TECHNICAL PAPER

Understanding Containers and SAS® 9.4 Container Deployment

Last update: November 2018

Contents

Introduction

Organizations have many options for how to build and deploy new applications and software, and the pace of change in these options is increasing. Many organizations want to take advantage of deployment benefits such as private and public clouds and of virtualization technology such as containers and virtual machines. SAS is keeping up with the increasing options and collaborates with its customers to properly build SAS software for many different environments.

As organizations adopt containers, one of their main questions is, "Does SAS software run on containers?" The simple answer is yes. The more detailed answer is that although SAS can run on containers, organizations need to understand the nuances of containers. They need to properly build a larger container environment to ensure that SAS software has the right resources to perform well in this environment. The goal of this paper is to cover key considerations when moving SAS software to a container environment. To do this, the paper reviews containers at a high level, helping readers understand how containers evolved, how orchestration engines work with containers, and what the benefits and limitations of containers are. With this background, the paper concludes by discussing key considerations when moving SAS 9.4 to containers, and a sample architecture is reviewed.

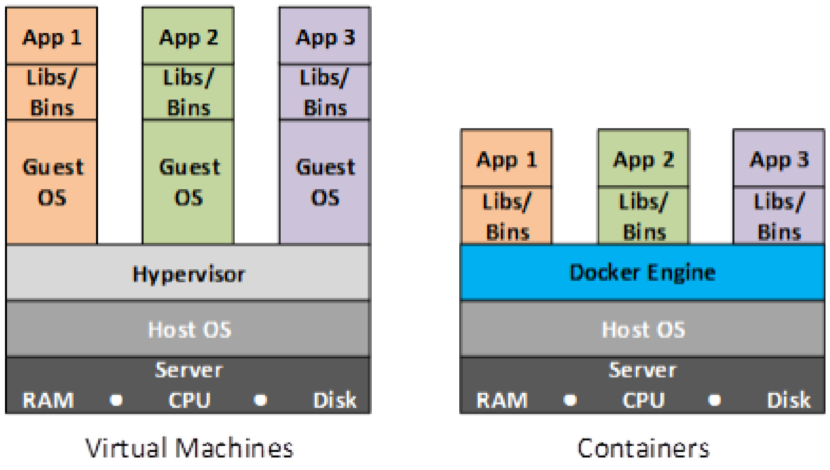
Evolution of Containers

IT organizations are tasked with managing critical systems that provide value to their organizations. Over time, hardware has become more powerful. Prices have decreased to the point that hardware can be bought as a commodity. IT shops are looking for ways to get more value out of their hardware and to streamline the management of software applications and systems key to their organizations.

Around the year 2000, the concept of dividing one piece of physical hardware into multiple smaller virtual servers or virtual machines (VMs) became commercially viable. This concept has experienced wide-scale adoption over the years. The technology works well, and it provides IT organizations greater flexibility in how they manage their servers and deploy applications. The technology works particularly well for applications that are expected to always be running and available to users. However, the VMs for these always-on applications are often sized for peak workload, which results in low resource utilization during off-peak hours.

The next evolution in IT was the advent of the cloud. There are many variations of cloud environments. This paper primarily focuses on public clouds. A public cloud is essentially a series of large data centers created by a vendor such as Amazon, Google, or Microsoft. These vendors procure huge quantities of commodity servers and storage. They allow third parties to pay to use these resources for their own applications and processing. The promise of the cloud is that companies can move away from the management and overhead of having their own data centers. They can focus on getting the most value out of their applications. Moving to the cloud offers organizations the ability to control costs by opening software applications only when they need them and by avoiding idle resources and the potential operating costs of those resources. Similarly, the vast resources of the cloud have made it much easier and cost effective to scale applications to additional cloud instances on demand as workloads spike, rather than maintaining a physical or virtual server large enough to handle peak workloads. This greater flexibility exposed some weaknesses in VMs, which led to the concept of containers.

There are several differences between VMs and containers. The following diagram helps illustrate some of them:

