NIMBIX

WHITE PAPER

HPC on Kubernetes

A practical and comprehensive approach

Leo Reiter
CTO
Nimbix, Inc.

Introduction

In 2012 Nimbix began running HPC applications using Linux containers and in 2013, launched the world's first container-native supercomputing cloud platform called JARVICE™. This platform was and is novel in various ways. First, it ran applications directly on bare-metal rather than virtual machines, utilizing Linux containers for security and multi-tenant isolation, as well as workload mobility. Second, it provided both a ready-to-run service catalog of commercial and open source workflows, in a Software-as-a-Service style, as well as a Platform-as-a-Service interface for developers and ISVs to onboard their own custom applications and workflows. At the time and largely to this day, most high performance cloud platforms leverage hypervisor virtualization and provide mainly Infrastructure-as-a-Service interfaces – mechanisms appropriate for Information Technology professionals but not scientists and engineers looking to consume HPC directly. The Nimbix Cloud, powered by JARVICE, overcame both the performance penalties of virtualized processing, as well as the ease of use challenges of IT-focused interfaces as such IaaS. The JARVICE software has since been released as an enterprise platform (called JARVICE XE) for use on-premises or on 3rd party infrastructure but retains all the usability and performance benefits (when run on bare-metal and with computational accelerators and low latency interconnects) as the Nimbix Cloud itself.

At the time Nimbix began deploying workflows in containers, there was neither a standard stable enterprise-grade format for packaging applications nor an available orchestration mechanism for deploying said containers on machines. Fast-forwarding to the present, we now take for granted both Docker and technologies such as Kubernetes. But before this, Nimbix had to invent the mechanisms and the "art", in order to bring products to market. The Nimbix Cloud and JARVICE XE have since run millions of containerized workloads in a myriad of different forms, solving real-world HPC problems in just about every industry and vertical. In 2019 Nimbix released HyperHub™ as the marketplace for accelerated and containerized applications, delivered as turn-key workflows to the JARVICE platform regardless of what infrastructure powers it. Not only can scientists and engineers consume containerized HPC seamlessly thanks to JARVICE, but ISVs supporting these users can monetize and securely distribute their codes without having to reinvent the wheel to do so.

In a somewhat related context, various container web service platforms have begun to emerge over the past few years, most notably Kubernetes. Google released the open-source Kubernetes platform to the world in 2014 as an evolution of tools it had used internally to scale web-based applications such as Gmail. From top to bottom Kubernetes is designed to scale web services based on (mainly) Docker-style containers behind load balancers and web proxies known as "Ingress" controllers. The architecture is ideal for delivering stateless request/response-type of services (e.g. RESTful APIs and other web applications). It also really simplifies development of said applications by automating the deployment patterns along standardized practices.

Just as JARVICE is not designed to serve out stateless web applications, Kubernetes is not designed to run HPC and other tightly coupled workflows efficiently. Both platforms utilize containers for the runtime of applications, but the workloads and compute methods they support are drastically different. In a world of standardization to improve operational efficiency however, it's not ideal to maintain different types of platforms for different types of applications. IT Organizations increasingly look to consolidate using a "layer cake" approach – e.g. the "infrastructure layer" should be able to run any type of application, in order to reduce the need for expensive specialized practices and additional labor.

Higher-level "layers" are there for the specifics, but rely on the underlying layers to cover the basics. Single-purpose platforms are generally phased out in favor of general-purpose ones – look no further than the rise of Unix and Linux systems since the 1970s and how they increasingly displaced the mainframe. For all but the most sensitive government and research type of deployments, so too should general-purpose platforms begin to displace traditional HPC. The missing link is the unifying technology to enable both the commodity Enterprise applications with more specific scientific and engineering ones on a common infrastructure management layer. As this paper will demonstrate, JARVICE XE combined with Kubernetes provides a practical approach to achieve just this.

Table of Contents

Linux Container Basics

A Linux container, most commonly formatted as a Docker container, can be thought of as just a runtime context for application bits. Unlike a virtual machine, a container shares the host operating system kernel with other containers. This makes containers a much lighter mechanism to run applications since they do not need to package an entire operating system stack with kernel and drivers. Instead, the focus is on the encapsulated application and its dependencies needed to run (shared libraries, configuration files, etc.). Containers can be thought of as a type of application virtualization, where traditionally virtualization in a hypervisor context has been machine level (hence the term "virtual machine").

A container at rest provides the files an application needs in order to run (binaries, scripts, configuration files, and libraries), together with some metadata to describe how to assemble the container runtime environment itself. In Docker terms, containers are packaged as stacked layers that can be cached individually on target systems, but it's up to the runtime environment to assemble the filesystem that containerized applications operate from.

