

Robustness.

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

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

Security.

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

Files and Data.

Tasks within DIRAC that specify their input data are threated in a special way. The WMS ensures that tasks are only started on resources where that data is available or it makes sure that that data becomes available.

Performance and Scalability.

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

Development Model.

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

4.2.3 Falkon

The Fast and Light-weight tasK executiON framework (Falkon) [31] was created with the primary objective of enabling many independent tasks to run on large computer clusters (an objective shared by most Pilot-Job systems). A particular focus on performance and time-to-completion for jobs on such clusters drove Falkon development. In addition to being *fast*, Falkon, as its name suggests, also focused on lightweight deployment schemes.

Pilot Resource Capabilities.

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

Resource Interaction.

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

Falkon has been shown to run on TeraGrid (now XSEDE), TeraPort, Amazon EC2, IBM Blue Gene/L, SiCortex, and Workspace Service [31].

Overlay Management.

The Dispatcher service in Falkon is implemented by means of a web service. This Dispatcher implements a factory/instance deployment scenario. When a new client sends task submission information, a new instance of a Dispatcher is created.

Workload Semantics.

Falkon has been integrated with the Karajan workflow language and execution engine, meaning that applications that utilize Karajan to describe their workflow will be able to be executed by Falkon.

The Swift parallel programming system [41] was integrated with Falkon for the purpose of task dispatch.

Task Binding Characteristics.

Cores and tasks are considered homogeneous. Tasks are pulled in to the nodes and are thereby of late-binding nature.

Task Execution Modes.

Falkon does not support MPI jobs, however - a limiting factor in its adoption to certain scientific applications.

Coordination and Communication.

Interaction between the components is by use of the Globus Web Services model.

Architecture.

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

The overall task submission mechanism can be considered a 2-tier architecture; the Dispatcher (using the above terminology, this is the Pilot-Manager) and the Executor (the Pilot-Agent).

The Dispatcher is a GRAM4 web service whose primary function is to take task submission as input and farm out these tasks to the executors. The Executor runs on each local resource and is responsible for the actual task execution. Falkon also utilizes *provisioning* capabilities with its Provisioner.

The Falkon Provisioner is the closest analogous entity to a Pilot: it is the creator and destroyer of Executors, and is capable of providing both static and dynamic resource acquisition and release.

Further, in order to process more complex workflows, Falkon has been integrated with the Karajan workflow execution engine [38].

This integration allows Falkon to accept more complex workflow-based scientific applications as input to its Pilot-like job execution mechanism.

Interface.

Simpler task execution can be achieved without modifying the existing executables - sufficient task description in the web service is all that is required to utilize the Falkon system.

Interoperability.

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

Multitenancy.

Each instantiation of the Dispatcher maintains its own task queue and state - in this way, Falkon can be considered a single-user deployment scheme, wherein the "user" in this case refers to an individual client request.

Robustness.

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

Security.

In its use of Globus transport channels it allowed for both encrypted and non-encrypted operation. The non-encrypted verion was used to gain most throughput, but obviously doing concessions on security for real-world use.

Files and Data.

The data management capabilities of Falkon also extend beyond the core Pilot-Job functionalities as described above.

Falkon has extended its capabilities to encompass advanced data-scheduling and caching.

Work has also been done on Falkon to expand its data capabilities. In addition to data caching and efficient data scheduling techniques, Falkon has adopted a data diffusion approach. Using this approach, resources for both compute and data are acquire dynamically and compute is scheduled as close as possible to the data it requires. If necessary, the data diffusion approach replicates data in response to changing demands [32].

Performance and Scalability.

As previously stated, the design of Falkon was centered around the goal of providing support to run large numbers of jobs efficiently on large clusters and grids. Falkon realizes this goal through the use of (i) a dispatcher to reduce the time to actually place tasks as jobs onto specific resources (such a feature was built to account for different issues amongst distributed cyberinfrastructure - such as multiple queues, different task priorities, allocations, etc), (ii) a provisioner which is responsible for resource management, and (iii) data caching in a remote environment [31].