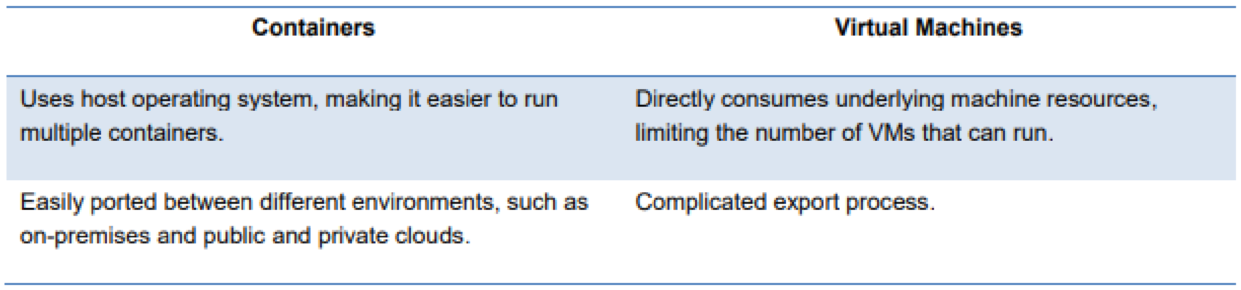


Both VMs and containers use an underlying compute infrastructure (think hardware resources such as CPUs and RAM). An operating system is used to manage the infrastructure. At this point, the very different approaches become apparent for how each technology uses the underlying infrastructure to distribute resources to applications.

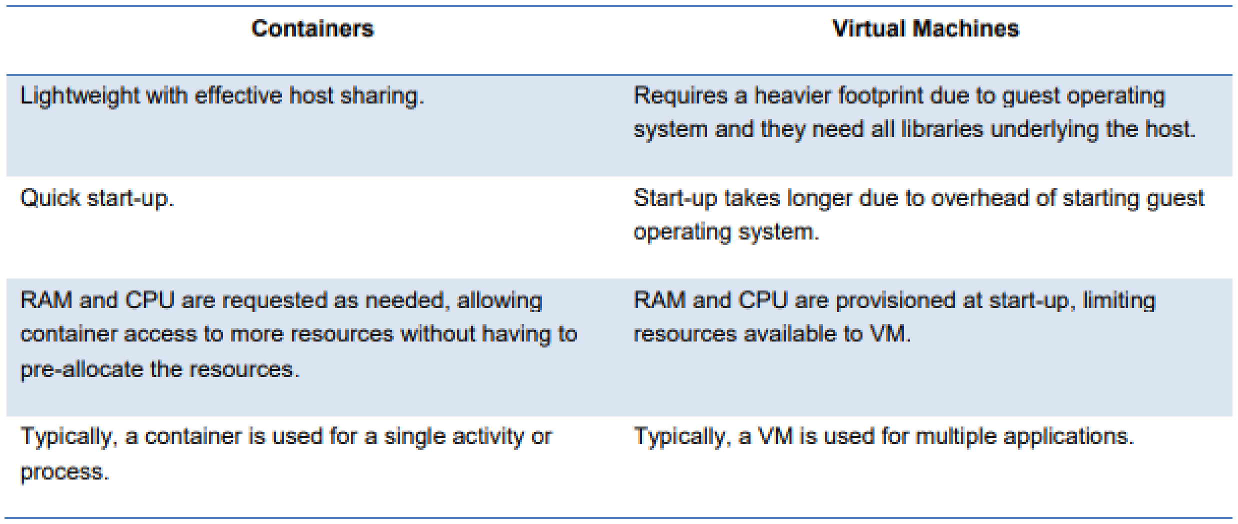
A hypervisor is used to monitor and control the VMs, which allows administrators to assign the available resources such as RAM and CPU among the individual VMs. Once the resources are assigned, each VM acts independently with its own operating system (referred to as a "guest operating system") that can be tuned separately for each VM. This design enables administrators to control VM resources, ensuring that an application has the resources that it needs. However, pre-allocation of resources introduces some performance risks. For example, pre-allocation can result in idle host resources reserved for a specific VM, and these resources are not available to other VMs. Similarly, pre-allocation can result in a VM that hits its resource limit and cannot obtain additional resources, even when the host operating system has available resources. Over time, VMs have evolved to minimize these limitations, but the net result is that administrators must limit how many VMs run on a host to ensure that each VM meets performance expectations.

In contrast, a container platform such as Docker manages isolated containers and the applications that reside in that container. Containers do not require a separate guest operating system. Rather, they use the same operating system as the underlying infrastructure. This removes the operating overhead of starting and running a guest operating system. As a result, the container uses fewer resources than a VM, and starting a container is much faster than starting a VM. As containers are initialized, all application components are included in the container so that it can run as an isolated unit. Because containers share the same operating system as the host, administrators do not need to allocate or reserve resources for a container to start up. This configuration enables users to spin more containers off the underlying infrastructure and have more efficient utilization of that infrastructure.

The following table summarizes the differences between VMs and containers:



Containers	Virtual Machines
Uses host operating system, making it easier to run multiple containers.	Directly consumes underlying machine resources, limiting the number of VMs that can run.
Easily ported between different environments, such as on-premises and public and private clouds.	Complicated export process.