At runtime, a container provides 3 basic mechanisms:

1. The filesystem – typically presented as a fully assembled "jail" that a containerized application cannot "escape" from; this also means that everything the application needs must exist within this filesystem, since it can't easily access the underlying host to leverage existing files and directories (unless the runtime environment is explicitly told to "bind" files and/or directories from the host –a practice that should be executed carefully due to the inherent security concerns around it).

2. The "namespaces" – in addition to the filesystem "jail", it's important that containers cannot "see" processes, network interfaces, and other objects outside their own context. Using namespaces allows applications to run in isolation both from each other as well as from the host system. A secure containerized platform such as JARVICE or Kubernetes will allow running multiple containers per host, without those containerized applications even being aware of each other nor of those that may be running on the underlying host operating system directly.

3. Access controls and resource limits - in Linux the primary mechanism for achieving this is known as "cgroups". One of the major benefits of containers versus virtual machines is the ease with which system devices can be "passed through" from the host to the container. Unlike with VMs there is no need for complex paravirtualization nor bus-level emulation methods since the applications share the same host kernel. It's, therefore, possible to connect devices such as computational accelerators (FPGAs, GPUs, etc.) to containerized applications easily and without overhead. But this must also be governed with extreme care as it can lead to a containerized application accessing any resource on the host system without restriction. Cgroups allow container platforms to restrict access to devices as well as general system resources such as memory and CPU cycles with very fine-grained control and minimum overhead.

The Container Ecosystem, Explained

Container Registry
A remotely accessible object store for container images (or containers at rest); generally fronted with a RESTful API for easy access from clients. Popular container registries include Docker Hub and gcr.io. Additionally, the Nimbix HyperHub provides workflow-level metadata and authorization to various container registries as a unified interface for synchronizing applications across clusters automatically.

Container Format
The defacto standard for containerized application packaging is Docker, but other formats exist as well (e.g. Singularity, but see below). Formatted images are “pushed” to the Container Registry and “pulled” by Container Runtimes.

Container Runtime
Once again Docker is the most popular container runtime, which actually pulls and runs containerized applications. But the Open Container Initiative (OCI) allows other engines, such as "*containerd*", to run Docker-style containers without modification. What container engine runs on a platform is less relevant than whether or not it can support Docker-style containers, given the popularity and vast adoption of this format.

Container Platform
If containers are the application in a container-native context, then the container platform is the operating system on which things run. The container platform provides interfaces for end-users and APIs to run and manage containerized applications. JARVICE in the Nimbix Cloud is the defacto container-native platform for HPC, while Kubernetes is increasingly becoming the defacto container native platform for web service applications. JARVICE XE actually interfaces with Kubernetes to run HPC applications on converged IT infrastructure.

An Alternative Approach to Linux Containers: Singularity

While Docker-style containers are increasingly ubiquitous and general-purpose, another format has made some inroads in HPC: Singularity. The main difference is in architectural philosophy. The Docker format (and Docker runtime) is intended to work in full isolation, providing its own network and system contexts to each container. It also allows a containerized application to gain administrative access (also known as “root”) within its “jail” and namespaces (see above). The container platform must ensure that the applications are properly resource managed to avoid security issues, but this is widely understood. In the Singularity school of thought, containers actually use the host for networking and interconnects, and do not allow containerized applications to gain root privileges even in their isolated runtime contexts. The design philosophy is to better support workload portability within the context of monolithic traditional HPC environments where everything is already installed on the host, such as MPI libraries (a key HPC component), etc. In the Docker philosophy, the host provides nothing other than the runtime environment setup itself to the containerized applications. Therefore Docker-style containerized applications must bring all of their own dependencies. While this results in slightly “fatter” images, it does simplify running diverse codes on the same systems at the same time and is much better suited to multi-tenant environments such as JARVICE.