Falkon has been tested for throughput and performance in such as applications as fMRI (medical imaging), Montage (astronomy workflows), and MolDyn (molecular dynamics simulation) and has shown favorable results in terms of overall execution time when compared to GRAM and GRAM/-Clustering methods [31].

The per task overhead of Falkon execution has been shown to be in the millisecond range.

Falkon has been demonstrated to achieve throughput in the range of hundreds to thousands of tasks per second for very fine-grained tasks.

Development Model.

The Falkon project ran from 2006 to 2011, source code is not available.

4.2.4 HTCondor

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

Architecture.

At the heart of it, HTCondor is a high-throughput distributed batch computing system. It accepts user tasks and executes them on one or more resources gathered in an HTCondor `pool`.

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

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

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

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

It is possible for two or more HTCondor pools to "collaborate" so that one pool has access to the resources of another pool and vice versa. In HTCondor, this concept is called `flocking` [14] and allows agents to query matchmakers outside their own pool for compatible resources. Flocking is used to implement load-balancing between multiple pools but also to provide a broader set of heterogeneous resources to user communities.

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

Resource Interaction.

In its native mode, HTCondor is the middleware, and therefore the discussion of resource interaction doesn't apply.

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

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

Overlay Management.

glideinWMS, a workload management system (WMS) [35] based on Condor GlideIns introduces advanced Pilot-Job capabilities to HTCondor by providing automated Pilot (`startd`) provisioning based on the state of an HTCondor pool.

Interface.

The main HTCondor distribution provided command line utils to setup these Glideins. These are now replaced by Bosco.

BoSCO is a user-space job submission system based on HTCondor. BoSCO was designed to allow individual users to utilize heterogeneous HPC and grid computing resources through a uniform interface. Supported backends include PBS, LSF and GridEngine clusters as well as other grid resource pools managed by HTCondor. BoSCO supports both, an agent-based (*glidein* / worker) and a native job execution mode through a single user-interface.

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

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

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

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

Multitenancy.

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

Development Model.

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

4.2.5 MyCluster

MyCluster [39] was developed to allow users to submit and manage jobs across heterogeneous NSF TeraGrid resources in a uniform, on-demand manner. TeraGrid, the predecessor of XSEDE, existed as a group of compute clusters connected by high-bandwidth links facilitating the movement of data, but with many different job deployment middlewares requiring cluster-specific submission scripts. MyCluster allowed all cluster resources to be aggregated into one personal cluster with a unified interface, being either SGE, OpenPBS or Condor. This enhancement to user control was envisioned as a means of allowing users to submit and manage thousands of jobs at once across heterogeneous distributed computing infrastructures, while providing a familiar and homogene interface for submission.

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

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

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

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

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

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

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

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

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

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

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

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

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

Interface.
The *Interface* of MyCluster depends on the choice of the underlying system as that one is exposed through the `Virtual Login Session`. The interface also allows users to interactively monitor the status of their **Pilots** and **tasks**.

Multitenancy.
The created cluster resides in userspace, and may be used to marshal multiple resources ranging from small local resource pools (e.g. departmental Condor or SGE systems) to large HPC installations. The clusters can be created per user, per experiment or to contribute to the resources of a local cluster.

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

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

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

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

4.2.6 PanDA
PanDA was developed to provide a multi-user workload

management system for ATLAS [?], which is a particle detector at the Large Hadron Collider at CERN which could handle large numbers of jobs for data-driven processing workloads. In addition to the logistics of handling large-scale job submission, there was a need for integrated monitoring for analysis of system state and a high degree of automation to reduce the need for user/administrative intervention. As PanDA is a multi-stakeholder project that is active for many years, it naturally went through many iterations. We try to refer to the current state as much as possible.

The core component of PanDA responsible for the Pilot aspects is AutoPyFactory[?]. It is handles Pilot submission, management and monitoring system. (AutoPyFactory supersedes the initial generation PandaJobScheduler, as well we the second generation system, the AutoPilot.)