Containers	Virtual Machines
Lightweight with effective host sharing.	Requires a heavier footprint due to guest operating system and they need all libraries underlying the host.
Quick start-up.	Start-up takes longer due to overhead of starting guest operating system.
RAM and CPU are requested as needed, allowing container access to more resources without having to pre-allocate the resources.	RAM and CPU are provisioned at start-up, limiting resources available to VM.
Typically, a container is used for a single activity or process.	Typically, a VM is used for multiple applications.

Container Orchestration

A large benefit of containerization is the ability to scale the number of containers that run on the underlying infrastructure. As the number of containers increases, IT organizations need a system to help to automate deployment and to manage and scale containerized applications. Container orchestration engines fill this role by allowing organizations to work with containers on an enterprise scale.

There are many container orchestration engines available. Engine functionality and the terminology used to describe functionality can vary by engine. Many engines are based on Kubernetes (https://kubernetes.io/). This paper uses Kubernetes as an example as the general features of orchestration engines are examined.

One of the main features of the orchestration engine is to manage application and environment complexity. Organizations need to deploy simple and complex applications in containers. Orchestration engines help manage more complex applications by using groupings and application maps to manage larger applications and the complexities associated with them. For example, Kubernetes provides the following features to manage application and environment complexity:

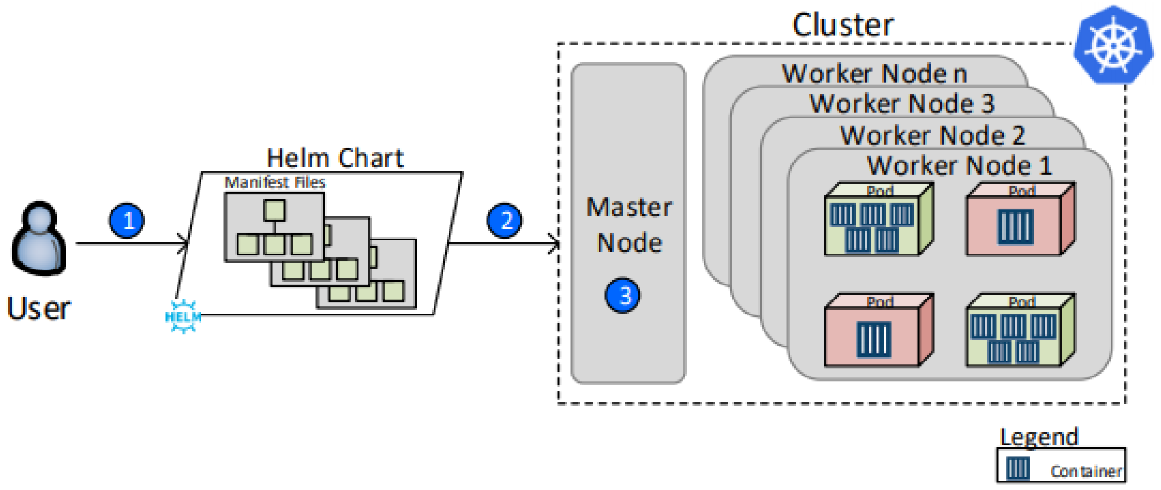
- Pod – A pod is the smallest entity Kubernetes can manage. It consists of one or more related containers. Kubernetes deploys a pod onto a single hardware host, which ensures that all the containers in the pod can talk to each other. Simple applications that can be deployed in a single container are deployed to just one pod, offering little additional advantage over the container. However, complex applications that need to deploy multiple components in multiple containers with dependencies experience advantages from this layer of abstraction. For example, applications that use microservices can deploy each microservice (with its unique dependencies) to its own container, and then use pods to group related microservices and to capture additional dependency information in the pod.

- Manifest Files – A manifest file is used by Kubernetes to provide instructions and configuration information for deploying all the components of an application across multiple containers. A manifest file provides a map of how the components work together, communicate with each other, and find each other when a component must be updated or recovered. A deployment can use multiple manifest files, which are generally organized using Helm Charts.

- Node (also known as a Worker Node) – A node is an individual host that can run one or more pods. The underlying host can be a physical server, virtual machine, or a cloud instance. The node concept adds a layer of abstraction from the underlying architecture of the host. This enables Kubernetes to focus on the available resources for pods, such as CPU and RAM, rather than on the details of the host itself.

- Cluster – A cluster is a group of one or more nodes. Kubernetes pools the resources of all nodes in the cluster, making all the resources available to containerized applications.

The following diagram highlights these relationships. Notice the master node in the diagram, which is used to manage the worker nodes.