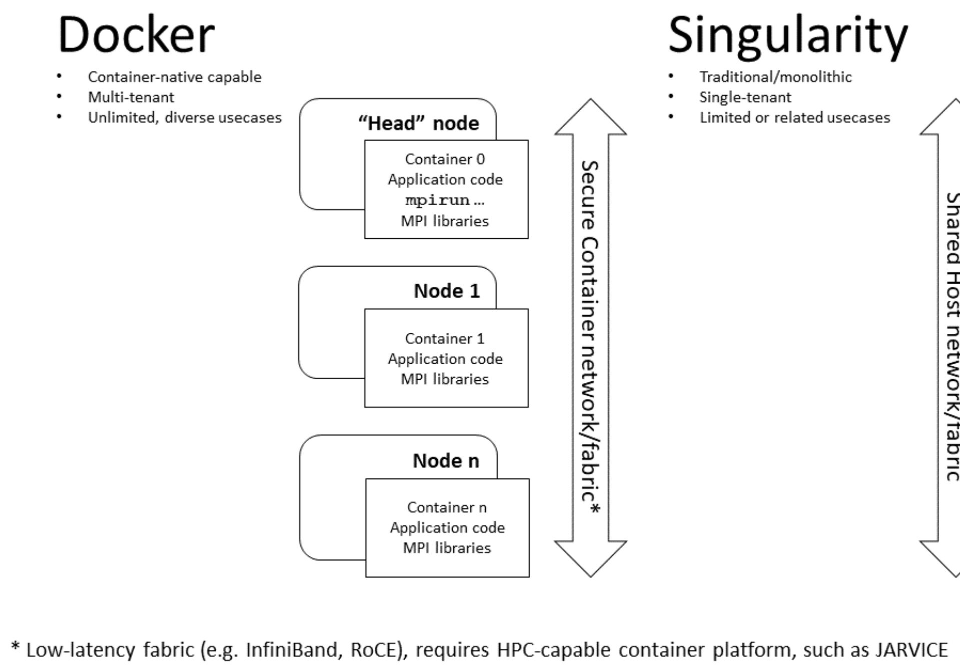


Figure 1: Containerized HPC Application Comparison

While Singularity is making inroads in traditional HPC, it is unlikely to challenge Docker-style containers for general purpose applications, and will therefore likely not be relevant in unifying HPC and commodity applications on converged IT infrastructure.

Container Native versus Containerized Applications with Docker

Ideally, applications running in containers are container-native. This generally means several things, but usually boils down to:

1. Minimized dependencies – only those libraries and configuration files needed for the specific purpose of the container are packaged; in fact, increasingly statically linked binaries such as those produced from the Go language are replacing application stacks, further simplifying and making container images leaner than ever.
2. Simplified operations – since containers are intended to be run on container platforms, it's not necessary to package large frameworks such as scalable HTTP(S) servers, firewall software, etc. The container platforms usually provide this functionality and simply proxy requests to containerized applications via standardized service ports. Scale-out is also handled by the container platforms, to further reduce complexity. In a Kubernetes environment, applications must follow simple rules for service discovery and the platform takes care of the rest. In a JARVICE environment, HPC applications are automatically configured in a runtime environment conducive to seamless MPI and other parallel distribution methods across nodes.

While container-native applications are of course a type of containerized application the reverse is not true. For example, it is possible (using well-understood methods) to containerize traditional applications for distribution and mobility, be they open source or commercial bits. Over the years Nimbix developed a methodology to do just this, given the breadth and complexity of existing HPC applications and the need to containerize them without modification:

1. A container image is built using the application's installer. In Docker terms, this means running installation scripts in "silent" mode (without the user interface, since there is no opportunity for user input when building container images), in the Docker file itself. Years of research and evolution led to highly optimized layers and support for even the most complex and extensive application suites. Again, the ISV codes are not container native, cannot be broken up into microservices, and are simply not aware of any sort of container runtime.
2. Additional layers are added to extend the container functionality, such as the graphical user interface (e.g. web-based desktop or shell), 3rd party integrations (e.g. for "cosimilation" across different vendor codes), and convenience (e.g. desktop applications that users typically run in conjunction with their HPC codes, such as text editors, etc). Nimbix has open sourced the graphical desktop environment for containers in a GitHub package called "image-common".
3. Workflow scripts, to automate running applications and plumbing license server connectivity, configuring parallel scale parameters automatically, etc. These scripts assume a JARVICE platform underneath but can easily be emulated locally for unit testing application workflows at low scale.
4. Metadata for workflow automation and "trade packaging" – simple declarative files are layered in, which help the JARVICE platform generate user interfaces and present high-level workflows to engineers and scientists. This helps "construct" commands and parameters to execute inside the container image at runtime. The "trade packaging" includes descriptive information, screen shots, and optionally EULA language, and is used in

generating the service catalog for HyperHub. This metadata is isolated and does not interfere with non-JARVICE platforms in any way – maintainers can continue to run their codes on generic Docker runtimes without JARVICE, but also without the end-user benefits of workflow automation.

In effect, all that is needed to containerize traditional applications is to automate an installation script, perform necessary post-install fixups, and develop wrapper mechanisms (e.g. workflow scripts) to parameterize and dynamically edit configurations to adapt to dynamic environments at runtime. A major challenge is the ephemeral nature of containers – unlike on a workstation or server there is no persistence unless files are stored in explicit volumes – so in some cases, considerable fixups are needed.

With the better part of a decade of experience doing this, Nimbix has both performed this feat on many popular application stacks and advised countless developers and ISVs on best practices for containerizing their own codes in a self-service fashion. The JARVICE platform even provides a CI/CD pipeline mechanism known as PushToCompute™, which can further help to enable and deploy traditional applications on heterogeneous platforms and architectures.