Design Goals.

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

Applications.

PanDA has been used to process data and jobs relating to ATLAS. Approximately a million jobs a day are managed [12] which handle simulation, analysis, and other work [24]. The ATLAS experiment itself produces several petabytes of data a year which must be processed and analyzed. Current funding for PanDA enables the project to reach out to new user communities.

Workload Semantics.

job type, priority, input data/locality,

Deployment Scenarios (VO/multiuser).

PanDA has been initially deployed as an HTC-oriented multi-user WMS system for ATLAS, consisting of 100 heterogeneous computing sites [24]. Recent improvements to PanDA have been designed to extend the range of deployment scenarios to non-ATLAS infrastructures making PanDA a general-use Pilot-Job [26]. When the pilot is launched, it collects information about the worker node and sends it to the job dispatcher. If a matching job does not exist, the pilot will end. If a job does exist, the pilot will fork a separate thread for the job and start monitoring its execution. The DDM(?) handles data transfer, must have database access, file stage-in/out and takes care of cleanup.

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

PanDA began as a specialized Pilot-Job for the ATLAS grid, and has been extended into a generalized Pilot-Job which is capable of working across other grids as well as HPC and cloud resources. Cloud capabilities have been used as part of Helix Nebula (CloudSigma, T-Systems, ATOS), the FutureGrid and Synnefo clouds, and the commercial Google and EC2 cloud offerings as well [12], extending the reach of PanDA to cloud resources as well. Future PanDA developments seek to interoperate with HPC resources [12].

Interoperability.

PanDA traditionally relies on Condor (and SGE?) for its interaction with DCIs. For recent endeavours in HPC they are using SAGA to interface with the queuing systems.

Architecture & Interface.

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

As PanDA is an *Advanced Pilot-Job System*, it enables functionality beyond that of a *Basic Pilot-Job*. PanDA contains some additional backend features such as `AutoPilot`, which tracks site statuses via a database, `Bamboo` which adds ATLAS database interfacing, and automatic error handling and recovery. PanDA contains support for queues in clouds, including EC2, Helix Nebula, and FutureGrid [12]. Enhancements to the userspace include `Monitor`, for web-based monitoring, and the `PanDA client`. `PanDA Dynamic Data Placement` [24] allows for additional, automatic data management by replicating popular or backlogged input data to underutilized resources for later computations and enables jobs to be placed where the data already exists.

Robustness.

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

Security.

PanDA on HTC grids mainly depends on GSI Certificate-based security and glExec to run as given user.

4.2.7 RADICAL-Pilot

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

Pilot Resource Capabilities.

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

Resource Interaction.

RADICAL-Pilot uses the Simple API for Grid Applications (SAGA) [17, 18] in order to interface with different DCIs. Through means of SAGA, RADICAL-Pilot submits the Pilot Agent as a **job** through the reservation or queuing system of the DCI. Once the `Agent` is started there is no direct *resource interaction* with the DCI anymore except for monitoring of the state of the `Agent` process. Using SAGA has enabled RADICAL-Pilot to expand to many of the changing and evolving architectures and middlewares.

Overlay Management.

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

Workload Semantics.

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

Task Binding Characteristics.

Task to **resource binding** can be done either implicitly or explicitly. A user can bind a task explicitly to a pilot, and thereby to the resource the pilot is scheduled to, making this a form of early binding. The user can also submit a task to a unit manager that schedules units to multiple pilots. In this case it is the semantics of the scheduler that decides on the task binding. RADICAL-Pilot supports multiple schedulers that can be selected at runtime. There currently exist two schedulers: 1) a round-robin scheduler that binds incoming tasks to associated pilots in a rotating fashion, irrespecive of their state, and thereby performing **early binding**. 2) a BackFilling scheduler that binds tasks to pilots once the pilot is active and has available resources, thereby making it a **late binding** scheduler.

Task Execution Modes.

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

Architecture.

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

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

Coordination and Communication.

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

Interface.