Here is what this conceptual diagram depicts:

1. A user initiates a request to start an application.

2. Kubernetes looks for the proper map that explains how to set up the application. The map details how many pods need to be started, the containers to place in each pod, and the dependencies within and between pods. For simple applications, the map could be a single manifest file that details simple dependencies. For a complex application, the map could be a Helm Chart that contains several manifest files that detail the application's complexity and dependencies.

3. The details of the map from step 2 are passed to the Kubernetes cluster master node to initiate the pods required for the application. (Pods contain one or more containers.) The master node distributes the pods over the available worker nodes in the cluster.

Outside of managing complex environments, orchestration engines address:

- Storage orchestration – Containers typically are designed to hold a general-use application. That application needs additional data to perform. In SAS terms, a SAS 9.4 container holds the SAS application libraries and executable (that is, the application files stored in the SASHOME directory), but not the other components needed to run a job such as data or a program. To make the other components available to the application, the container needs access to external storage. The orchestration engine can map that storage to a container when the engine is started.

- Resource scaling/bursting – The modular nature of containers makes them very appropriate for being deployed in the cloud. One feature of the cloud is the ability to deploy applications on the minimum amount of resources needed. Then, you scale up the resources when workloads increase. Orchestration engines enable organizations to set up scaling strategies for containers, to specify the minimum size of the cluster available to containers, to establish policies for when to add hosts or burst up the size of the cluster, to specify the largest size the cluster can grow to, and to determine when to remove hosts from the cluster.

- Resiliency – Containers need to be reliable so that when a user calls a containerized application, they can depend on the application performing the expected task. Orchestration engines help ensure this reliability by adding resilience to containers so that the container automatically recovers from events such as container failure, hardware failure, and other container-health-related issues. The orchestration engine monitors container and resource state. When the container encounters an issue, it restarts the container and cleans up the issue. For example, when a node running one or more containers dies or goes offline, the orchestration engine reschedules all containers running on that node.

- Batch workloads – Orchestration engines can assist an organization in scheduling batch jobs. They provide the details of what items need to flow into a batch job such as data and programs, as well as where to place output from the batch job.

Although this paper focuses on Kubernetes, SAS is unbiased about which orchestration engine an IT organization chooses. Organizations can choose the orchestration engine that best meets their use cases and internal standards. Here are some popular orchestration engines:

- Docker Swarm (https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/) is a container orchestration engine provided as part of the core Docker Engine.
- OpenShift (https://www.openshift.com/) is provided by Red Hat and is based on and extends Kubernetes.
- Mesos (http://mesos.apache.org/) from Apache is a managed open-source general orchestration engine that can be used to manage containers.
- Amazon Elastic Container Service (ECS) (https://aws.amazon.com/ecs/) is specific to AWS and Docker.
- Amazon Elastic Container Service (ECS) for Kubernetes (https://aws.amazon.com/eks/) is specific to AWS and Kubernetes.
- Azure Kubernetes Service (AKS) (https://azure.microsoft.com/en-us/services/kubernetes-service/) is specific to Microsoft Azure and Kubernetes.
- Google Cloud Kubernetes Engine (https://cloud.google.com/kubernetes-engine/) is a Google Cloud platform-specific Kubernetes offering.

Benefits of Containers

As you better understand the evolution of containers and how they compare to VMs, several benefits of containers and container orchestration engines become apparent.

- Cloud native: Containers are designed to take advantage of the power and benefits of the cloud. This includes resiliency, the ability to scale, and the ability to quickly deploy new applications. The main benefit of being cloud native is that public cloud providers make it very easy to start new container environments by providing orchestration services. For example, Amazon provides Amazon ECS, Google provides the Kubernetes Engine, and Azure provides AKS.
- Dynamically scale: Containers can scale vertically or horizontally. Containers can be set up to run on one or more underlying hardware infrastructures. If administrators start to see resource constraints in their container environments, they can scale vertically by increasing the size of the underlying hardware servers. Or, they can scale horizontally by distributing work to additional servers. For on-premises container deployments, an orchestration engine such as Kubernetes can add hosts to the container cluster. In the Google Cloud Platform, you could use the Google Kubernetes Engine to horizontally scale a cluster. In AWS, you can use an AWS Kubernetes engine or a service specific to AWS (such as AWS Auto Scaling) to dynamically scale the environment when specific conditions are met.
- Quick spin-up: Containers share the underlying operating system of the hardware infrastructure that they run on. They do not have the start-up overhead of booting up their own operating system when instantiated. This results in containers starting in seconds, making it ideal for on-demand batch and interactive work.

- Better utilization of hardware resources: Containers have direct access to the CPU and RAM of the underlying infrastructure, which enables them to request the resources that they need at run time, instead of pre-allocating the resources like VMs. Couple this with the fast spin-up and spin-down of containers, and you can distribute more containers on hardware than you can with VMs. This is often referred to as “higher container density” on the same hardware.