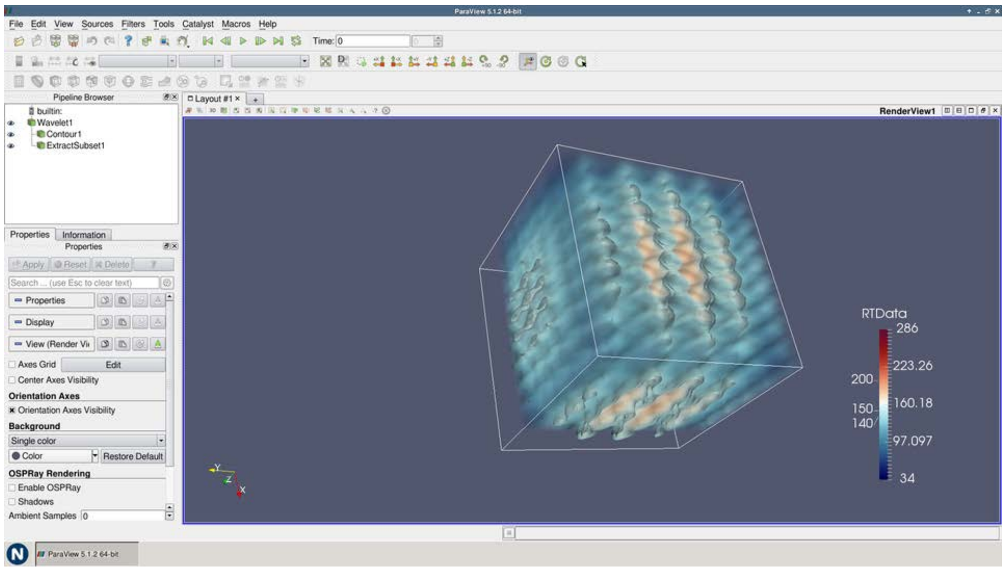


Figure 2: Sample containerized (not container-native) application running on JARVICE platform

Examples and Reference Material
Nimbix provides various examples of containerized traditional and container-native workflows on GitHub. These sources can be used as patterns to produce Dockerfiles and metadata for various types of HPC applications.

Additionally, the JARVICE Developer Documentation includes complete reference. Note that while all of this assumes a container runtime environment that sets up and mimics a dynamic HPC cluster, it is not strictly mandatory to use the JARVICE platform itself.

HPC on Kubernetes

The following section will examine the options for running HPC workflows on Kubernetes.

JARVICE vs Kubernetes Application Pattern Support
For comparison, the following table illustrates the level of support for various common application patterns between Kubernetes and the original JARVICE platform powering the Nimbix Cloud. For reference, the original JARVICE platform predates Kubernetes.

Pattern	Example	JARVICE Support	Kubernetes Support
Service-oriented application (SOA)	Scale-out MVC (model/view/ controller) application – e.g.: content management system	Basic: can run single or multiple containerized images at predetermined scale, but does not provide automatic load balancing or HA.	Full: can run as a multi-container image and scale services individually, as well as provide service discovery and load balancing.
HPC: "embarrassingly parallel", or "perfectly parallel" application	Monte Carlo simulation	Full: container environment setup automatically spans multiple nodes with scale decided at launch; "master" node can set up and shard data, etc.	Partial: a Kubernetes "Deployment" can indeed launch multiple containers in parallel, but the scale is "best effort" at launch time as "gang scheduling" is not possible; also it is not possible for the application to perform different functions on the "master" container as the "slaves", so sharding and setup must be performed either in advance or manually. This is not architecturally compatible with most existing applications.
HPC: tightly-coupled parallel solver	Computational Fluid Dynamics (CFD)	Full: ready-to-run for MPI-initiated solvers, whether direct via mpirun or indirect via an application front-end to setup data and processing. JARVICE provides a dynamic cluster complete with fabric setup, SSH trust between nodes (parallel containers), and generated machine files for MPI, etc.	None: a Kubernetes "Deployment" is not suitable for this mechanism as it cannot guarantee scale neither at launch nor at runtime, does not support "gang scheduling" to queue jobs until all parallel resources are available, does not automatically elect and run code on a "master", and does not automatically configure fabric for applications.
AI: Accelerated parallel training	Distributed Deep Learning (DDL)	Full: supports HPC-style deep learning similarly to other parallel solvers, which is architecturally compatible with distributed training frameworks such as Horovod.	None: for the same reasons as for parallel solvers; alternative training workflows must be used (assuming framework support) when scale is needed.
AI: Accelerated real-time analytics	Inference	Full: when using technologies such as FPGAs and novel inference hardware, JARVICE can provision and host single or multi-node services at predetermined scale.	Full: assuming Kubernetes plugins exist for accelerated hardware, can support these stacks much like it does SOAs (see above).