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

Interoperability.

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

Multitenancy.

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

Robustness.

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

Security.

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

Files and Data.

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

Performance and Scalability.

RADICAL-Pilot is one of the largest pilot based consumers of resources on XSEDE[]. Per Pilot Agent many thousands of concurrent ComputeUnits can be managed. As multiple pilots can easily be aggregated over multiple distinct DCIs, RADICAL-Pilot can achieve scalability in many dimensions.

Development Model.

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

4.3 Overall observations

***mark: Maybe this subsection should go and address it in the discussion and conclusion

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

5. DISCUSSION AND CONCLUSION

***matteo: In Section 3 we use pilotjob. We will have to move to Pilot or decide to use the less correct pilotjob here too. ***matteo: Following discussion, we decided to use Pilot. Section 3 has been updated, the other Sections will have to follow.

Sections **??** and 4 offered respectively a description of the minimal capabilities and properties of a Pilot system; a vocabulary defining 'pilot' and its cognate concepts; a classification of the core and auxiliary properties of a paradigmatic pilot system implementation; and the analysis of an exemplar set of pilot system implementations. Considered altogether, these contributions show how the notion of Pilot system indicates a paradigm for the execution of tasks on distributed resources by means of resource placeholders.

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

5.1 The Pilot Paradigm

The Pilot paradigm identifies a type of software system with both general and unique characteristics. This paradigm is general because, in principle, it does not depend on a single type of workload, a specific infrastructure, or a unique performance metric. Systems implementing the Pilot paradigm can execute workloads composed by any number of tasks with disparate requirements. For example, as seen in §4, different 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 the type of resource on which a Pilot system can execute given workloads. As seen in §**??**, the Pilot paradigm demands resource placeholders but does not specify the type of resource container that should be exposed by the target resource. HPC, Grid, or Cloud resources all expose type of containers that are currently leveraged by Pilot systems to create resource placeholders.

Finally, while usually Pilot systems are designed to maximize the throughput of task execution, the Pilot paradigm is independent from any performance metric. Depending on the given workload and the target resource, a Pilot system implementing the Pilot paradigm can be design to maximize or minimize a specific element of task execution.

The Pilot paradigm is unique because of three distinctive characteristics: resource placeholding, multi-level scheduling, and early or late binding. A Pilot system does not offer just an alternative implementation of a scheduler, of a workload manager, of a batch system, or of any other special-purpose software component. A Pilot system implements a specific patterns of workload execution based on multi-level scheduling and distributed task execution. Every software system with these three characteristics, built upon the logical components and the functionalities described in §3.1 is an implementation of a Pilot system as seen in Section 4.

***shantenu: Need thinking: Following on from discussion yesterday, is early/late binding a logical consequence of multi-level scheduling? Can we absorb the latter into the former? ***matteo: Following the revision of S3, early/late binding seems to be related to having resource placeholder while multi-level scheduling seems to be the means to have place-holders. Both early and late binding refers to the same entities: tasks bound to pilots, i.e. place holders. Multi-level scheduling refers to two pairs of entities: container (i.e. jobs) scheduled to DCIs, and tasks scheduled to placeholders. This opens the issue of whether binding is a form of scheduling, whether binding and scheduling are the same. We may want to discuss about it.

***matteo: The following paragraphs address the following: PJs have well defined semantics and model. PJs dont help with data intensive applications.

The scope of the Pilot paradigm encompasses multiple types of resource. Usually, the tasks executed on pilots leverage mainly the cores of the compute nodes for which the pilot is a placeholder. Tasks may also use data or network resources but mostly as accessories to their execution on computational resources. Local or shared filesystems may be used to read input and write output data while network connectivity may be required to download, for example, the code of the task executor. In this context, it is important to notice that the focus on computational resources is not mandated by the Pilot paradigm. In principle, pilots can be placeholders also and exclusively for data or network resources depending on the capabilities exposed by the middleware of the target DCIs. For example, in Ref. [pilot data], the notion of Pilot-Data has been conceived using the power of symmetry, i.e., Pilot-Data taken to be a notion as fundamental to dynamic data placement and scheduling as Pilot is to computational tasks.