- Portability: Containers are decoupled from their operational environments, which enables each container to run as an isolated unit. As a result, containers are ideal for porting applications between environments. This portability means you can easily migrate containers between on-premises deployments and the cloud, as well as between different public and private cloud vendors.

- Workload isolation: Each container is a self-contained unit. Containers are not aware of other containers running on the same host. They cannot impact the processes running on another container in the host. This is referred to as “container isolation.” Orchestration engines can be used to apply multiple layers of isolation to a container. For example, the components of an orchestration engine (such as pods, nodes, and clusters in Kubernetes) add layers of isolation.

- DevOps benefits: DevOps is concerned with the smooth transition of an application through the software development lifecycle (SDLC). Containers offer several advantages:
    - Consistent Dev environments: Containers can be created once and deployed multiple times. Deploying development environments as containers enables organizations to provide each developer with their own Dev environment, while ensuring that the environments are consistent. This ensures that the work that one developer does in their environment does not impact other environments.

    - Eliminate differences in the multiple environments in the SDLC: The SDLC usually consists of multiple levels of work in different environments such as Dev, Test, and Prod. Each level in the SDLC is typically a completely isolated environment. Companies work hard to maintain consistency between these environments to avoid environment-specific rework on applications as they progress through the SDLC. Using containers for all levels in the SDLC ensures a consistent operating environment for applications as they progress through the SDLC.

    - Flexibility: The features of containers offer DevOps a great deal of flexibility in how it deploys its different levels in its SDLC. For example, companies can choose to deploy multiple levels of their SDLC to one container cluster (that is, to Dev and Test in the same cluster). Or, they can use the portable nature of containers to distribute each level in the SDLC to different container clusters.

    - Ease of upgrades: The nature of a container makes it straightforward to apply software changes to one underlying container image. Then, changes are propagated to different levels in the SDLC after proper testing.

• Cost savings: The ability to dynamically scale containers, get better utilization of hardware resources, and achieve simplification of overall application maintenance represent opportunities to reduce the cost of operating and supporting applications.

Container Limitations

There are several significant benefits to containers, but it is important to consider the limitations. Many of these limitations can be overcome in a properly planned environment.

- Ephemeral – Containers are designed to be transient, where the container lasts only as long as it takes to complete the required task. Well-designed containers tend to have just one application and are general use. For example, a SAS 9.4 container typically installs only the SAS software executables and configuration. It does not have the items required to run a specific job, such as the data or a program. Because containers are transient and tend to have just one application installed, a container does not have the means to save additional information within the container itself. This means that IT shops need to have a strategy to store inputs and outputs of containers in a centralized place that can be made available to containers.

- SAS 9.4 stateful components – The SAS 9.4 Intelligence Platform has several components that require a stateful environment. These components include the SAS Metadata Server, SAS middle tier, and SAS Java clients such as SAS Enterprise Miner. It is difficult to deploy these components in a containerized environment and retain the benefits of a container. Currently, SAS recommends that these types of stateful SAS implementations be deployed in VM environments or directly on cloud instances rather than through containers.

- SAS client availability – Currently, the main clients available in a SAS 9.4 container are web clients such as SAS Studio and Jupyter Notebooks. These clients can be included in the container. All components required to run the client reside in the container.

- Job size – Containers are not designed to run extremely large distributed jobs that require hundreds or thousands of cores. This is because the container is limited to the amount of resources available on a given host, making that host's RAM and CPU resources the maximum amount that a job or container can work with.

Key Considerations When Moving SAS to Containers

There are key considerations whenever someone builds a SAS environment. These include I/O bandwidth, compute resource requirements, job automation, and how users interact with the new environment. These same considerations exist when organizations look to migrate SAS to a containerized environment. This paper examines some of these considerations at a high level. However, it is best to work with your SAS account team as you build your environment to make sure it will best meet your needs.

I/O

SAS is designed to deal with very large volumes of data. This can cause SAS to be an I/O-intensive application. When SAS encounters performance problems, I/O is often the main issue. Properly accounting for I/O in your architecture is key to a performant environment. In considering the complete I/O needs of SAS, it is important to first understand the aspects of SAS that generate I/O.

1. Input and output data – SAS can work with input and output data in a wide variety of formats, including text files, SAS data sets, relational databases, and Hadoop. Flat files and SAS data sets need to be stored external to the container, typically in a storage device such as in a SAN or in an AWS Elastic Block Store (EBS). Relational databases and Hadoop are external to the SAS container. They should be set up with the proper resources to handle the requests that SAS and other applications place on the system.