HPC on basic Kubernetes

Since the platform itself does not provide adequate HPC support (as explained above), here are some options and workarounds.

Single Pod Solvers
If the need is multi-core/multi-thread rather than multi-node, an MPI-based solver can be provisioned as a single container in a single pod, and bound to a single node. This eliminates the need for fabric setup and "master/slave" type configurations. The solver can simply be launched to use shared memory interconnect on the provisioned CPU cores and threads. Depending on the underlying node capacity, this may suffice for some forms of HPC solves, but obviously restricts scale to whatever can fit on a single node.

External Control
For multiple pods, one possible workaround is to launch all containers in "standby mode" – for most applications this means simply performing an init or launching an SSH server explicitly. An external process can then discover what pods are actually available for a given deployment, what the container addresses are, and proceed to construct a machine file and launch the MPI-based solver to distribute the work. Obvious drawbacks are complexity, as this requires taking apart an application or running it in different stages, as well as managing individual containers explicitly. What's more, it still does not guarantee launch scale as the Kubernetes scheduler does not support gang scheduling, so rather than queue a set of pods until the full capacity is available, it will simply bind whatever it can and continue to do so until all pods are bound[1].

External control can also be performed manually, with a user watching the available pods and proceeding with the setup once the desired scale is bound.

Regardless, establishing SSH trust between containers will still be required – this can either be done with generic trust at build time (easy, but not secure), or explicitly after launch (more complex and difficult to automate in the application layer).

Embarrassingly Parallel Solvers
While a bit simpler to scale, embarrassingly (or "perfectly") parallel solvers still need setup and may require sharding data before launching. If the platform cannot coordinate a guaranteed set of containers and hand off control to a setup process automatically before starting work, similar problems exist even though the solver architecture lends itself better to a stateless replica style of system.
If the algorithm never needs to coordinate (not even at setup time), this is likely not any sort of HPC solver to begin with, but may lend itself well to run on a standard Kubernetes platform.

HPC on Kubernetes with JARVICE XE

JARVICE XE bridges the gap to running HPC codes on Kubernetes with 2 major advances:

Two LevelHPC Scheduler
The scheduler provides 2 levels, one that converts a traditional HPC job request into a set of Kubernetes pods, and a gang scheduler that binds pods to nodes, queuing entire jobs if the requested scale is not available. Additionally the gang scheduler provides the following important functions:

1. "Best fit", ideal for heterogeneous deployments –the JARVICE XE pod scheduler will always try to place pods on nodes with the fewest total resource, including accelerators, etc. This is different than taking load into account, and assuming there is no attempt at genuine oversubscription, results in better resource utilization.

[1] In Kubernetes terms, binding a pod means placing a pod (and its encapsulated containers) on a worker node for processing; this is the equivalent of a job scheduler choosing a node to run work on, and in turn running the work.

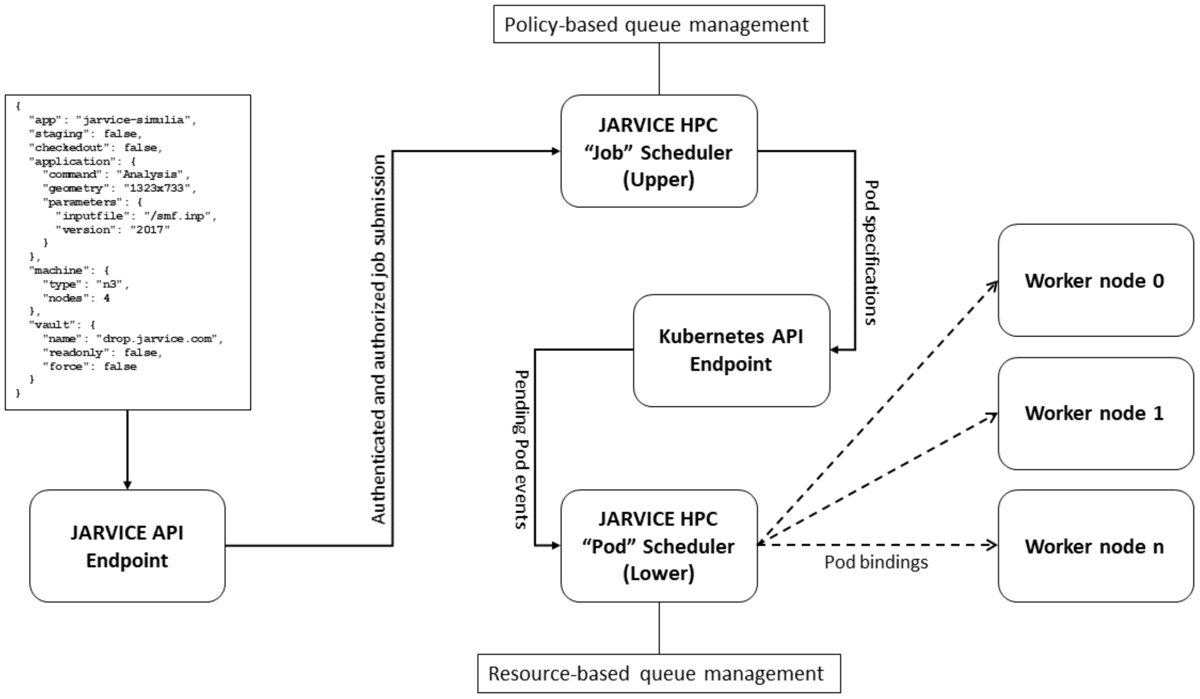
For example, in a cluster where a fraction of the nodes have GPUs in them, it does not make sense to ever schedule CPU-only work on them unless all of the CPU-only nodes are in use. If only current load is taken into account, this can easily lead to wasted cycles on nodes with more scarce and novel devices. This scheduler implements lessons learned in resource management on the Nimbix Cloud, which is a multi-tenant heterogeneous deployment.