The generality of the Pilot paradigm has been modeled in Ref. [23]. This investigation was motivated by the desire to provide a single conceptual framework – the so called $P*$ Model – that would be used to subsume the design characteristics of the various Pilot system implementations, a sample of which have been analyzed in detail in Section 4. ***matteo: Is this still the case? I think this has become one of the goals of this paper. How do you want to mention P* here? What do you want to say about it? Is the paragraph above sufficient? Interestingly, the P* model was amenable and easily extensible to Pilot-Data. The consistent and symmetrical treatment of data and compute in the model led to the generalization of the model as the *$P*$ Model of Pilot Abstractions*.

***matteo: The following paragraphs address the following: (i) PJs are just for HTC and they just circumvent job queuing delays; PJs unfairly game HPC queuing system; (ii) PJs have to be tied to specific infrastructure and infrastructures have to be tied to specific pjs; (iii) PJs are such a simple concept, it doesn't need more attention: Conversely, everyone should write their own PJ just because they can; Note: the second part of this statement is better addressed when discussing the fragmentation of the pilot landscape. (iv) PJs are stand-alone tools passive (system) tools, as opposed to user-space, active and extensible components of a CI;

The generality of the Pilot paradigm may come as a surprise when considering the most common requirement that motivates its implementations. Leveraging multi-level scheduling so to increase the execution throughput of large workloads made of short running tasks has been achieved without devising a new paradigm. For example, as seen in §4 PanDA, Falkon, or DIRAC were initially developed as single-point solutions, focusing on either a type of workload, a specific infrastructure, or the optimization of a single performance metric. Appreciating the properties of the Pilot paradigm becomes necessary once requirements of infrastructure interoperability, support from multiple types of workloads, or flexibility in the optimization of execution are introduced. Satisfying those requirements requires abstracting the specificity of middlewares, infrastructure architectures, and application patterns. As shown in this paper, this process of abstraction means to develop an understanding of the Pilot paradigm.

Appreciating the Pilot paradigm means also to understand that its implementations are not circumventing the infrastructure middleware but simply abstracting away some of its properties in order to optimize one or more user-defined performance metric. Multi-level scheduling is not replacing the infrastructure-level scheduler. As clearly illustrated in §**??**, resource containers are still created on the target infrastructure by means of that infrastructure capabilities and the second level of scheduling is completely contained within the boundaries of the resource container. Optimizing the usage of the resource containers, for example by minimizing idling time, is an implementation issue, not an intrinsic limitation of the Pilot paradigm. Furthermore, it should be noted that shifting the control over tasks scheduling away from the infrastructure middleware does not imply that the end-user will become necessary responsible for an efficient resource utilization. A Pilot system classify as a middleware component and, as seen with HTCondor, it can be implemented as an integral part of the infrastructure middlware stack.

5.2 Pilot Paradigm in Context

HTC vs HPC. We have identified that the Pilot Paradigm in itself is very common and on an abstract level applies to all Pilot systems we have discussed. This is also illustrated by the mapping to the P* model. While arguably not a fundamental difference in the application of Pilots, there certainly are structural practical differences in using Pilots on HPC vs HTC. While in the *temporal* dimension the ratio of **Tasks** and Pilots might be similar for both type of infrastructures (i.e. Pilots run for 12 hours and tasks for 1 hour), the *spatial* dimension is a different story. Let's define HPC as O(10) sites with a tightly coupled system. These systems typically have a head/login/service node where the Pilot will operate from. For one site, the ratio of Pilots vs compute resources will typically be O(1):O(Nodes) and therefore the ratio of Pilots vs **Tasks** will be O(1):O(Tasks). With the given O(10) sites this leaves us with a Pilot:**Tasks** ratio of at most O(10):O(Tasks) On the other hand, if we define HTC as O(100) sites of lousely coupled systems, without a central node to foster these resources, the ratio of Pilots to compute resources will typically be closer to O(1):O(Cores). These differences have not only implications on the type of applications that can be run, but also on the nature of the multi-level scheduling. In the HPC case there is a large degree of freedom for the Pilot layer scheduling while in HTC the scheduling on that level is much more pre-determined.