   Regardless of the data format, the key to good I/O performance is to co-locate SAS in the same data center/availability zone as your largest and most frequently used data. Ensure that you provide proper I/O throughput to those data sources. For databases, you want a strong network connection in the data center, such as a 10 GB Ethernet or faster. For flat files and SAS data, you want robust throughput, where SAS recommends at least 100 MB/s per core of throughput to your container cluster.

2. SASWORK – This is a location where SAS writes intermediary files that do not persist between SAS jobs. For example, if you sort a data set in SAS using one variable (such as geography) for the sort, SAS makes a temporary file that is twice the size of the original data to perform the sort. The temporary file goes away after the sort is complete. Users can choose to store temporary copies of data in SASWORK that persist until the session ends or the user deletes the data. SASWORK can quickly grow. It is important to provide adequate space for SASWORK to maintain good performance.

   SASWORK can point to ephemeral storage in a container or to an external storage location, depending on how much space you anticipate needing for this temporary storage. If you do store SASWORK in external storage, there is no need to back up this storage. However, proper I/O throughput still applies to external storage. SAS recommends at least 100 MB/s per core of throughput to SASWORK.

3. Swap space – This is a storage area that can be used when physical memory is full. Swap space should be 1.5 times the size of physical RAM or 250 GB, whichever is less.

4. SAS user files or home directories – This is a general category that addresses files that users need to run a job or that users generate as the output of a job. These files include user-specific configuration files, user programs, SAS program logs, and the output of jobs in a variety of formats such as PDF or HTML. These files tend to be smaller in size, so I/O throughput is not as important. However, architects should ensure they provide adequate storage for each user for storing these files. This can be done in storage devices such as a SAN, on a file server for on-premises implementations, or in cloud-specific locations such as AWS S3 for cloud deployments.

RAM

RAM is an important consideration when building a SAS environment. SAS tries to do as much work as possible in memory when it is processing data. Because SAS often works with large data sizes, the performance of a job can be greatly impacted by providing SAS with additional RAM. SAS typically recommends that you provide at least 8 GB of RAM per core in your container cluster to ensure a well-performing environment.

CPU

The final consideration for a performant SAS environment is CPU. Traditionally, SAS recommends that organizations provision enough CPUs to handle their peak workloads. This leaves the potential for idle CPUs, which is not an efficient use of resources. In contrast, containers make it very easy to start with a modest container cluster, and then to dynamically scale up the cluster as workloads require additional resources. The idea is that an organization should start with a minimum cluster size of one or more hosts, and then determine the conditions when an additional host needs to be added, such as when the available cluster hits 80% CPU utilization. Organizations can set up a maximum cluster size so that the cluster never scales above that size. When workload on the cluster starts to taper off, organizations can set up criteria for dropping hosts from the cluster.

To take advantage of this flexibility, organizations need to determine the scaling or bursting strategy and the technology that makes the most sense for them. There are numerous ways to accomplish bursting, including using orchestration engines like Kubernetes or cloud-native equivalents like Google Kubernetes Engine and Amazon ECS for Kubernetes. The next section of the paper contains sample architectures that include the ability to burst to additional hosts.

One other CPU-related factor to consider is how many CPUs to make available in each host. The maximum CPUs available to an individual container is determined by the maximum number of CPUs available on the underlying host.

Job Execution

There are non-performance-related considerations that need to be included in any properly architected environment. One key consideration is how to make SAS available for use. Many organizations considering containers have numerous batch jobs that need to be automated. These are often referred to as “production jobs.” These jobs can efficiently be automated through an orchestration engine, such as Kubernetes, or through a native cloud service, such as AWS Batch. These orchestration engine options enable you to schedule jobs and provide mechanisms to specify where to pull inputs needed for the job and where to push results from the job.

When specifically considering SAS batch jobs, the following job components need to be accounted for:

- SAS batch program – The orchestration engine needs to push the SAS batch program (*.sas file) to the container. This is what SAS software executes.

- Data – Input data for the batch program can be provided in several ways. Flat files and SAS data sets can be proactively pushed to the container as part of the batch start-up process. Data in a variety of formats can be pulled into the container by the SAS batch program when it is needed. To accomplish this, the container needs to have external storage mounted that points to SAS data sets or flat files. Or, the container needs access to systems that host relational databases or Hadoop. The path to these data sources (what SAS refers to as a “LIBNAME”) can either be embedded in the batch program itself or stored in a SAS autoexec file that can be associated with the container at start-up.

- SAS log file – SAS batch programs create a log file (*.log file) that contains the details of the job run, including any errors or warnings that were generated by the batch job. It is a best practice to push these files to a location that persists when the batch job completes and the container spins down.