2. Configurable resource weighting – different service providers may have different economics when it comes to what is more valuable – e.g. large memory nodes versus GPU nodes. The JARVICE XE pod scheduler can be configured to weigh these appropriately.
3. Advisory limit support – in multi-tenant or even multi-user environments it is sometimes necessary to restrict the amount and type of resource certain users and teams can consume. JARVICE XE supports self-service and administrator-imposed limits even if more resource is available at the time of scheduling jobs. This is different than the namespace limits Kubernetes already supports, because in that case trying to exceed limits results in failed object creation. In the JARVICE XE case, jobs will queue when limits are reached, which is more along the lines of what end users expect. When it comes to scheduling batch HPC jobs, this is still appropriate for "queue and forget" operation, where users don't need to worry about resource management at the time they schedule work.
4. Tenant isolation – in multi-tenant environments it's often important for security or compliance reasons to prevent nodes from running work belonging to multiple tenants. Even in single tenant environments there can still be regulatory restrictions between teams of users. Labeling nodes and defining resource types that target them is an obvious way to achieve this, but that is very static and may not yield the best utilization in cases where the exact machines are not restricted. Instead, JARVICE XE can ensure that dynamically, no two tenants or teams share the same nodes for work at the same time.

The upper level part of the scheduler also presents resource collections to users as "machine types", which is a much more natural way for end users to select scale and capability when running work. Machine types can of course request very granular resources, but this complexity can be abstracted from the end user, who for example simply decides to run a job on 32 16 core nodes for a total of 512 cores using MPI. JARVICE XE converts this request to the appropriate pod replica count and resource request before handing off to the pod scheduler for binding. This entire mechanism is opaque to the end user, making system operation much simpler.

The lower level part of the scheduler (also known as the pod scheduler) can in fact run side by side with the default Kubernetes pod scheduler, but of course race conditions may exist if competing for the same resources. Kubernetes does not use global critical sections when binding pods, so it's possible to oversubscribe resource (leading to pod eviction) if both the default and the JARVICE XE pod scheduler are trying to bind pods to the same nodes. The best practice is to use labels as well as taints to ensure JARVICE has exclusive control of scheduling on hardware used specifically for HPC. The JARVICE pod scheduler does support multiple namespaces, so it's in fact possible to have several deployments of JARVICE XE on the same cluster scheduling work at the same time without the risk of race conditions.

The scheduler is accessible via API or via point-and-click web portal. As mentioned above, JARVICE XE uses metadata from the applications in the HyperHub™ catalog to define workflows for the end user, rather than requiring users to write PBS or Slurm scripts to launch work.



```
{
  "app": "jarvice-simulia",
  "staging": false,
  "checkedout": false,
  "application": {
    "command": "Analysis",
    "geometry": "1323x733",
    "parameters": {
      "inputfile": "/smf.inp",
      "version": "2017"
    }
  },
  "machine": {
    "type": "n3",
    "nodes": 4
  },
  "vault": {
    "name": "drop.jarvice.com",
    "readonly": false,
    "force": false
  }
}
```

Policy-based queue management
JARVICE HPC "Job" Scheduler (Upper)
Pod specifications
Kubernetes API Endpoint
Pending Pod events
Authenticated and authorized job submission
JARVICE API Endpoint
JARVICE HPC "Pod" Scheduler (Lower)
Pod bindings
Worker node 0
Worker node 1
Worker node n
Resource-based queue management