MPI, OpenMP, etc.. In the P* model and in the discussion so far we have mainly been concerned about the abstract notion of **Task**, i.e. the application payload to be run on the DCI. While this abstraction is very natural it does pose challenges in practice when we are operating in a real-life heterogeneous environment, i.e. what does "run" mean .... And if we are able to define what it means, how do we pass that information down the stack?

Generality or optimisation. The existence of many Pilot systems is tempting to frown at from a perspective of Not-Invented-Here-Syndrome. Workload management. [this has a clear link to the pilot and application layer paragraph]

Pilot and Middleware Layer. The definition of the term 'pilot' and the critical review of its implementations highlight the relation between Pilot systems and middleware as commonly defined in [cit, cit, cit]. The notion of pilot as a resource placeholder is something more than a job as defined in the context of Grid- based middleware. For example, cloud-based DCIs ***matteo: Use of DCIs for plural needs to be consistent across the paper. introduce notable exceptions and differences in the way in which pilots can be provisioned.[cit, cit] Within a IaaS [cit], Virtual Machines (VMs) and not jobs are used for their provisioning. VMs can often be instantiated without waiting into a queue with a predictable instantiation time, usually minimal when compared to the queuing time often experinced on Grid-based DCIs [ref Vishal paper]. Furthermore, the limitations on the execution time of a VM can be virtually absent, a limitation that on grid-based middleware is imposed by the queue walltime. Clearly, overheads are introduced by having to deal with VMs and not simple jobs and the model adopted within a IaaS-based DCI to assign resources to each VM can affect the flexibility of the whole Pilot-Job system. [cit] A similar assessment could be done for a DCI deploying a PaaS model of cloud computing.[cit]

Pilot and Application Layer. The current state of workflow (WF) systems [2, 36] provides a motivating example for the P* Model and the Pilot-API: even though many WF systems exist (with significant duplicated effort), they provide limited means for extensibility and interoperability. We are not naive enough to suggest a single reason, but assert that one important contributing fact is the lack of the right interface abstractions upon which to construct workflow systems; had those been available, many/most WF engines would have likely utilized them (or parts thereof), instead of proprietary solutions. Significant effort has been invested towards WF interoperability at different levels – if nothing else, providing post-facto justification of its importance. The impact of missing interface abstractions on the WF world can be seen through the consequences of their absence: WF interoperability remains difficult if not infeasible. The Pilot-API in conjunction with the P* Model aims to prevent similar situation for Pilot-Jobs. ***matteo: To be abstracted from workflow to application.

Pilot and AAA. Because the Pilot can run as a different "user" than the "owner" of the **Task** there are challenges for many aspects of Authentication, Authorization and Accounting (AAA). The issue mainly arrises because of the fact that multi-level scheduling decouples the **DCI** user from the **Task** owner. While technically one could make the choice not to be concerned about this decoupling and only be concerned with the DCI credential of the Pilot, in practice this does not hold true. [ref glexec].

Pilot and the Enterprise. Over the past years, Hadoop [3] emerged as distributed computing for data-intensive tasks in the enterprise space. While early versions of Hadoop were very monolithic tightly coupling the Map Reduce programming framework with the underlying infrastructure resource management. With Hadoop 2, YARN [?] was introduced as a central resource manager supporting heterogenous workloads (and not only Map Reduce). YARN provides support multi-level scheduling enabling the application to deploy their own application-level scheduling routines on top of Hadoop-managed storage and compute resources. While YARN manages the lower resources, the higher-level runtimes typically use an application-level scheduler to optimize resource usage for the application. Applications need to initialize their so-called Application-Master via YARN; the Application Master is then responsible for allocating resources in form of so-called containers for the applications. It then can execute tasks in these containers.