- SAS output – Administrators should account for output the job creates, including output data, SAS output files 9 (*.lst), and other output.

Most organizations have a need to provide users with a live or active connection to SAS for program creation or to interactively run SAS jobs. These interactive SAS jobs have many of the same considerations as batch jobs do, along with additional items that are related to the interactive nature of the work.

- SAS files – SAS users need to have a location to store files specific to SAS that are related to their SAS jobs. These files include the SAS programs that they are writing, any log and output files that they want to retain from a job, and potentially user-specific SAS configuration and autoexec files. These files are unique to the user (or to a group of users), need to be stored in persistent storage outside of the container, and should be mounted to the container when it starts up.

- Data – Users need access to data for their SAS job. Data can come in a variety of formats, including SAS data sets and flat files, relational databases, and other sources such as Hadoop. SAS data sets and flat files need to be stored on external storage that is mounted when the container starts up. Relational and Hadoop data sources need to be visible to the container in a way that the container can communicate with the database when users specify connection information within a SAS program. Administrators or users can choose to pre-define connections to commonly used data sources in a SAS autoexec file that is either embedded as part of the container image or that is available on a mounted external storage device.

- Client interfaces – This paper is focused on SAS 9.4 programming-only interfaces. Users need an interface to interact with SAS 9.4. The two common interfaces used for this purpose are SAS Studio and Jupyter Notebooks.

    - SAS Studio is a web-based interface that comes with SAS 9.4. SAS Studio and the web-tier components associated with it are deployed in the SAS 9.4 container image. Once the container is spun up, users can use a web browser on their local PC to connect to SAS Studio.

    - Jupyter Notebooks are another web-based interface that can be used to interact with SAS 9.4. Jupyter Notebooks are an open-source way to connect to different applications. SAS has provided a SAS_KERNEL within GitHub to allow Jupyter Notebooks to connect to SAS 9.4.

• Overstuffing a container – "Overstuffing" refers to building a container that includes more than one application. Although it is most common to see containers built to house a single application, there are instances where organizations might choose to overstuff a container. For example, data scientists that want access to SAS and open-source tools like R and Python might want to have a container that includes all of these applications.

Code Changes

The final thing to consider when moving SAS 9.4 to containers is code changes. The majority of SAS code remains the same regardless of the underlying infrastructure and operating system. However, there are certain key circumstances that might require changes as organizations move toward containers.

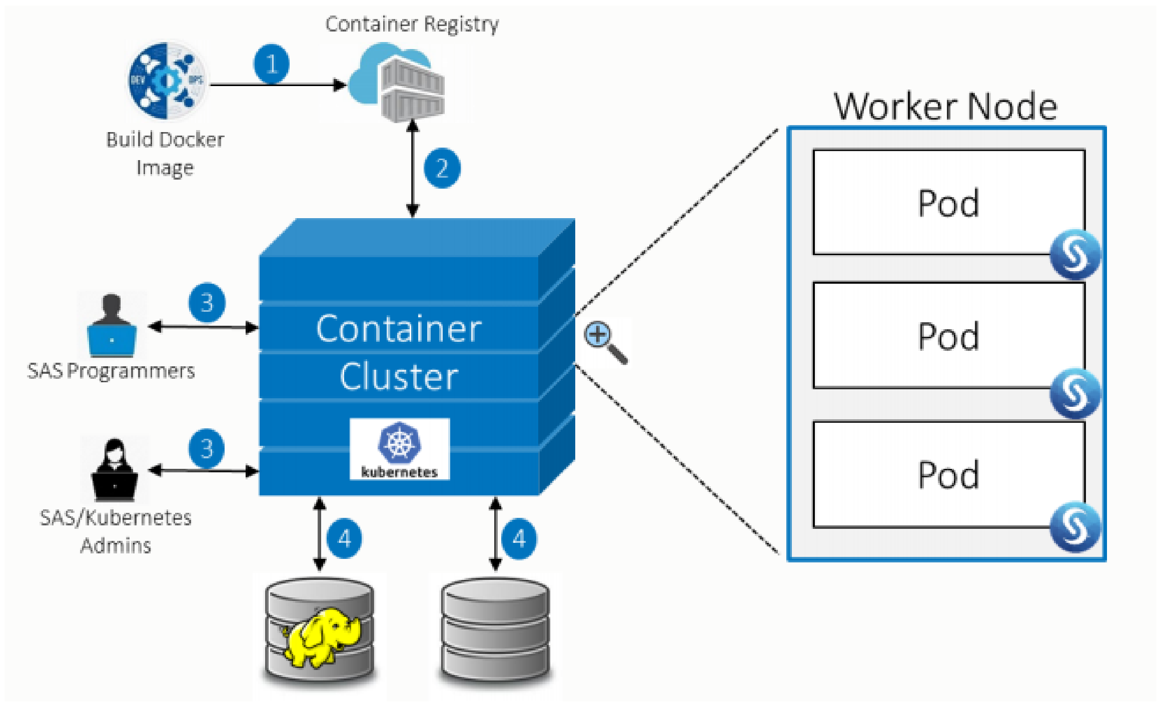
- Product mix changes: It is a best practice to have your SAS container environment include the same SAS products as your previous SAS environment. If the new container environment does not include the same products as the previous SAS environment, then any SAS procedures that are provided with the missing product will not work and need to be removed from existing programs.