Figure 3: JARVICE HPC Scheduler Architecture on Kubernetes: submission flow for a sample 64 core batch simulation

HPC Runtime Environment

The second major advance JARVICE XE brings to Kubernetes is the HPC runtime environment, created dynamically when jobs start. This environment performs the following functions:

1. Configures either a batch run or an interactive interface based on parameters from the scheduler's workflow request
2. Ensures finite completion of workflows, whether solvers succeed or fail (by default, using the Kubernetes "Job" API, failed solvers would restart automatically); since failures are a normal part of HPC job processing, it's important these are captured and treated as completions on their own, so that users have the opportunity to review workflow parameters and retry with the appropriate adjustments. Automatic restarts with the same parameters would result in an endless loop of failures that the end user may never detect.
3. Establishes a clear "master/slave" topology, with a "head node" design that initiates solvers to fan work out to worker nodes (slaves). This means traditional HPC codes can run unmodified on JARVICE, rather than implement complex master election logic before they can perform any work.
4. Automatically generates machine files that can be passed directly into mpirun command lines, for example, and ensures that slaves are ready to receive the request thanks to established SSH trust, fabric configuration, etc. Each job runs in its own dynamic cluster environment that presents itself quite naturally to traditional HPC software.
5. Supports any type of Kubernetes PersistentVolumeClaims, including ReadWriteOnce, and dynamically shares them across all nodes in a job. This means a user can launch a multi-node job with a block storage volume,

and the solver will automatically get shared filesystem access from all nodes without the risk of data corruption or indefinite queuing due to storage contention. JARVICE XE also supports NFS and CephFS shared filesystems directly without the need to manage Kubernetes volumes underneath, if so desired. Additionally, JARVICE XE's runtime environment can attach storage interfaces that are not natively supported in Kubernetes, by defining host-level mount points as part of machine definitions. This enables parallel storage systems such as WekaIO[2].

6. Supports network-mounted home directories, based on user login, including for automatically deriving user identity ID's (UID and GID) from login name and files in their network mounted home directories; this eliminates the need for complex configurations in containers in order to access ActiveDirectory, etc.
7. Manages secrets, configurations, services, and ingress automatically on a per-job basis. For example, if a user runs an interactive job, JARVICE XE will automatically configure an ingress for it so that the end user can access it remotely via a web browser.

As mentioned above, Nimbix provides web-based desktop environments as open source layers for any container, so this is a very effective way to provide graphical user interfaces to traditional HPC applications. JARVICE XE uses secure tokens to ensure users do not gain access to each other's desktops unless they explicitly share links or impersonate each other (an optional feature of JARVICE XE).

High Scale versus High Throughput Job Scheduling

Job scheduler requirements often conflate use cases and it's important to understand the difference. JARVICE XE, at the platform level, provides a high scale job scheduler, meaning that users can submit jobs that run on many machines at once. Jobs can however take several seconds to several minutes (depending on workflow complexity, container cache status, etc.) to start processing. The platform-level scheduler itself is appropriate for running large jobs that take a finite but considerable amount of compute time to execute. This is again inspired by serving on-demand HPC from the Nimbix Cloud for the better part of a decade. Most engineering/simulation or distributed deep learning jobs take several minutes, hours, or days to run, even at scale.

For high-throughput scheduling, it's best to not rely on the platform itself to run the individual jobs. This is where it's appropriate to schedule a single large job as a dynamic cluster, and then embed a job scheduler inside it. Nimbix provides a simple example pattern on HyperHub for this called "HPC Test Environment" (container source available on GitHub), which actually deploys a Slurm cluster dynamically that is ready to schedule work once it starts. While commonly used for testing or for providing on-demand academic-style clusters to end users, this base image also fits certain use cases very well, like many bioinformatic codes that tend to run array jobs on sequencer data, etc. Once the job starts and the embedded scheduler is fully configured, job submission latency within the dynamic cluster is no different than on physical static systems. It then becomes efficient to run many short-lived jobs without incurring platform overhead of provisioning containers and other objects on infrastructure underneath. The obvious drawback is that in an on-demand environment, a user consumes all of the resource for the duration of the dynamic cluster's lifetime, versus jobs ending and

[2] Host mounts bound into containers should be used with great care and only when necessary, such as when the storage technology does not lend itself to being managed with technologies such as the Container Storage Interface (CSI). The JARVICE platform provides appropriate controls for specific patterns only.

relinquishing resources automatically in the high-scale platform scheduling model. Either users or applications themselves must actually terminate the dynamic cluster in the embedded scheduler model in order to return resources to the system.