As in the HPC environment, the support for different application workloads and job types: long-lived vs. short-lived applications, homogeneous vs. heterogeneous tasks is a challenge. Pilots address many of these requirements in HPC and HTC environments. Similar frameworks are emerging for YARN: Llama [?] offers a long-running application master for YARN application designed for the Impala SQL engine. TEZ [?] 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-/re-allocate resources. REEF [?] is a similar runtime environment that provides applications a higher-level abstractions to YARN resources allowing it to retain memory and cores supporting heterogeneous workloads.

5.3 Future Directions and Challenges

Sociotechnical. ***shantenu: What is the future of PJ? ; ***shantenu: Pilot-Jobs have potential, but it is not being realized due to ad hoc nature of theory and practise, ; Fragmentation/balkanization of the pilot landscape. While in this paper we also talk about HTC and HPC and their differences and commonalities, we have refered to an existing and growing demand for convergence between the two from an application perspective. [ref MTC?] HPC users are no longer exclusively interested in running one large job, but applications become more versatile and dynamic and are conflicting with the conventional thougths and operational realities on how to "use" HPC resources. [ref cray, cram, etc.] If the HPC world is going to accept this mode of operation, then recognizing, embracing and investing in the Pilot Paradigm is inevatable.

Engineering. ***shantenu: Why should we do to enhance the usability? ; ***ashley: I would argue that our work is important to Pilot-Jobs because by expressing a common model, we enable researchers to 1) understand the commonalities between existing Pilot-Job approaches in order to 2) motivate innovation + construction of "next-gen" Pilot-Job systems. E.G., implement the basics (or work from an existing system) + understand where the boundaries/unexplored territory is without having to first completely understand all 15+ existing Pilot-Job systems and their unique vocabulary.

While we are in no way to argue for a one-size-fits-all Pilot we do believe that our work has shown that there is enough common ground that establishing a new recognized layer in the stack would not only help conceptually, but would also pay-off from a practical engineering standpoint. There are many intricate details for example in the runtime environment, that sharing efforts on that layer would allow the various systems to focus their attention where there is less overlap and more tailored to their own communities. [ref Open RunTime Environment] Standardisation?

Contributions. ***shantenu: (i) we provide first comprehensive historical and technical analysis, (ii) set the stage for a common conceptual model and implementation framework, and (iii) provide insight and lessons for other tools and higher-level frameworks, such as Workflow systems possibly

Acknowledgements

This work is funded by the Department of Energy Award (ASCR) DE-FG02-12ER26115 and NSF CAREER ACI-1253644. This work has also been made possible thanks to computer resources provided by TeraGrid TRAC award TG-MCB090174 and BiG Grid. This document was developed with support from the US NSF under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed".

6. REFERENCES

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

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

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

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

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

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

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

[8] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *International Conference on High-Performance Computing in the Asia-Pacific Region*, 1:283–289, 2000.

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

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

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

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

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

[14] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: load sharing among workstation clusters. *Future Generation Computer Systems*, 12(1), May 1996.

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

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

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

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

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

[20] J. Kim, W. Huang, S. Maddineni, F. Aboul-Ela, and S. Jha. Exploring the RNA folding energy landscape using scalable distributed cyberinfrastructure. In *Emerging Computational Methods in the Life Sciences, Proceedings of HPDC*, pages 477–488, 2010.

[21] S.-H. Ko, N. Kim, J. Kim, A. Thota, and S. Jha. Efficient runtime environment for coupled multi-physics simulations: Dynamic resource allocation and load-balancing. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 349–358, Washington, DC, USA, 2010. IEEE Computer Society.

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

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

Computer Society.

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

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

[26] P. Nilsson, J. C. Bejar, G. Compostella, C. Contreras, K. De, T. Dos Santos, T. Maeno, M. Potekhin, and T. Wenaus. Recent improvements in the atlas panda pilot. In *Journal of Physics: Conference Series*, volume 396, page 032080. IOP Publishing, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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