- References to an old file system structure: Ideally, an organization should have the file directory structure mounted to the container resemble the file directory structure on its previous SAS system, especially for directories that contain SAS data or other files that a program might call. This allows SAS statements that reference the directory structure, such as FILENAME and LIBNAME, to operate properly without any changes to the code. However, when it is not possible to mirror the previous SAS directory structure, organizations need to update any path this is included in a user-written SAS program.

- Operating system migration: Typically, the operating system that the container environment uses is similar to the operating system of the environment where a SAS program was created (that is, Windows to Windows or Linux to Linux). However, the container operating system might not match the operating system in the environment where the code was created. When that is the case, you should review code for operating-system-specific references such as the file structure (that is, Windows using c:\dir structure versus Linux using /dir) or operating system commands within your code. These references tend to be after the X command or SAS statements such as LIBNAME and FILENAME.

Reference Architecture

Container environments are very flexible. Organizations have lots of options for how they choose to build their container environments. For example, organizations can choose to run their containers on premises, in a public cloud, or in a private cloud. Once an organization decides where it will run the container, it has several options for containerization software and the orchestration engine. The list of options is endless, and it is difficult to cover all the options an organization might have in one reference architecture diagram. The goal of this section is to present a high-level conceptual architecture diagram and use it to explain the flow of how a SAS job would run in a containerized environment.



Here is the flow of the container process:

- The first step in any containerized environment is to build the container image itself. There are many ways to build this container image, and SAS continues to work to simplify the process. Once the container image is complete, it is registered to a container registry (step 1), making it available for container calls.

- When a user or batch process needs a container, the user or a scheduling application asks the orchestration engine to request a container. The orchestration engine pulls the current copy of the container image from the container registry (step 2).

- As the orchestration engine pulls the container image, it looks at any container configuration files associated with the container image. These configuration files can specify quite a bit of information, including what storage devices to mount to the container. Similarly, the process calling the container can specify any additional steps that need to occur with the container. For example, the calling process can specify SAS content, such as data, to push to the container as it spins up.

- Once the orchestration engine understands the requirements of the container image, it finds a worker node in the container cluster that has the bandwidth to accept the new container and start the container on that node. As a reminder, one or more containers can be run in a pod on an orchestration engine, and a worker node can contain more than one pod. (The diagram illustrates this by showing an expanded worker node on the left.)

- When a container is available, the appropriate users have access to the container and can run SAS code on the container. For interactive SAS jobs, the users access SAS through web-based clients (step 3) such as SAS Studio and Jupyter Notebooks. For batch jobs, users and administrators have access to the output of the job once the job is complete. That output needs to be stored on external storage to persist after the container shuts down.

- SAS jobs being run in containers need access to external data and other content. This data and other content can be pushed to the container at start-up (as detailed in the third bullet). Or, the interactive user or batch program can pull the data and content from external storage devices (step 4). These devices can be typical storage subsystems like a SAN or NAS, which holds SAS files, SAS data sets, and flat files. These devices could be structured and semi-structured storage applications such as a relational database or Hadoop cluster.

- Any SAS work performed in the container that the user or batch job wants to persist needs to be written out to external storage devices (step 4).

- Once the SAS batch job is complete or the interactive user is done with their session, the container shuts down. As the container shuts down, any work done in the container that has not been pushed to external storage is lost. The copy of the container image is discarded.

Conclusion

Container adoption is on the rise. Many organizations are exploring the containerization of applications. From a SAS perspective, containers are one of many architectural options that SAS supports. This paper provides key points to consider to ensure a properly performing SAS environment. This paper is intended to introduce readers to what is possible and to help start conversations about running SAS in a container at your organization. As conversations progress within your organization, SAS and your SAS account team are available to help provide additional information and insight around this deployment option.

Release Information	Content Version: 1.0 June 2021
Trademarks and Patents	SAS Institute Inc. SAS Campus Drive, Cary, North Carolina 27513 SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. R indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

To contact your local SAS office, please visit: sas.com/offices