Converged IT Infrastructure with Kubernetes and JARVICE XE

JARVICE XE typically runs as a set of services on a Kubernetes cluster, deployed via Helm chart. It's trivial for cluster administrators to apply updates and manage JARVICE XE itself as any other framework on the system. JARVICE XE handles the entire user interface above that, including user HPC applications, identity, authentication, authorization, job control, accounting, and auditing. It supports multiple tenants (or teams) without having to deploy separate instances. In short, it makes enabling HPC on a large Kubernetes deployment as straightforward as deploying any other types of applications.

Converged deployments benefit organizations in the following ways:

1. Reduction of specialized skills the same team that manages the IT infrastructure can now manage HPC with minimal training; since JARVICE XE delivers curated workflows from HyperHub, this includes HPC application engineering as the workflows are already ready to run.
2. Improved utilization – while the best practice is to segregate HPC and commodity workloads on the same cluster, there is no need for redundant control planes, storage, or networking; this improves economies of scale of a cluster, and makes it easier to plug in specialized hardware without "step functions" and complex administrative interfaces.
3. Data sharing between workloads a common use case would be combining an SOA application with an HPC one on the same infrastructure. For example, a full deep learning pipeline could feature an SOA-based inference service that reads trained models from the same storage that an HPC-based DDL training workflow produces either on demand or continuously. The SOA could trigger training via web service API when there is new data, or for reinforcement purposes, and JARVICE ensures that the DDL training architecture runs correctly.
4. Migration and recovery simplification – cluster administrators can rely on the same tools and techniques to handle commodity versus HPC workloads when there is a need to move, replicate, or recover a cluster.

Multi and Hybrid Cloud with JARVICE XE

Because JARVICE XE provides a processing API and a common, synchronized service catalog (from HyperHub), workload mobility is very seamless. The same workloads run on any cluster providing JARVICE XE, including the Nimbix Cloud which retains compatibility.

Even in the case where different cloud infrastructures provide different types of resources, JARVICE XE applications retain compatibility, and well-defined patterns for machine types and resource requests ensure portability. JARVICE XE does not currently manage data movement and migration across multiple clouds, but supports whatever mechanisms are employed underneath to achieve this. In the case where users do not ever run work on multiple clouds (but different teams of users do), this is generally not an issue as the data will reside with the compute at all times.

JARVICE XE provides some additional features to ease multi-cloud deployments:

1. Helm chart deployment on Kubernetes clusters makes managing JARVICE XE on multiple infrastructures seamless, including deploying updates and changing deployment parameters

2. Scripted support for managed Kubernetes endpoints on public clouds such as Amazon EKS and Google GKE; this is a higher level abstraction than the Helm chart itself, supporting single command deployments of JARVICE XE in a matter of minutes.
3. Public and private application synchronization across clusters; not only are upstream HyperHub applications synced, but also select private and custom applications, further improving workload mobility.
4. Single sign-on with and without federation; JARVICE XE supports both Active Directory and SAML2, allowing user identity to follow across multiple deployments with consistency.
5. "Single pane of glass" – the same JARVICE XE interface operates any cluster
    a. Available Q4 2018: URL-based federation – users visit the cluster or zone they wish to run work on by targeting its user portal ingress URL.
    b. Available Q2 2020: administrator and team-controlled federation from the same portal URL –users launch and manage jobs on multiple clusters from a single web portal URL.
    c. Available Q2 2020: bursting of jobs from one cluster to another based on site-configured policies and rules; in the JARVICE multi-cloud model, machine types are pinned to clusters, so burst policies define which machines to leverage (provided all other constraints are met) when the originally requested ones are busy.

Because multi-cloud deployments are often a technique for security and compliance, JARVICE XE can be used to restrict access to compute locations for tenants, teams, and individual users based on login credentials and group memberships. This can also include access to applications (e.g. to enforce export-control), data sets, and even hardware.

In all cases, the underlying management, whether single or multi cluster, is completely transparent to the applications themselves. This truly enables a "publish once, run anywhere" model for HyperHub application containers. containers.

Conclusions

- Kubernetes is emerging as the standard Enterprise container deployment platform, but does not support HPC
- JARVICE in the Nimbix Cloud is the leading on-demand HPC platform for containerized applications
- JARVICE XE brings HPC to Kubernetes-managed infrastructure, including a rich (and ever expanding) catalog of ready-to-run open source and commercial workflows
- JARVICE XE provides a runtime environment for traditional applications that can run unmodified and not need to be converted to container-native architectures; this is especially valuable when deploying commercial stacks where the source code is not available nor the architectures configurable enough to change dramatically
- JARVICE XE supports private, hybrid, public, and multi-cloud deployments
- JARVICE XE makes it easy for Enterprise IT departments to add HPC to their portfolio of services without having to build additional practices, tooling, or specialized skills

www.Nimbix.net

linkedin.com/company/nimbix facebook.com/nimbix twitter.com/nimbix